The Helios Operating System

PERIHELION SOFTWARE LTD

May 1991

COPYRIGHT

Printed in the UK.

Acknowledgements

The Helios Parallel Operating System was written by members of the Helios group at Perihelion Software Limited (Paul Beskeen, Nick Clifton, Alan Cosslett, Craig Faasen, Nick Garnett, Tim King, Jon Powell, Alex Schuilenburg, Martyn Tovey and Bart Veer), and was edited by Ian Davies.

The Unix compatibility library described in chapter 5, *Compatibility*, implements functions which are largely compatible with the Posix standard interfaces. The library does not include the entire range of functions provided by the Posix standard, because some standard functions require memory management or, for various reasons, cannot be implemented on a multi-processor system. The reader is therefore referred to IEEE Std 1003.1-1988, IEEE Standard Portable Operating System Interface for Computer Environments, which is available from the IEEE Service Center, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA. It can also be obtained by telephoning USA (201) 9811393.

The Helios software is available for multi-processor systems hosted by a wide range of computer types. Information on how to obtain copies of the Helios software is available from Distributed Software Limited, The Maltings, Charlton Road, Shepton Mallet, Somerset BA4 5QE, UK (Telephone: 0749 344345).

Contents

Chapter 1

Introduction

All is flux, nothing stays still,
Nothing endures but change.
Heraclitus

This quotation seems appropriate to the world of computing. Every year brings new and faster computers with more memory and better input/output facilities. There are many different measurements of computer power, the most commonly quoted ones being MIPS (millions of instructions per second) and MFLOPS (millions of floating point operations per second). Both of these are increasing for two reasons: greater circuit density and faster clock speed. Circuit density is an indication of the number of electronic components that can be put onto a given chip. The more components a chip has, the more things can happen during a given time interval. (Individual operations become more powerful). For example a typical processor chip has an on-board floating point coprocessor, rather than attempting to perform floating point operations in software. As components become smaller, less effort is required to drive them, and they can change their state faster. This means that the clock speed (the number of operations in a given time interval) can be increased.

1.1 Hardware limitations

The current trend in computing is for an order of magnitude improvement in performance every four or five years. This means that today's computers provide ten times as many MIPS and ten times as many MFLOPS as their equivalents five years ago, equivalence being defined in terms of the price of the computer. In five years, computers should be ten times more powerful. It is useful to consider how long the rate of development can continue.

In computing it is very easy to become somewhat blasé about orders of magnitude. Figure 1.1 illustrates some of these orders of magnitude, for units of time and space. The difference between the time taken for a computer to execute one instruction and the time taken for the seconds digit of an lcd watch to change once is comparable to the difference between an hour and the whole duration of human civilisation. An order of magnitude improvement every four or five years is rather impressive, but can it be sustained ?

Consider circuit density. Today's computers are based primarily on silicon chip technology. The electronic components used to build a computer are embedded in the

surface of a small piece of silicon. Advanced chips use features approximately one micron across: the size of an electronic component on the chip is just a millionth of a metre. A silicon atom has a diameter of approximately $2.35 * 10^{-10}$ metre. Hence an electronic component is about 4000 atoms wide. Atoms are not the smallest building blocks of nature, and it may prove possible to use smaller building blocks at some point in the future to build computers. At present this is pure speculation.

To build faster chips we need smaller components. Suppose for the sake of argument that it will prove possible to use a single atom as an electronic component. Since silicon chips are essentially two-dimensional objects this would give a maximum improvement of $4000 * 4000$, about seven orders of magnitude. The limits of nature will probably prevent us from coming even close to this.

Multiple	Prefix	Time (seconds)	Space (metres)
10^{-12}	pico	light moves 1cm	subatomic particles
10^{-11}			
10^{-10}			one atom
10^{-9}	nano		
10^{-8}		one instruction	
10^{-7}		one floating point operation	
10^{-6}	micro		feature on current chip
10^{-5}			
10^{-4}			
10^{-3}	milli		easily visible to eye
10^{-2}			
10^{-1}			size of floppy disc
10^{0}		one second	one metre
10^{1}			
10^{2}		minute	large building
10^{3}	kilo	$1/4$ hour	ten minutes' walk
10^{4}			
10^{5}		day	
10^{6}	mega	week	radius of earth
10^{7}			
10^{8}			
10^{9}	giga	human lifetime	distance to moon
10^{10}		1000 years	
10^{11}		human civilisation	
10^{12}	tera	homo sapiens	size of solar system

Table 1.1: Approximate orders of magnitude

Next, consider clock speed, another important factor in processor performance. Suppose that a single operation involves one signal moving from one end of a chip to the other end, again somewhat of a simplification. Today's chips are typically about a centimetre across. A signal travelling at the speed of light will take about 30 picoseconds to move this distance. Hence a processor could perform 30 thousand million such operations every second, corresponding to a 30000 MHz processor, just three orders of magnitude faster than today's chips.

These calculations are by no means perfect. For example, it may be possible to start building three dimensional chips instead of two dimensional ones, and the average size of a chip may shrink below one centimetre as circuit density increases. The calculations ignore quantum effects that become significant for small numbers of atoms, as well as heat dissipation problems for such tightly packed electronics. However, the implication is that advances in the current technology will cease after another fifteen to thirty years, with processors somewhere between a thousand times and ten million

times more powerful than today's.

1.2 Actual requirements

> Power tends to corrupt, and absolute power corrupts absolutely.
> *Acton*

An obvious question to ask at this point is what all this computing power will be used for. A single processor will provide somewhere between a gigaflop and a teraflop of performance. Are there really problems which need such power? More important, are there problems which need even more ?

Predicting a few years ahead in the field of computing, let alone fifteen or thirty years, is a risky business. However, the answer to both of the above questions is a resounding "Yes". Even today there are problems in science and engineering which require more computing power than the limits of nature appear to allow. These include, but are not limited to:

1. Quantum chemistry. It has been known for some time that the behaviour of atoms and molecules in chemistry is defined by Schrödinger's equation.

   $$d^2/dx^2\Psi(x) + 2m/\hbar(E - V)\Psi = 0$$

   This equation has been solved fully only for the simplest problems. Hence chemists are forced to work with computer models that generate numerical approximations. Current models are limited to fairly simple molecules and small numbers of atoms. More computing power would allow slightly more complex models.

2. Cosmology. One cosmic-sized problem is attempting to work out how the universe could start from a big bang and end up looking the way it does today, matching the astronomical data. Other problems in cosmology involve looking at smaller objects than the entire universe such as galaxies, quasars, black holes, stars and solar systems.

3. Fluid dynamics. This involves examining the behaviour of gases and liquids in the vicinity of solid objects such as pipes and the wings of aeroplanes.

4. Materials science. This requires the modelling of solid objects such as the materials used to build car engines, and hence being able to design better ones.

5. Biology and biochemistry. In particular, analysing the sequences of the DNA molecules
   that define our genetic make-up.

6. Weather forecasting and longer term global climate modelling.

7. Processing information. By the end of the 1990s, the various satellites in earth orbit are expected to produce a terabyte of data every day, which should be processed somehow.

8. Artificial reality, building realistic computer models of this and other worlds, and allowing humans to interact with these models in real time.

9. Artificial intelligence and artificial life, reproducing the behaviour of biological systems and hopefully improving on them.

Most of these problems have one thing in common: they are essentially open ended. Providing more computing power simply allows the scientists to build larger, more complicated, and presumably more accurate models. Each improvement should give more useful data, but there will not be a definitive solution. Already there are plans to build teraflop computers to meet these needs, and pentaflop (10^{15}) and exaflop (10^{18}) computers would be gratefully received by the scientific community.

Science may be a driving force for supercomputer development, but the needs of ordinary personal computer users must also be considered. It may seem unlikely that word processing, spreadsheet, and database applications will need processors much more powerful than today's. However, as more features are added to existing applications as new applications are added, and as the underlying system software becomes more flexible, even ordinary personal computers will need ever more MIPS and ever more MFLOPS for some time to come.

1.3 Architectural improvements

> The wondrous architecture of the world.
> *Marlowe*

Given that we cannot rely on scientific breakthroughs to produce the sort of performance we are going to need, is there anything that can be done at the computer architecture level to achieve the required speed-ups ? It is often said that existing computers are based on the classical von Neumann design. There is a central processing unit or CPU with some memory and I/O devices attached to it. The CPU reads an instruction from memory, executes it, and then reads the next instruction from memory. Typical instructions move data from one place to another, test the value of a piece of data, transfer control to some other location, or perform arithmetic on some data. Only one instruction at a time gets executed, and hence the computer is said to run sequentially.

In practice this purely sequential architecture did not last very long. I/O operations such as punching a paper tape took much longer to execute than ordinary instructions, so computer architects designed their computers to perform I/O in parallel with the main stream of execution. The CPU initiates an I/O operation and, some time later, it either polls the device to see whether the operation is finished or it receives an interrupt. Hence there is computation and I/O occurring in parallel.

Some floating point operations take a much longer time than their equivalent integer operations. Hence it is useful to have a separate floating point processor working in parallel with the main CPU, controlled by the CPU. A floating point operation is initiated and, some time later, the CPU checks whether or not the operation has finished or it gets informed when the operation has finished. The hardware may do this automatically. To make full use of a separate floating point processor requires some extra work in the compiler.

Vector processors take this concept a step further. Instead of there being one floating point processor there are many, typically 64 or so. All the floating point processors perform the same operations at the same time, but on different data. Typically this data consists of matrices and vectors, where the different parts of the matrix can be manipulated separately. A great deal of work is required in the compiler to be able

to detect when different bits of data can be operated on in parallel, and, except when dealing with matrices and vectors, it is difficult, if not impossible, to make efficient use of a vector processor.

A single instruction such as adding two numbers can be subdivided into several different stages: fetching the instructive code, fetching the data, performing the arithmetic and storing the results. An instruction pipeline exploits this by performing the stages of several instructions in parallel. Instruction **n** stores its result while instruction **n + 1** does some arithmetic and instruction **n + 2** fetches data. Since programs contain many branch instructions and each instruction can be divided into only a limited number of stages, there are limits to the practical length of a pipeline.

Memory caches are another way of speeding up processors. It is common for current processors to be able to work significantly faster than the main memory, resulting in a memory bottleneck. To overcome this problem, fast memory caches can be used. Instead of all memory accesses going to the external memory, there are one or more cache units between the CPU and memory. These cache units can work faster than the external memory, and contain the contents of frequently accessed memory locations. Cache memory is expensive, so there are limits on the amount of cache that can be put on a processor.

A fairly recent development in microprocessor technology is Very Large Instruction Word processors or VLIW. With these processors a single instruction no longer contains a single operation, but several. For example, a single instruction could contain an integer addition, a floating point operation, and a conditional jump. The CPU contains several units, including one or more integer arithmetic units, floating point units, and a control flow unit. Keeping all of these busy requires a great deal of effort in the compiler. Furthermore it is very difficult to keep all the units busy. For example, if the CPU contains ten different integer arithmetic units then it is most unlikely that any normal piece of code could be compiled to use all these units at the same time.

Putting all these features together, we can foresee a single processor on one chip with the following features:

1. A main CPU containing several parallel units, typically two integer arithmetic units, between one and 64 floating point units depending on whether or not the CPU is intended for vector processing, and a control flow unit.

2. An instruction pipeline executing different instructions.

3. There is an instruction able to hold 64K or more of the currently active programs, as well as one or more data caches occupying between 64K and a megabyte each.

4. Hardware support for special operations such as signal processing and graphics operations, because these use up much of the CPU time in existing processors.

5. Asynchronous I/O support, requiring a minimum of effort by the processor.

6. All the units making up the processor (the main CPU and the supporting hardware) work in parallel. A certain degree of synchronisation between the different units is required, for example a CPU cannot execute an instruction unless it is has been fetched by the pipeline, which in turn cannot fetch it unless it is in the instruction cache.

Exploiting all of these features may produce a single CPU that is perhaps one or two orders of magnitudes faster than the conventional von Neumann architecture. Undoubtedly there will be further developments at the computer architecture level, such as multiple instruction streams and self timed (asynchronous) CPUs, which will provide some extra speed-ups. However, existing processors already use many of these features. Hence such developments in CPU architecture cannot by themselves provide the required improvements in performance. It is necessary to look elsewhere for a solution.

1.4 Parallelism

> Many hands make light work.
> *Heywood*

A single processor cannot provide the required performance. This leaves the possibility of using more than one processor to solve a single problem, the field of parallel processing. A state of the art processor can provide between 10 and 100 megaflops. Hence if we can build a machine with between 10000 and 100000 such processors, we have a teraflop computer. With processors a thousand times faster, and with ten million or a hundred million such processors, we could build an exaflop computer. However is it really possible to have ten processors working on the same problem, let alone ten thousand or ten million ?

For some problems, fortunately including many of the scientific problems described earlier, it is possible to answer affirmatively to at least part of this question. Currently it is fairly common to solve scientific problems on some tens of processors, and machines with some hundreds of processors are being installed. Teraflop machines with tens of thousands of processors are at the design stage, and there are no major problems at the hardware level building such machines. However, producing software to control and run on such machines can still be quite difficult.

Merely taking some hundreds of conventional processors, together with some memory and I/O facilities, is not sufficient to produce a parallel machine. An analogy is appropriate. Consider a team of human programmers, working together to produce a large software system. Given sufficient time a single programmer could produce the whole system, but the job would usually take far too long and the system would be out of date by the time it was finished. Instead, a team of programmers are made to work together to build the system. These programmers cannot work independently from each other. Every programmer must produce some part of the system, which will work with the parts produced by other programmers to give a working system. Every programmer must collaborate with his or her colleagues to ensure that the various parts will fit together, or the final system cannot work. In other words, the programmers must **communicate** with each other. The amount of communication, and the way the communication is organised, will vary from system to system and from company to company. It may be sufficient merely to exchange specifications when the project starts. It may be desirable to have regular daily meetings, or to have meetings only when considered necessary. The programmers may interact directly, or they may have to go through a chain of command. The exact details vary, but some amount of communication will be required. When not communicating, the programmers can work independently from each other developing their code.

The same is true of applications running on a parallel machine. In the simple case every processor will run some piece of code responsible for solving part of the problem. Each piece of code is one component of the application. Components must communicate with components running on other processors, to exchange data. The amount and nature of the communication may vary. For some applications it is sufficient for the various components to get some data when they start up, and share results when they finish. For other applications large amounts of communication are required for every step in the calculation. The various components may interact directly, or they may communicate only via some master component. Unless the components can communicate somehow they cannot work together on the same problem.

Every processor within a parallel computer must satisfy two primary requirements. They must be able to do computation, for example floating point arithmetic. They must also have some means of communicating with each other, otherwise the various processors in the parallel computer cannot work together to solve a problem. Different parallel machines vary in the ways that communication is achieved, and in the relative speeds of computation and communication.

1.5 The Transputer

The Inmos [1] Transputer family comprises a number of processors particularly appropriate for building parallel computers. Every processor contains a number of **links**, serial lines providing fast communication between processors. The processors most commonly used are the T800 and the T805, which have floating point hardware as well as four communication links, thus providing fast computation as well as communication.

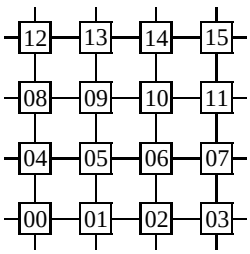


Figure 1.1: A Transputer network

Consider Figure 1.1. This shows a network of 16 Transputers, each represented by a single block. Each Transputer contains a conventional CPU, capable of integer and floating point arithmetic, conditional branches, and all the instructions you could expect in a processor used to build a sequential computer. Each Transputer also has four links, and most of these links are connected to links in other processors. A software component running on processor `05` in the diagram could communicate directly with components running on processors `01`, `04`, `06`, and `09`. However, if this component needs to communicate with a component on processor `15` then life becomes more complicated. The two processors are not directly connected, so it is necessary to

[1] Inmos and Transputer are trademarks of the Inmos group of companies

somehow route messages through various processors from source to destination. Also, it is necessary to start all the components on the various processors, set up the communication between them, and so on. Having communication support in the hardware does not solve all the problems in building a parallel computer.

The network shown in the diagram is configured in the form of a grid. In hardware terms such grids can be extended fairly easily in all four directions, to produce an arbitrarily large parallel computer. In practice there are limits on the size of such a computer, depending in part on the applications to be run on it. Consider a machine of 10000 such Transputers, in a grid of 100 by 100. To route a message from one corner of the grid to the opposite corner involves going through 198 links, and hence requires some CPU time in 197 processors. If there is too much such communication the parallel computer will spend nearly all its time routing messages rather than performing useful computation. Different applications will vary in the amount of such communication. Using different network configurations may reduce the problem, but will not eliminate it.

When Transputers were first released, the only software support was the **occam**[2] language. This language is specifically designed to run on Transputers, with built-in support for communication exploiting the Transputer links. However, the current version of the language, /bf occam 2, contains no support for routing messages through a network, required by many applications. Starting up all the components on all the right processors, setting up the communication between them, and so on are jobs left essentially to the programmer. Furthermore occam is rarely used other than for programming Transputers. Most programmers are familiar with languages such as C and Fortran, and do not want to learn another programming language. Most programmers are familiar with a particular programming environment, typically some version of the Unix [3] operating system, and wish to continue using such environments on parallel computers. To address some of these issues, additional software is required.

It should be noted that the Transputer family is relatively old in computing terms. The first Transputers became available in 1984. At the time of writing Inmos have announced a new family of processors (the T9000 series) offering significant improvements in performance, both computation and communication, as well as hardware support for message routing. On the other hand the T9000 has new features such as limited memory management, which normally require an operating system to exploit them. Hence the need for additional software remains.

1.6 Helios

> Then in all the world they do their work.
> *Akhnaton's hymn to the sun*

The Helios[4] Parallel Operating System has been designed to run on parallel computers. Such computers contain processing units, and fast communication between the processors. Many such parallel computers are built using Transputers, and Helios runs on these machines. However, Helios also runs on parallel computers built using processors other than Transputers. This book describes some of the aspects of Helios.

The design goals of Helios are ambitious.

[2] occam is a trademark of the Inmos group of companies
[3] Unix is a registered trademark of AT&T
[4] Helios is a trademark of Perihelion Software Limited

1. To provide a general-purpose operating system for parallel computers, independent of any specific hardware.

2. To provide an operating system with a very high degree of compatibility with existing systems, by supporting international standards such as Posix[5].

3. To provide a development environment that will be familiar to existing programmers, so that programmers do not have to learn new ways of using a computer merely because it is a parallel computer instead of a sequential one.

4. To allow parallel applications to be developed using conventional programming languages such as C or Fortran, without the need to learn new languages or programming constructs.

5. To allow such parallel applications to run with the greatest amount of efficiency consistent with the other design aims.

6. To allow such applications to be moved from one parallel computer to another, quite possibly based on a completely different family of processors, with a minimum of effort.

7. To provide a high degree of fault tolerance. The system as a whole must be able to recover from the failure of any one software component or piece of hardware, subject to physical limitations imposed by the hardware itself.

8. To be independent of the number of processors in the network. Parallel applications can be developed on a single processor if desired, and then run unchanged on several hundred processors.

Work began on Helios in the autumn of 1986. It is a new operating system, not a re-write of some previous system, although obviously some parts of Helios incorporate ideas developed in other operating systems. The first commercial release was Helios 1.0, released in the Summer of 1988. This was followed by 1.1, Autumn 1989, and 1.1A, an upgrade to 1.1 shipped in early 1990. Helios 1.2 was shipped in December 1990, again followed by an upgrade some months later. Helios 1.2 supports a very high degree of Unix compatibility, large processor networks of some hundreds of processors, and it allows multiple users to share such large machines. At the time of writing work is proceeding on the next release, Helios 1.3, and this book is intended to accompany that release. Most of the book is relevant to earlier versions, and will be appropriate to later versions also.

1.7 Target hardware

> We aim at the infinite.
> *O.W. Holmes Jr.*

Helios has been designed to work with a wide range of machines. One such machine is a parallel mainframe computer. Such a machine would contain hundreds of processors, and would support tens of users logged in at once. Some users would be running

[5] Posix refers to the standard defined by IEEE Standard 1003.1-1988

large parallel applications, which together would use up most of the available processors. Other users would be developing parallel applications, developing ordinary sequential applications, sending or reading electronic mail messages, or even playing games. Most users would access the machine via a local area network, typically Ethernet [6], and the machine itself would be in a separate air-conditioned room maintained by computer operators. Some users might plug in their own private networks of processors, in order to make use of the larger number of processors. To avoid I/O bottlenecks such large parallel machines must be equipped with a number of fast hard discs, and tape units for backup purposes and for holding very large amounts of data.

Imagine a different type of parallel machine also running Helios. Consider an automated factory floor, with large numbers of devices such as robot arms performing the work, and various metres to monitor what is happening. Some devices need several processors to control them, while some other processors could control several devices. The various processors need to communicate and exchange data. For example a processor controlling a robot paint spraying arm needs to be informed when the object to be painted is ready. In effect all the processors controlling pieces of hardware can be thought of as a parallel computer that happens to spread over a factory floor rather than being contained inside just one box.

A third Helios system is shown in Figure 1.2. The diagram shows a workstation, complete with high resolution graphics display, hard disc, and local area network connection. Helios has been designed to provide a high degree of compatibility with Unix systems, so it supports the X Window System [7] for graphics and ethernet software such as remote login facilities and network file systems. The workstation shown has all the I/O devices attached to just one processor, making it equivalent to a conventional workstation. Alternatively the I/O devices could be attached to different processors, one to handle the graphics display, another to perform disc I/O, and so on, offering better performance at a greater cost. If desired it would be possible to connect this workstation to a larger network of processors, some tens or even hundreds of processors, probably over a period of time, and thus turn the workstation into a supercomputer. All existing software will continue to run, and parallel applications simply run faster as more processors are added.

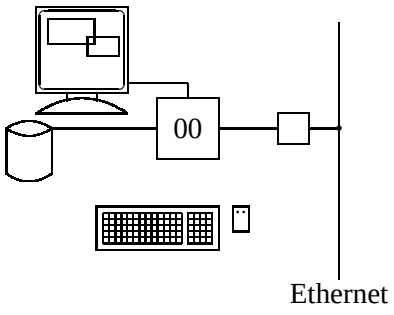


Figure 1.2: A workstation

[6] Ethernet is a trademark of Xerox Corporation
[7] The X Window System is a trademark of MIT

1.8 About this book

> Another damned, thick, square book!
> *William Henry, Duke of Gloucester*

The purpose of this book is to give a description of much of the Helios parallel operating system. It does not attempt to describe every single feature, command, or library routine provided by the system, because that would require several books of this size. Instead the book concentrates on the main components of Helios. The book is aimed at users of Helios and also at anyone with an interest in parallel computing generally.

The book is divided into sixteen chapters, including this introduction. Each chapter has been written as a self-contained unit, and can be read independently from the others. This book does not describe the full range of Helios commands. Comprehensive information can be obtained in *The Helios Encyclopaedia*, available from Distributed Software Limited, The Maltings, Charlton Road, Shepton Mallet, Somerset BA4 5QE, UK.

Chapter 2

Networks

2.1 Introduction

The purpose of this chapter is to describe all of the aspects of the Helios networking software. This is not an easy task, because Helios runs on a wide range of machines. At the bottom end of the range would be a single PC plug-in board, typically with a single T800 Transputer and perhaps two megabytes of memory. At the top end would be large multi-user networks of 400 or more Transputers, with perhaps twenty or so users at any one time accessing the network in a variety of ways. Networks may consist of hardware produced by several different manufacturers.

To control all the different types of network Helios uses a single set of programs: the networking software, in conjunction with a number of configuration files. Section 2.2 of this chapter gives an outline description of the various programs and configuration files, using a simple network as an example. It describes how this network starts up, and how a user can access the resources in the network.

Section 2.3 gives a description of the various types of network, varying from a single-processor system to a large supercomputer. This section describes which components of the networking software should run and why. The exact details of configuring such a network are left to section 2.7: 'Configuring networks'.

Designing hardware or choosing the right hardware to buy involves compromises. A link running at 20 MHz will transfer data faster than a link running at 10 MHz, but for a shorter distance. A crossbar link switch allows flexibility in setting up the network, but causes a delay when transferring data. Section 2.4 describes some of the hardware produced by various manufacturers, with an emphasis on how the hardware affects the networking. This section also gives an introduction to the topic of network topologies (and how they can affect performance).

The networking software contains a considerable number of commands which interact with the various servers. These commands are explained in section 2.5

Section 2.6 describes the various configuration files in detail, in particular the network resource map and how it is affected by the hardware. This information is then used in section 2.7, which repeats most of the networks described in section 2.3 and shows how to configure them.

2.2 The components of Helios

This section describes the various components of Helios (including the networking software) and how they interact. The simple network shown in Figure 2.1 will be used as an example.

2.2.1 A simple network

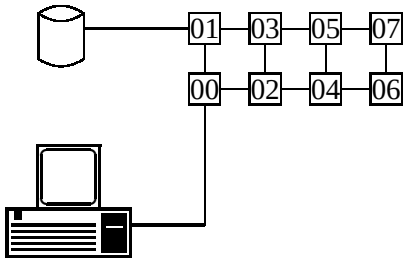


Figure **2.1** A simple network

This network consists of eight Transputers, labelled `00` to `07`. Each Transputer is a microprocessor with its own private memory, which is not accessible by the other processors. It is recommended that each processor has at least one megabyte of memory, although applications may well require more than this. Each Transputer is equipped with four **links** which are used by Helios to achieve fast communication between the processors. The four links are generally referred to as link 0 to link 3, and in diagrams link 0 is conventionally the bottom one as shown in Figure 2.2.

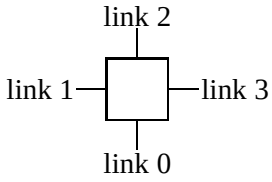


Figure **2.2** Link numbering

Like any other microprocessor, a Transputer has to be **booted** with some software before it can perform useful work. Transputers have two bootstrap mechanisms: ROM bootstrap, where the software is held in 'read only' memory; and link bootstrap, where the software is sent down the link by a neighbouring processor. ROM bootstrap involves extra hardware and hence is rarely used. Before a processor can be booted, it must be **reset**. On most hardware a processor is automatically in a reset state when it is powered up, but not always. The hardware must provide some other reset mechanism which can be used by the networking software. In addition to the eight Transputers, the network contains a **host processor** or **I/O processor**. Typically this would be a Sun workstation or an IBM[1] PC or compatible, but a wide variety of machines can

[1] Registered trademark of International Business Machines, Inc.

be used. When the whole network is powered up the I/O processor will usually be booted with the host operating system. For a Sun workstation this would be SunOS[2], a version of Unix, and for a PC this would be MS-DOS[3]. A program, the I/O server, can then be run on the I/O processor and initiate the bootstrap of the Transputer network. Frequently the I/O processor also serves the rather useful role of power supply for the whole network.

The I/O processor is not (usually!) a Transputer and hence it is not naturally equipped with Transputer links. Since links are needed for both communication and bootstrap it is necessary to add some special hardware, a link adapter, to the I/O processor. For example, a typical PC plug-in board such as the Inmos B008 contains a C012 link adapter. It should be noted that the link adapter can be a bottleneck for many applications. In theory a 20 MHz link can be used to transfer up to 1.70 megabytes per second. Helios can achieve 1.62 megabytes/second using Posix-style **read()** and **write()** calls acting on pipes. By contrast a typical link adapter can achieve between 50 and 200 kilobytes per second, just 10 percent of this speed. Since the I/O processor may have to service the I/O requirements of a considerable number of more powerful processors, another factor is the processing speed of the I/O processor. The network in Figure 2.1 attempts to solve part of this bottleneck problem by attaching I/O hardware, in this case a hard disc, directly to a Transputer. This I/O facility cannot be used until processor `01` has been booted and some additional software, probably the Helios filing system, starts running on that processor. After this initial hurdle the hard disc can be used to bypass both the communication bottleneck of the link adapter and the processing bottleneck of the I/O processor. Of course there is a price to be paid. In addition to the cost of the extra hardware, some of the processor's CPU time and some of its memory will be taken up by the filing system.

Processors in a network are frequently referred to as **network nodes**. This matches mathematical graph theory, where the processors are nodes or vertices and the links are the graph edges. Graph theory is commonly used when designing or analysing Transputer networks.

2.2.2 The Helios naming scheme

Before describing individual components of the Helios software it is necessary to describe the Helios naming scheme. This is illustrated in Figure 2.3.

At the top of the naming tree are one or more levels of **network** names. In the example network `/Cluster` constitutes the root of the naming scheme. Below the network level or levels is the **processor** level. Every processor is given a name when it is booted. For non-trivial networks it is conventional to use simple numbers, but there is nothing to stop the user from configuring the network with processor names such as `tom`, `dick`, `harry`, `fred`, `john`, and so on. Below the processor level is the **server** level. For example, processor `00` is shown with two servers, **tasks** and **ram**, and will run several others as well. A server provides a **service** of some sort. For example, a file server provides a file I/O service, and a logger server provides an error logging service. Usually each server is a separate program, although it is possible for a single program to act as more than one server.

[2]Trademark of Sun Microsystems

[3]Registered trademark of Microsoft Corporation

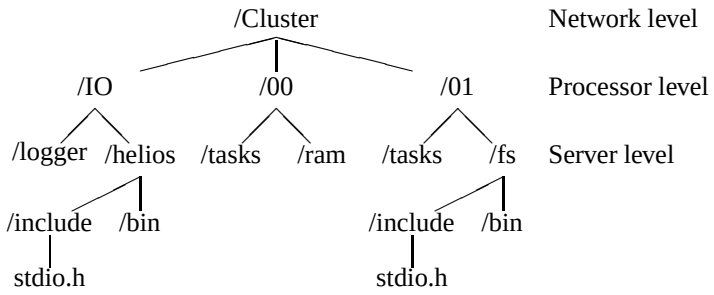


Figure **2.3** The Helios naming scheme

Below the server level is an ordinary directory level. For example, the file server called **/Cluster/IO/helios** contains ordinary directories **include** and **bin**, and the **include** directory contains a **sys** subdirectory and a file **stdio.h**. However, at all levels of the network hierarchy the same protocols are used. Consider the **ls** command, which is used to list the contents of a directory. The command **ls /Cluster/IO/helios/include** would produce a normal directory listing, as expected. The command **ls /Cluster/00** would produce a listing of the servers running in processor `00`.[4]

In the diagram there are two files **stdio.h**, held in two separate file servers. These have different network addresses: one of them is **/Cluster/IO/helios/include/stdio.h**, and the other one is **/Cluster/01/fs/include/stdio.h**. This shows that **all** objects in the naming scheme have a unique network name. Incidentally, it is the responsibility of the user to ensure that these two files are consistent, and that a C compiler could use either of them. It is likely that the C compiler will be made to use the **/01/fs** version by preference, because using this file server avoids the bottleneck of the I/O processor's link adapter.

For a variety of reasons, not least of which is the tediousness of typing in long names, it is not always necessary to give full network names when accessing an object. The minimum that must be supplied is the server name plus the full path within that server.

For example, **/helios/include/stdio.h** will work, but **/stdio.h** will not because it does not contain enough information. This introduces the possibility of ambiguity. For example, there might be two servers called **/helios** in the network, one in the I/O processor and the other one on processor `01` with the file server installed as **/helios** instead of **/fs**. In such a case accessing **/helios/include/stdio.h** will usually access the nearest server, which is not necessarily the correct one. Ambiguity can always be resolved by giving more or all of the full network name of an object. Users can provide their own services if required, by writing new server programs that understand the protocol used between Helios clients and servers: the **general server protocol**.

[4] Strictly speaking this is not correct. For efficiency reasons every processor maintains a table of known names of servers and processors, and the **ls** command would list the appropriate entries in this name table. Hence any names not yet known, because they have not been accessed yet, would not appear in the directory listing.

This is not always easy, and many users of Helios can work perfectly happily using just the standard services provided.

After this discussion of Helios naming it is time to consider the first piece of Helios software that must be run to boot the processor network.

2.2.3 The I/O server

The I/O server is the main and often the only piece of Helios software that runs on the host processor and not on a network processor. All other software such as compilers, shells, network management, and user applications, run on the network processors. The I/O server has two main jobs: booting the first network processor or **root processor** and providing various services to allow access to the resources of the I/O processor.

The I/O server is a large but flexible piece of software. Flexibility is achieved by reading in a configuration file, **host.con**, when it starts up. Options include the exact nature of the link adapter hardware, the location of the Helios system files, and whether or not particular services such as a mouse device should be provided. Four **host.con** options affect the networking software: **root_processor**, **io_processor**, **bootlink** and **enable_link**.

Every processor in the network needs a name when it is booted up, and this includes the root processor. The I/O processor forms part of the Helios network, so it too needs a name. These names can come from the **host.con** files, although the I/O server will use default names **/00** and **/IO** if the entries in **host.con** are missing. The **bootlink** option specifies which link on the root processor is connected to the I/O processor, usually but not always link 0. The **enable_link** option is used to connect into a running network rather than to boot up a network, if the I/O processor does not contain its own private processor but just a link adapter. The first job of the I/O server is booting the first Transputer in the network. This requires several stages.

1. Carry out any hardware initialisation necessary to start up the link adapter.

2. Reset the root processor, which may have the side effect of resetting some or all of the rest of the network at the same time.

3. Send in a small bootstrap utility **nboot.i**. This performs some hardware initialisation on the Transputer side and then waits for further instructions from the I/O server.

4. Instruct the bootstrap utility to read in the **system image** or **Nucleus**, and send in this Nucleus. The bootstrap utility now transfers control to the first component of the Nucleus, the **Kernel**.

5. Send in some additional configuration information needed by the Kernel.

After these bootstrap stages Helios is up and running on the root processor, and the I/O server now takes a passive role. In particular it starts up a number of servers, providing various I/O facilities for the Helios network. Exactly which servers will be available depends on the host machine. For a PC host a typical list would be:

- **/logger**, an error logging service.

- **/window**, a pseudo-windowing system which provides multiple full-screen windows and a hot key switching mechanism to move between windows.
- **/helios**, a file server providing access to the main Helios files.
- **/a, /c, /d**, additional file servers for the various disc drives **a:**, **c:**, and **d:**.
- **/rs232**, access to the PC's serial ports.
- **/centronics**, access to the parallel ports.
- **/pc**, a limited communication facility between programs running under Helios and programs on the PC.

On a Unix host the list might be:

- **/logger**, an error logging service.
- **/window**, multiple real windows, using an X window display.
- **/helios**, a file server providing access to the main Helios files.
- **/files**, a file server for the whole of the Unix filing system.

All of these servers are part of the Helios network, within processor **/IO**. It must be emphasised that, following the bootstrap, the I/O server is purely a passive object. It waits on the link adapter for incoming requests (to read data from an open file, for example), it services these requests, and sends replies back into the Transputer network. All 'intelligent' software such as shells, compilers, networking software, and users' applications, runs on the Transputer network.

It is not essential to have an I/O processor in the network. The alternative is to have a ROM based system, where one processor in the network executes a ROM bootstrap when the system is powered up. Typically such a bootstrap routine would read a Helios Nucleus from the first track of a hard disc and transfer control to this Nucleus.

2.2.4 The Nucleus

The Nucleus is the part of Helios that is present on every processor in the network. The I/O server boots the Nucleus into the root processor, and some time later the networking software boots a Nucleus into every other processor. The Nucleus consists of six parts: Kernel, System library, Server library, Utility library, Processor Manager, and Loader. The relationship between these and the rest of Helios is shown in Figure 2.4.

- The Kernel is responsible for the processor hardware. On a Transputer this involves monitoring the links and the event line. In particular, the Kernel has link guardian processes for every link connected to another processor running Helios, waiting for messages sent from that processor and forwarding them to the appropriate destination (possibly another link). In addition, when the Kernel starts up it detects the amount of memory in the processor and initialises the memory

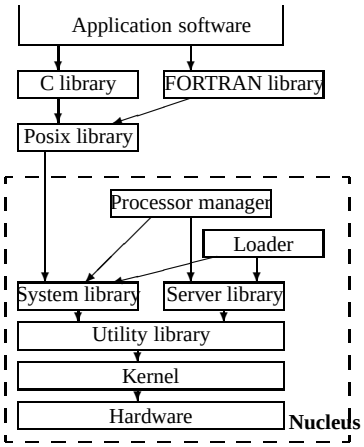


Figure **2.4** The Helios Nucleus

allocation system. The Kernel provides low-level calls such as message passing, semaphore synchronisation, and creating and destroying processes. Most of these involve atomic operations, which means running at high priority on a Transputer or in supervisor state on other processors. Normal application programs rarely if ever need to use Kernel calls, since higher-level library routines usually provide the same functionality and are much easier to use.

- The System library provides the basic interface between clients and servers. It contains library routines such as **Open()**, **Read()**, and **Seek()**. The System library routines take their arguments and pack the required data into messages, following the format defined by the General Server Protocol. The System library then sends the message to the appropriate server and usually waits for a reply. The server receives the message, acts on the request, and sends back a reply. At the System library level it is irrelevant whether the client and the server run on the same processor or different ones, which is not true of the Kernel level.

  The System library also provides a number of more specialised routines such as **Execute()** to run another program, which actually just involves more specialised interaction with servers. Application programs rarely need to use System library calls, because higher-level libraries such as the Posix library are generally more convenient.

- The Server library exists to make it easier to write Helios servers. It contains code to maintain directory structures within memory, to handle automatically many incoming requests, and to support buffering of data. Even with the Server library writing Helios servers is a difficult operation, and should not be attempted lightly.

- The Utility library provides a number of library routines that have to be in the Nucleus but did not ‘belong’ in one of the previous libraries. This includes routines to manipulate areas of memory, such as **memcpy()** and **strcpy()**.

- The Loader is a Helios server, a program rather than a library, which takes care of pieces of code loaded into its processor, and which ensures that code is shared between programs where possible. When a program has to be run on a processor an entry is created inside the **/loader** directory on that processor and the code is fetched, usually from a disc. If the program makes use of Resident libraries not currently in memory, these libraries are fetched automatically.

- The Processor Manager provides another Helios server, **/tasks**, which allows clients to run programs or tasks on that processor. It takes care of signals sent to a particular task, such as the SIGINT signal if the user presses control-C, and ensures that any resources used by a program are freed when the program exits. The Processor Manager performs several other necessary functions such as keeping an accurate time of day in the absence of a real-time clock and maintaining name tables for that processor.

The above six parts are present in every Helios Nucleus. It is possible to add additional servers to the Nucleus. For example, in a system booted from ROM rather than from an I/O processor the Nucleus would have to contain the Helios file server, so that additional bootstrap files could be read from the disc. The Processor Manager performs one additional function not listed above. If the processor has been booted by an I/O processor or from ROM, the Processor Manager will execute the program **/helios/lib/init**. This program must perform the next step of the bootstrap process.

2.2.5 The Init program

The initial bootstrap stage is similar for all networks and all applications, and involves getting a Helios Nucleus up and running on one processor in the network. Following this stage different applications have very different requirements: a work station system should get a windowing system up and running and start a login session; a factory control system must boot up the network and run appropriate software on specific processors. To allow easy configuration of the system, Helios runs the **init** program and this in turn reads **/helios/etc/initrc**, which is the text resource file. Changing this text file allows the user to perform much of the system configuration. A typical **initrc** file might look like this.

```
#
# This is a comment line
#
# First, set up the windowing system
ifabsent /window run -e /helios/lib/window window
console /window console
#
# Then start the networking software
run -e /helios/bin/startns startns -r /helios/etc/default.map
#
# Wait for the Session Manager to be active
waitfor /sm
#
# And start a user session
run -e /helios/bin/newuser newuser mary
```

An early design decision for Helios was to provide multiple windows wherever possible. These could be real windows on a graphics display, either attached to a Transputer or the I/O processor. Alternatively a pseudo windowing system with hot key switching may be used. The **window** server may be part of the I/O server or it may have to be loaded into the processor network. The first lines of the **initrc** file deal with these two cases.

The test **ifabsent /window** will fail if the **window** server already exists, probably as part of the I/O server, so no further action is needed. If the window server is absent a window server is loaded from the disc and started up. This might be a terminal emulator running under the X window system, or some other windowing system. The next line, **console /window console**, creates a new window called **console** inside the specified server. From now on this window will be used as the standard stream for all subsequent commands, instead of the error logger.

Following the initialisation of the windowing system the initrc file specifies execution of the command called **startns**. **startns** is short for Start Networking Software, and this command starts up the Helios network server which forms the backbone of the networking software. The network server is responsible for booting up the whole processor network, for allocating processors to users, and for monitoring the network and ensuring recovery when individual processors are crashed. The **startns** command also starts up a separate program, the Session Manager, responsible for starting user sessions.

The line **waitfor /sm** suspends the bootstrap process until a server with that name appears. **/sm** is the Helios name for the Session Manager, so in this case the bootstrap is suspended until the Session Manager is up and running. Note that this is not the same as waiting for the whole network to be booted. Booting up a network can take many seconds, depending on the number of processors, the configuration, and the hardware.

The final line executes the **newuser** command. This command interacts with the Session Manager and requests it to start a new session for the user 'Mary', in the current window. Assuming the system is suitably configured for a single-user system, the Session Manager will not require a logging-in phase. Instead it starts up a separate program, a Task Force Manager, to handle the session and execute commands for that user.

2.2.6 The network server

The network server constitutes the backbone of the Helios networking software. It has a number of different jobs.

- Initial bootstrap of the whole network.
- Control of the network hardware, particularly the reset and link configuration hardware.
- Allocating processors to users as and when required.
- Monitoring the network for errors such as crashed processors, and attempting to recover from such errors by resetting and rebooting the crashed processor, if the hardware allows.

The network server is never started up directly by the user. Instead there is a separate command **startns** (see the **initrc** file described in the previous section). This program reads in a network resource map, defining the network. For the network shown in Figure 2.1, the text form of the resource map might be:

```
subnet /Cluster {
  Reset { driver; ~00; tram_ra.d}

  processor 00 { ~IO,    , ~01, ~02; }
  processor 01 { ~00,    ,    , ~03; run -e /helios/lib/fs fs scsi; }
  processor 02 {    , ~00, ~03, ~04; run /helios/lib/lock; }
  processor 03 { ~02, ~01,    , ~05; }
  processor 04 {    , ~02, ~05, ~06; }
  processor 05 { ~04, ~03,    , ~07; }
  processor 06 {    , ~04, ~07,    ; }
  processor 07 { ~06, ~05,    ,    ; }
  processor IO { ~00; IO }
}
```

In this map the network as a whole is given the name `/Cluster`. The network consists of nine processors. One of these, `/Cluster/IO`, is an I/O processor. The other eight, which are `/Cluster/00`--`/Cluster/07`, are assumed to be Transputers. In addition there is one line indicating the hardware facilities available to the network server for resetting processors in the network.

Looking at the individual processors in more detail, all processors are shown with their link connections. Consider processor 00: link 0 is connected to the I/O processor; link 1 is not connected; link 2 is connected to processor 01; and link 3 is connected to processor 02. The connections for the other processors are specified in the same way. In addition, when processor 01 is booted up the network server will run the program **/helios/lib/fs** on that processor, using the arguments specified with the same syntax as the **initrc** file.

The resource map shown here is fairly simple, as is the network it represents. The full syntax of resource maps is given in section 2.4, and section 2.5 contains resource maps for a wide range of networks.

The **startns** program reads in the resource map and a separate configuration file **nsrc**, starts up the network server, and sends the information to the network server. The **nsrc** file controls whether or not optional facilities such as password checking are enabled. The network server installs itself as the server **/ns**, receives the network details from startns, initialises any machine-specific hardware such as link switches, and boots up the network. The network server proceeds to monitor the network for failures and handles requests such as allocating a set of processors to a user.

The network server, like all Helios servers, understands the General Server Protocol. This means that commands such as **ls** will work on it. In particular, the command **ls -l /ns** might give the following results.

```
f r-------- 103   257 Mon Apr  2 16:19:20 1990 00
f r-------- 103   257 Mon Apr  2 16:19:23 1990 01
f r-------- 103   257 Mon Apr  2 16:19:23 1990 02
f r-------- 103   257 Mon Apr  2 16:19:24 1990 03
f r-------- 104   257 Mon Apr  2 16:19:24 1990 04
```

```
f r--------   0   257 Mon Apr  2 16:19:24 1990 05
f r-------- 103   257 Mon Apr  2 17:34:54 1990 06
f r--------   0   257 Mon Apr  2 16:19:25 1990 07
f r--------   0   259 Mon Apr  2 16:19:19 1990 IO
```

The **/ns** directory contains an entry for every processor in the network. Please note that the object **/Cluster/00/ns/01** is different from the processor **/Cluster/01**. The former merely provides a convenient way of performing certain operations on the latter. This is reflected in the object types shown in the first column, type file, whereas a real processor is actually a directory of servers.

The next column indicates the direct access ordinary users have to these objects, which is very little. In a multi-user environment it is essential that users cannot reset or reboot other users' processors. More subtly, they are not allowed to reset their own processors if this involves disconnecting part of the network, and clearly protection mechanisms like this are useful even in a single-user environment. A user's Task Force Manager may have greater access to certain processors, and that user's applications can use these greater access rights when using networking library calls.

The third column is the account number, indicating who currently owns which processor. This account number corresponds to somebody's user id as extracted from the password file (see the section on the Session Manager below.) In this example all processors with account 103 are currently 'owned' by user Mary, and the processors with account 0 are currently in the system pool of free processors.

The next column usually refers to the size of an object. The 'size' of a processor is a rather dubious concept: 7.2 cm^2 is not very useful information. Hence this field is actually used to store the current state of the processor. This state encodes various bits of information. The bottom byte indicates the **processor purpose**, for example whether it is an I/O processor, a normal Helios processor, a processor reserved for use by the system, or something else. The top three bytes encode the current state of the processor, for example whether or not it is running. Rather than forcing users to interpret this information by decoding the bits Helios provides a command **network** which, given the **show** option, will display the current state of the network.

The fifth column is a date stamp corresponding to the time when that processor was last booted or rebooted. Note that processors 00 and IO were booted within a short time of each other – the boot time of an I/O processor is the time when the I/O server started running. Then there is a short delay before the other processors are booted, corresponding to the time it takes for the network server to start up. Processor 06 has a much later boot time, which indicates that at about that time processor 06 crashed and was rebooted.

The network server is responsible for administering the network. That is a large job, and the network server is a fairly large program. In the interests of modularity there are separate programs, the Session Manager and the Task Force Manager, to administer users' sessions. In a network there will be a single network server responsible for administering the network. There will also be a single Session Manager, responsible for all users. There may be a number of Task Force Managers, one for every user currently logged in.

2.2.7 The Session Manager

In addition to the network server, the **startns** program starts up the Session Manager which is responsible for administering all users' sessions. When the Session Manager installs itself in the name table as **/sm**, the **init** program detects this and the **initrc** command **waitfor /sm** succeeds. **init** now executes the **newuser** command to start a user session, by interacting with the Session Manager. If the **nsrc** configuration file indicates that no password verification is required, the Session Manager does not generate 'login' and 'password' prompts. Creating a new session involves starting up another program, the Task Force Manager, as described later in this section. The Session Manager is a Helios server, like the network server, and hence it can be listed with the **ls** command. Typical output might look like this.

```
d r- z-       0   0 Mon Apr 2 16:19:22 1990 Windows/
f r-        103   1 Mon Apr 2 16:19:28 1990 mary
f r-        104   1 Mon Apr 2 16:30:03 1990 jon
f r-        103   1 Mon Apr 2 17:10:52 1990 mary.1
```

The first entry in the directory is **Windows**. This is a subdirectory, holding details of all the windows currently known to the Session Manager on which it should accept user sessions. The **newuser** command can be used to register a window with the Session Manager.

The next three entries indicate the users currently logged in. User Mary first logged in shortly after the network was booted, as a result of the **newuser** command in the **initrc** file. About ten minutes later user Jon logged in through a different windowing system. For example, if the processor network is connected to an ethernet then user Jon may have logged in through a telnet session. Some time later user Mary logged in again, through a different windowing system. In order to maintain the uniqueness of all names in the network the Session Manager had to append a number to the name Mary for this second session, or there would have been two objects `/sm/mary`. Creating a new session always involves checking the password file, which is called **/helios/etc/passwd**, even if password checking is not enabled, because that file contains other information relevant to the session. A typical line in the password file might be:

```
mary::103:0:mary smith:/helios/users/mary:/helios/bin/shell
```

The number 103 indicates a unique user identifier, which is used in several other places within the networking software. The column `/helios/users/mary` specifies the home directory for that user, and the final column indicates the command to run for that user when the session starts up, in this case the Helios shell.

In a listing of the Session Manager the various fields have the following meanings. All sessions are of type **file**, and users are unable to delete each other's sessions. The account field indicates the user identifier, obtained from the password file. The size field does not make much sense: it is hard for a processor to work out that the user is 1.80 metres tall, and again this information is of little use. The time stamp indicates when the user logged in.

There are various ways of starting a new user session. The conventional way is to log in by using a window. However, this window may be in an I/O processor, or on a dumb terminal, or it may be a telnet session for an ethernet login. Alternatively,

a session may have to be created to support remote execution of commands, using the Unix **rsh** command, for example. All these cases involve creating a new entry in the **/sm** directory, and the code to do this is built into various utilities. In the **initrc** file used for this example, there is a **newuser** command. The Session Manager will start up a separate program, the Task Force Manager, to handle an individual user's requirements.

2.2.8 The Task Force Manager

The **initrc** command

```
run -e /helios/bin/newuser newuser mary
```

registers the current window with the Session Manager, causing a new entry to appear in the subdirectory **/sm/Windows**. In all such windows the Session Manager will run the **login** program to let people start a new session. In this case a user name has been given as the argument to **newuser**, so the first time that **login** is run it will default to that user. Depending on the **nsrc** file there may or may not be a prompt for a password. Once **login** has all the information needed to create a session for a particular user, it causes a Task Force Manager to be started for that user. The current window will be used as the output window for diagnostic and error information, and eventually **login** will start up a shell running in that window. By this point the password file will have been consulted, so the user's home directory and the particular shell to execute will be known.

The Task Force Manager is another Helios server. It will install itself in the name table as `/mary`, or whatever the user name happens to be. The server contains a number of subdirectories: **domain** and **tfm**; again these can be listed with **ls**.

The command `ls -l /mary/domain` might give the following output.

```
f rw- - - - - - -103  257 Mon Apr  2 16:19:24 1990 00
f rw- - - - - d- 103  257 Mon Apr  2 16:19:23 1990 01
f rw- - - - - d- 103  257 Mon Apr  2 16:19:23 1990 02
f rw- - - - - d- 103  257 Mon Apr  2 16:19:24 1990 03
f rw- - - - - d- 103  257 Mon Apr  2 16:19:24 1990 04
```

The **/domain** directory contains the various processors currently owned by the user. The fields have the same meaning as in the **/ns** directory of the network server. The term 'owned' may be inappropriate, since the processors are actually on loan from the system pool and will be returned to that pool when the user has finished with them. A user's domain of processors will grow and shrink as required. If an application needs more resources than are available in the current domain the Task Force Manager will request additional resources from the network server, and return these resources when they are no longer required. In addition there is a **domain** command which may be used to perform operations such as pre-allocating a group of processors. In a single-user environment it often makes sense to pre-allocate all processors, and this could be done with the command:

```
domain get /01 /02 /03 /04 /05 /06 /07
```

It is not necessary to allocate processor 00 in this way, because this processor (probably) runs the user's Task Force Manager and hence is automatically part of that user's domain. The remaining processors start off in the system pool. The domain command can be used with different arguments to release processors back to the system pool or to get further information.

A listing of the **/tfm** directory might give the following information.

```
t rw----da 0   0  Mon Apr 2  16:19:24 1990 shell.1
t rw----da 0   0  Mon Apr 2  16:19:48 1990 shell.6
t rw----da 0   0  Mon Apr 2  18:10:42 1990 ls.82
d rw----da 0  88  Mon Apr 2  18:10:32 1990 pi.78/
```

The **/tfm** directory lists the tasks and task forces which the Task Force Manager is currently running on behalf of the user. There are three single programs, two shells and the **ls** program, and there is one task force or collection of programs. Each entry has an extension; for example, the first shell was the first program run on behalf of that user, and the second shell was the sixth program.

The first shell in the **/tfm** directory is created by the **login** program, when the user's session is created. In fact **login** will execute whatever program is specified in the password file, but this will almost certainly be a shell. The shell is started up with a capability for the Task Force Manager, and after the usual shell startup a prompt will be displayed. The user is now able to type in commands, and can run applications. Essentially this completes the Helios bootstrap process.

When the user logs out, the first shell will terminate. The **login** program will be informed about this, because it started up the shell. It can now terminate, causing the Session Manager to run another **login** program in the same window. Also, the user's Task Force Manager will terminate and release all resources back to the system pool.

2.2.9 Summary of the bootstrap process

The whole bootstrap process can be summarised as follows:

1. The I/O server boots a Nucleus into the root processor, or a ROM bootstrap causes a Nucleus to start up.
2. The Nucleus initialises itself and runs the **init** program.
3. The **init** program reads the **initrc** file and runs **startns**.
4. **startns** runs the network server and the Session Manager.
5. The network server boots up the network.
6. Simultaneously **init** runs the **newuser** command.
7. **newuser** registers the window with the Session Manager.
8. The Session Manager runs **login** inside the window.
9. **login** creates a new session, causing a Task Force Manager to be started, and then runs the login shell inside that Task Force Manager.

10. The user gets a shell prompt and can start executing commands. The Task Force Manager obtains resources from the network server as required, and returns these back to the system when they become free.
11. After a period of time, the user logs out from the login shell.
12. The Task Force Manager terminates, releasing resources back to the free pool.
13. The login program terminates, and is restarted by the Session Manager.
14. Another login prompt appears in the window.

2.3 Some example networks

This section describes a range of Transputer networks, and outlines the software required and its configuration. The networks range from single-processor systems to multi-user networks. Section 2.5 describes how to configure each of the networks. It is hoped that the reader will recognise at least one of these possible networks as the appropriate one for them.

2.3.1 Single-processor embedded systems

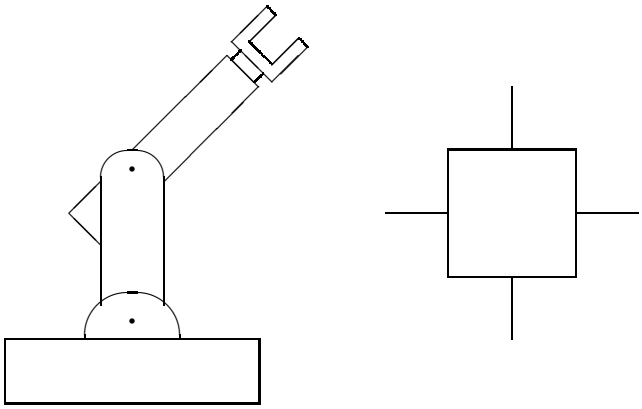


Figure **2.5** Single-processor embedded systems

For single-processor embedded system applications, the processor is used to control a piece of hardware such as a robot arm or a video recorder.

For such an application there is little point in having an operating system at run-time. Operating systems tend to need a hard disc for I/O, and they use a considerable amount of memory. Instead the processor will boot from ROM when it is powered up, and the entire application is held in this ROM. The application will be implemented with a standalone system such as occam or Helios Standalone C. However, there is a question of how this software is developed in the first place, and having an operating system during the development stage may be a distinct advantage. During development the target processor would be part of a normal network using one of the configurations described later in this section, and it would be booted from a link. ROM code would be produced as a final stage.

2.3.2 Single-processor workstation

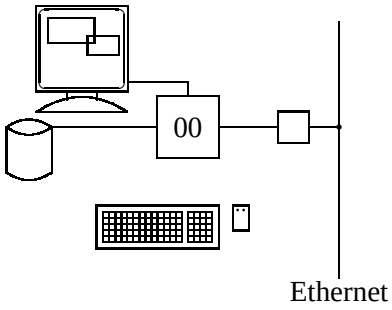


Figure **2.6** A single-processor workstation

A single processor may be equipped with sufficient I/O hardware to produce a complete workstation. The processor would boot from ROM on 'power-up', and load a Nucleus from the hard disc. This Nucleus would contain the filing system. The Nucleus can now start up the init program as before. The **initrc** file would start up software to interact with an ethernet device, giving conventional local area network capability. Keyboard and mouse servers should start up to interact with those devices, and an X window server could follow to give a high resolution graphics display.

Much of the Helios networking software is no longer required, since there is only one processor in the network. There is little point in starting up the Helios network server or to have a per-user Task Force Manager. To protect users from each other Helios insists that every user has one processor. This is because some processors, such as the Transputer, do not have memory management hardware, thus allowing users' programs running on the same processor to corrupt one another. Hence in this single-processor network it will not be possible to log into the machine over the ethernet. It is necessary to start a user session for the workstation's owner, so a Session Manager must be run. It may be configured to require password checking or not. The Session Manager will start up a shell session for the user, but not a Task Force Manager. There should be a separate user id such as `shutdown` which, instead of running a shell, causes the hard disc to be synchronised and so allows the workstation to be shut down safely.

Some or all of the links on this processor will be free. Hence it is possible to connect this workstation into a larger network to produce one of the configurations described later in this section. With this configuration there is potential for catastrophe. If the user gets the configuration files seriously wrong, the machine may not boot up. The same is true if the filing system is badly corrupted, or if the Nucleus is held on the root sector of the hard disc. There are various solutions to this problem. One approach is to support an alternative floppy disc bootstrap mechanism instead of the hard disc bootstrap, typically by pressing a switch or holding down a key when the machine is turned on. This will give a minimal system which should allow the user to perform any necessary repairs. Another more complicated approach is to have an ethernet bootstrap facility. A third approach is to connect a working Helios system to the broken one, boot the processor through a link, and repair the broken system with the working one. All three approaches will require considerable expertise.

2.3.3 Workstation with I/O processor

It is possible to build a system with similar functionality to the workstation by reusing some existing hardware. Typically the user might start with an IBM PC or compatible and plug in a Transputer card, with just one Transputer. Such a network would rely on the PC for all its I/O, both file I/O and screen I/O. An upgrade might involve adding a second Transputer with graphics hardware, and running the X window system on this processor. The PC's screen would be used only for error logging, for debugging, and for generating 'beeps', although the PC's keyboard, mouse and hard disc are still required. The next upgrade is likely to be a hard disc, probably with an SCSI interface, to avoid the bottleneck of the PC's link interface. This would also provide a secure filing system, with multi-user protection, rather than the unprotected filing system of the PC. The final upgrade is likely to be an ethernet connection, giving similar functionality to the standalone workstation described in the previous section.

The gradual approach has advantages and disadvantages. The individual stages are likely to be cheaper than buying a complete workstation at once, but the end result is likely to be more expensive. Every time an addition is made the system configuration will have to be changed, which may be a minor cause of headaches. The workstation will probably come fully configured, apart from the details of user ids, passwords, home directories, and so on. The PC is still usable for conventional software such as spreadsheets and word processing packages, unlike the workstation.

The software is essentially the same as for the workstation. In this case the I/O processor will do the bootstrap rather than a piece of ROM code, and initially the host filing system will be used rather than the Helios filing system, but these do not significantly affect the configuration of the networking software. It should be noted that it is not possible to build a secure system with just the PC filing system. The various system files which must be secure, such as the password file, could be changed simply by leaving Helios and editing them under MS-DOS. To build a secure system, rather desirable for a multi-user environment, the Helios filing system must run within the network.

2.3.4 Workstation for developing parallel software

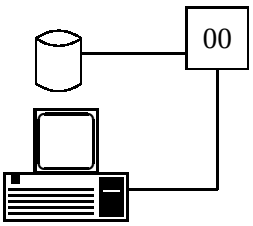


Figure **2.7** A workstation for developing parallel software

In the description of the single-processor workstation it was mentioned that there is no point in running all of the networking software on a single-processor system. This is not always true. Some parallel programming systems, notably the Helios CDL, allow parallel software to be developed on a single processor and then moved to a multi-processor system without change, provided the official guidelines are followed.

Testing parallel software requires all of the networking software to be present since, for example, the Helios shell does not know much about task forces or collections of programs, and how to map these onto a network. Such knowledge is built into the Task Force Manager, and should not be duplicated unless absolutely necessary (in the interests of memory economy as well as for other reasons).

2.3.5 A small network

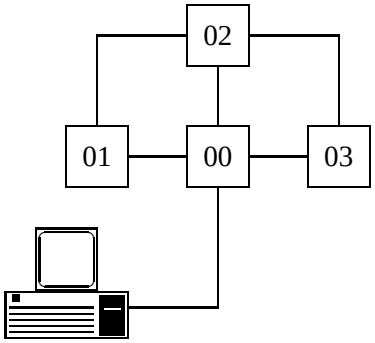


Figure **2.8** A small network

The next level of complexity is to change from a single-processor network to a small network, of perhaps four processors. There are two main ways of organising such a network. The first approach is not to use the networking software because the network is too small to make it worthwhile. This has the advantage of greatly reducing the amount of configuring. Helios provides commands which allow bootstrap of other processors without the need for networking software, and these commands could be executed from the initrc or the shell's **login** files. It should be noted that with certain types of hardware it is fairly difficult to initialise the hardware correctly, and hence this option may not be viable.

The second approach for a small network is to run the networking software, configured as a single-user system. This allows all the networking software to be run on the same processor, saving some memory. The detailed configuration will be similar to the next network.

2.3.6 A fairly small single-user network

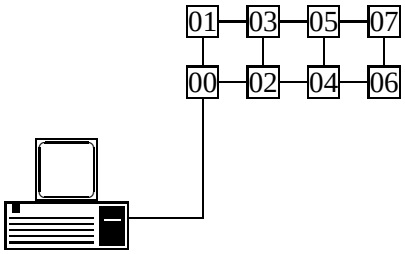


Figure **2.9** A fairly small single-user network

Once the network grows past a certain size, booting by hand is less reliable. Hence the user must run the networking software to boot up the network. As the number of

processors increases it becomes more important that the network is monitored automatically for failures. Also, it becomes less likely that the network will not be used to execute parallel software, and the relative overhead of running the networking software becomes quite small.

2.3.7 A network with configuration hardware

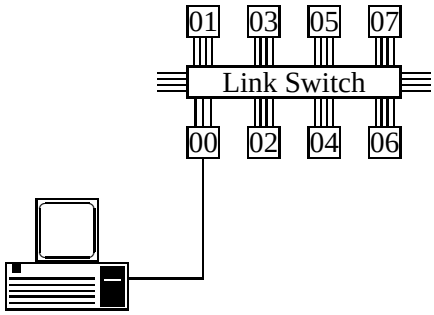


Figure **2.10** A network with configuration hardware

The networks shown so far are assumed to have hard-wired links. Connections between processors may be fixed permanently because that is the design of the board. The connections may involve manipulating pieces of wire. A different type of hardware uses a link switch. Some or all of the links of the various processors enter the link switch, which must be programmed to make appropriate connections. In such a network the resource map which describes the network to the network server takes on a new meaning. Instead of specifying what the network **actually** looks like, it specifies what it **should** look like. The network server initialises the link switch and sets up the desired network. This involves going through a hardware-specific interface. It will be difficult to boot up such a network without a network server, because the configuration hardware is complex. Supporting link configuration adds significantly to the complexity of the networking software. By default, link configuration is only used when the network server starts up. Helios makes no attempt to support automatic dynamic reconfiguration in response to workload or to help map a problem onto a network. Instead the network is assumed to be static. Since the Helios Kernel implements automatic message routing, the need for dynamic reconfiguration is usually extremely small.

2.3.8 A single-user supercomputer

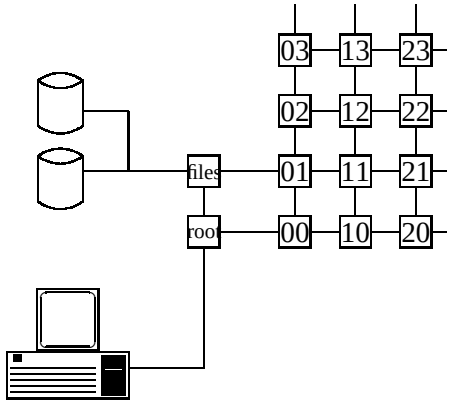


Figure **2.11** A single-user supercomputer

In software terms there is little difference between booting up eight processors in a network and booting up 64, except that the latter may take several seconds more. Allocating 64 processors to just one user is fairly expensive in financial terms, but can make sense for 'compute intensive' jobs. On a 64-processor network, automatic detection of failures is essential because it can take a long time for a user to detect that one processor has stopped working. Hence the network server must run continuously. Such a large network will be used only for running parallel software, so a Task Force Manager is also essential.

2.3.9 Several single-user systems

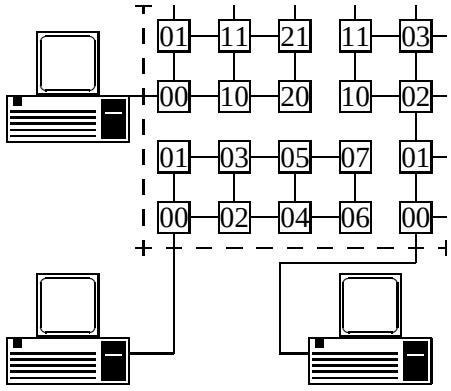


Figure **2.12** Several single-user systems

Given an array of 64 processors it is possible to have perhaps four user networks of 16 processors each, with no overlap at all between the networks. This is a safe way of managing the network, since the users have no way of interfering with each other's networks. However, it may be an inefficient way of using the resource. If a user is currently making use of just two of the 16 processors the remaining 14 are idle and not accessible to any of the other users. To reallocate the resources between the users, so that some get more processors, will involve a considerable amount of work: (1) Terminate some or all current sessions to ensure that the network is idle. (2) Reconfigure

the network with the link switches to the desired allocation. (3) Change the resource maps which define every user's network. (4) Reboot. There is a further complication: changing the resource maps usually involves a Helios session.

Configuring such a system is essentially a case of having four separate sets of single-user configuration files. In addition the controlling software, which is provided by the hardware manufacturer rather than being part of Helios, must be set up correctly. This is specific to the implementation.

2.3.10 A single-user process control system

Processors designed for multiprocessing, such as the Transputer, can be very useful in a process control system, because they combine processing power and communication in one package. Care must be taken with communication, such as using suitably shielded cables and adequate buffering, or using optical connections instead of electrical ones, but such hardware details do not affect the software or the configuration.

When the network is powered up suitable software must be run on all the processors, which can be done by specifying the programs in the resource map. The network server will boot up the network and run all those programs. Furthermore, if a processor crashes the network server will attempt to reboot it and restart the software, without the need for any user intervention. Fault tolerance will be important, so the network will have to be strongly interconnected to allow continued communication even in the presence of crashed processors.

The network must contain either an I/O processor or a processor booting from ROM, to get everything started. There must be a filing system from which the various pieces of software can be loaded, and a display to give monitoring information. High resolution graphics may be inappropriate for some process control applications, so the display may be just a terminal attached to a serial line. The monitoring software may explicitly monitor the other programs in the network. Alternatively it may be implemented as a Helios server, with the various programs acting as clients.

For many process control applications a single-user system is all that is required. In fact the network may even be configured as a zero user network, with all the software started up automatically by the network server and the init program, and with no user sessions. This network may well be merged into a larger network, providing an integrated system within say a whole factory rather than just on part of one factory floor. In the interests of fault tolerance, the system should still be designed as small networks booting up separately and then connecting, to avoid a single central service responsible for everything. Within this larger network, it may be desirable to have a multi-user system.

2.3.11 A small multi-user network

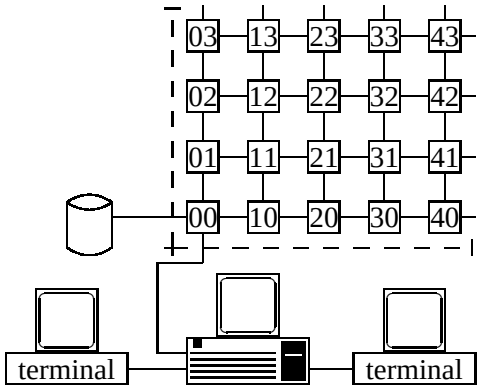


Figure **2.13** A small multi-user network

In the networks described so far there has been at most one active user at any one time, although different users could log in at different times. Adding multiple users involves considerable complications at the software level, but few changes to the configuration files.

Consider the network shown in Figure 2.13. There is a network of perhaps eight or sixteen processors, booted by a PC I/O processor. Attached to this I/O processor are two or more ordinary terminals, using the PC's serial ports. The intention is to boot up the network from the I/O processor and then allow multiple users to log in using the dumb terminals. Alternatively the I/O processor may be equipped with some local area network hardware such as ethernet, and users may wish to log into Helios remotely over such a network. There may not be an I/O processor. The system may consist of a standalone workstation with an ethernet connection and perhaps a serial line to give a system console for error messages. The users have to share the network in a fair manner, which will need a certain amount of cooperation between them.

Sharing a network fairly means that processors should be obtained from the system pool when required, and released back to the system pool when free again. Users should be discouraged from pre-allocating large domains of processors, since those processors would no longer be usable by others. Recovery from errors also becomes more important in a multi-user environment, since users should be inconvenienced as little as possible by the mistakes of others. Hence in a multi-user network it is essential to have a network server running continuously. To allow users to log out and in at any time the Session Manager must also run continuously. Finally, every user will be given a separate Task Force Manager to administrate that session. Since processors have to be allocated from the system pool and released again, these Task Force Managers must also run continuously. A side effect of this is that multi-user configurations will use up more of the available resources, including processor memory, than single-user configurations. The I/O processor has a special ability in such a network. Without an I/O server Helios will be unable to access the serial lines of the host, so the I/O server must not exit while other users are logged in. Furthermore the I/O server must boot up the first processor, and has the ability to reboot this processor at any time. This could be unfortunate if other users are still logged in. The recommended way to avoid these problems is to treat the I/O processor as a system console, which is

used for administration rather than for user sessions. All user sessions go through the terminals. Of course there is nothing in Helios to stop users from ignoring this advice and using both the I/O processor and the terminals for sessions.

2.3.12 Two connected single-user networks

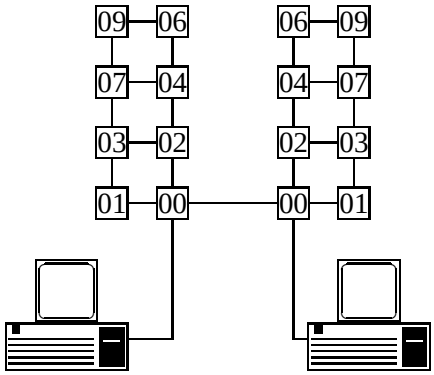


Figure **2.14** Two connected single-user networks

In the case of two separate networks, perhaps plugged into separate I/O processors, the users may want to connect their networks together to exchange data, rather than to share all of the network resources including the processors. The networks are booted up separately, as single-user networks. The link connecting them should be declared as a special external link in the network resource maps, so that the network server will know that there may be a Helios network at the other end of the link. To connect the networks involves running a program to enable the connecting link in one of the networks. If the networks are already connected, this will be a 'no-op'. The networks can be disconnected again by running another program to disable the connecting link. The exact commands differ, depending on whether a network server is currently running, but since the operation involves simple networking it is easier if the network server is running.

There is a potential problem with naming the two networks. If both users give their network the same name, such as `/Cluster`, the naming system becomes ambiguous. There will probably be two processors called `/Cluster/00`, two called `/Cluster/IO`, and so on. The users must give their networks separate names. One network could be called `/maryNet`, the other `/jonNet`. The naming tree would now look something like Figure 2.15.

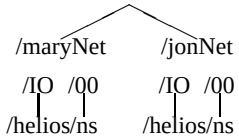


Figure **2.15** Multiple connected networks

User Jon could access a file `/maryNet/IO/c/work/test.c`, with no doubt as to the location of this file. Of course users may not have access to all of the others' resources. The Helios filing system enforces a protection mechanism and, if a user does not have a suitable capability, a particular file may not be accessible.

In this configuration it is not possible to use a processor in another user's network. If one of the networks has been configured to support multiple users the user can log into that network over the link and use processors in the remote network, through a separate session using the same window. The user cannot run an application distributed over processors in more than one network.

2.3.13 A large multi-user network

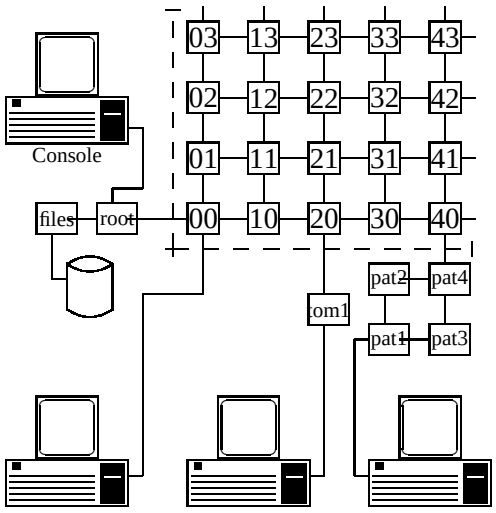


Figure **2.16** A large multi-user network

Given a large backbone of processors it is possible to build a powerful multi-user system. When the network is powered up, the bulk of it is booted, and the network server and Session Manager continue running. The bootstrap can involve either a ROM boot or an I/O processor, with the error logging going to a system console. This console is used only for maintaining the network, not for user sessions.

There are various ways in which users can use this network, illustrated in Figure 2.16. On the left is an I/O processor with just a link adapter, no processor. The I/O server on that processor is configured so that it never attempts to boot up the processor attached to its link adapter. It enables the link, and waits for something to happen. The resource map used to boot up the bulk of the network should indicate an I/O processor at the other end of this link. When the network server detects that the link is enabled, it will locate a window server at the other end of the link and inform the Session Manager that a user is waiting to log in. The Session Manager does the rest, starting up a Task Force Manager for that user and so on. When the user logs out or terminates the I/O server, the session will be terminated.

In the middle is an I/O processor with one processor attached. The I/O server boots this processor with a Nucleus and the **init** program starts running. A network server is started to initialise this small network, but there is no need for a Session Manager. Once the network has been initialised the **joinnet** command can be used to enable the link to the main network and make the small network part of the main one. It is now possible to run the **newuser** command to register the window with the Session Manager. The processor `tom1` in the small network will only be allocated to sessions

started from that network, and will not be accessible to other users.

The third approach has a small network rather than a single processor. The configuration is the same as for the second: a network server is run to boot up the small network; then the **joinnet** command is used to make this small network part of the main machine.

A standalone workstation can be used instead of an I/O processor for the second and third case, using exactly the same configuration, although a standalone workstation consisting of a link adapter but no processor or I/O processor does not make sense.

2.3.14 A mainframe computer

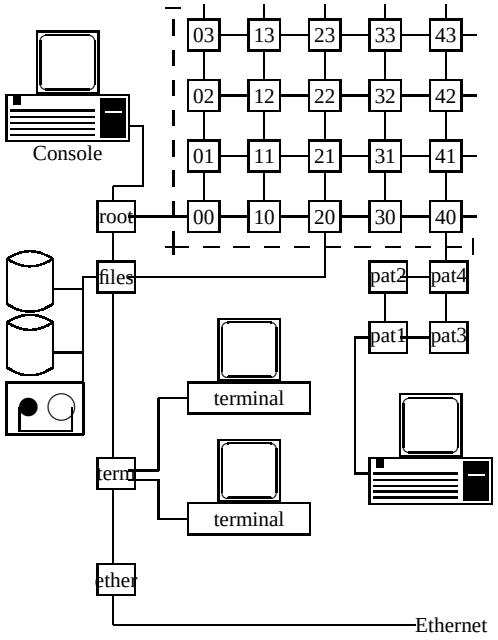


Figure **2.17** A mainframe computer

The large multi-user network described in the previous section, with a central reliable backbone of processors and a central network server and Session Manager can be extended to give a large and powerful system. There is no technical reason why it could not be expanded to several hundred processors, with a number of fast discs, tape streamers for backup purposes, and one or more ethernet connections. In addition to logging in through I/O processors or standalone workstations the network could support serial line terminals, ethernet logins (possibly from remote X window system terminals), or even dial-up logins, if a serial port and a modem are added. Such a network provides much the same facilities as a traditional mainframe.

2.3.15 Networked mainframe computers

There is one main problem with the mainframe approach to building networks. They rely on a single network server to allocate resources between users, and the network

server is a single program running on just one processor. The workload of the network server depends on two things:

1. The number of users, which will affect the rate at which requests come in, and

2. The size of the network, which will affect the amount of work to be done for each request. When a certain number of users and a certain number of processors is exceeded, the network server will become a bottleneck. At present there is insufficient data to determine when this will happen.

2.4 The real world

The previous section described various networks which, at least in theory, can be built quite easily with Transputers. In practice most Helios users purchase off the shelf hardware, and this will have built-in limitations which may make it unsuitable for certain applications.

This section describes four different systems, representing different suppliers of Transputer hardware: the Inmos TRAM system, the Parsytec MultiCluster and SuperCluster[5] systems, the Telmat T.Node[6] and the Meiko Computing Surface. These systems differ significantly in the hardware used to reset processors, the configuration hardware, and so on. The purchase price of the hardware also varies considerably, but that topic is not considered further here.

Given a network of processors, a common question is how to interconnect them conveniently in a way that is appropriate for the application or applications desired. A correct interconnection may be more important for a small network, where communication costs may have to be minimised if the application is to run efficiently. For a larger network, particularly a multi-user network, attempting to optimise the interconnections is less important and requires more effort. This section outlines the topic of network interconnections.

2.4.1 Different hardware

This section describes four different hardware systems, with different strengths and weaknesses. These have a significant effect on the networking software, and on the suitability of the hardware for different applications.

2.4.2 Inmos

Inmos, as the manufacturer of the Transputer, have a strong influence on the industry as a whole. In 1987 Inmos introduced the **TRAM** system, an ‘industry standard’ for building hardware based on Transputers. The TRAM system has since been adopted by a number of other manufacturers. However, by 1987 several other manufacturers had already implemented their own Transputer systems which are not compatible with the TRAM scheme, and which have taken a major share of the market place. In addition,

[5] Parsytec, MultiCluster and SuperCluster are trademarks of Parsytec GmbH.
[6] Telmat and T.Node are trademarks of Telmat Informatique

the TRAM scheme has a number of weaknesses which makes it inappropriate for many applications.

The idea behind the TRAM scheme is quite simple. Manufacturers produce TRAM **modules**, small or medium sized circuit boards typically with one Transputer and some memory. These modules can be plugged into a TRAM **motherboard**, to build a network of processors. A typical motherboard might have between five and sixteen **slots**, and a module can use up anything from one to eight slots. In addition to processing modules with just a Transputer and memory, it is possible to have specialised modules with such features as graphics displays, SCSI peripheral interfaces, and ethernet connections. The various modules are connected together automatically in a simple pipeline. Consider the network of Figure 2.18.

The motherboard has ten slots, like the Inmos B008 board. It is filled with six modules. There is a size four module occupying slots 0, 4, 7 and 3, say the Inmos B417 module with a T800 and four megabytes of memory. Then there are four size one modules, in slots 1, 2, 5 and 6, say four Inmos B411 modules each with a T800 and one megabyte. Finally there is a size two SCSI module such as the Inmos B422, with a T222 and a SCSI interface. Jumpers will be required on the first module, to avoid breaking the link pipeline.

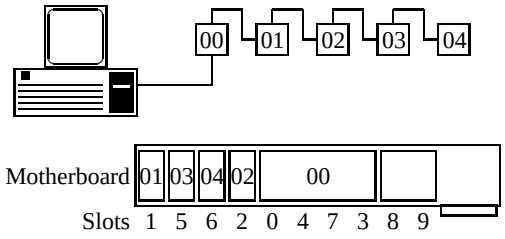


Figure **2.18** A single-TRAM motherboard

Link 0 of the first module goes to the I/O processor. Link 2 of the first module goes to link 1 of the second module, link 2 of the second module is connected to link 1 of the third module, and so on. In addition some motherboards have a link switch to allow the remaining links of each module to be connected according to the user's requirements.

It is possible to chain together several TRAM motherboards, to produce a larger network. To achieve this the tail of the pipeline, link 2 of the last module, goes through an external connector into the head of the next pipeline (link 1 of the first module on the second board). Usually this will require some soldering on the **patch area** of the second board. Inmos produce the B211 Transputer Evaluation Module, a rack which can take up to ten TRAM motherboards such as the B012, each of which has sixteen slots for TRAM modules. Clearly it is possible to build very large networks with TRAM modules. It should be remembered that the recommended minimum amount of memory on a processor running Helios is one megabyte, even though there are TRAM modules with much less memory.

The flexibility of the TRAM scheme and the availability of a wide range of different modules are its main advantages. However it suffers from two great disadvantages, both related to the reset scheme. The root processor, occupying slot 0 of the first board, can be reset from the I/O processor. The remaining processors can be reset in one of two ways, usually depending on the state of a jumper. The first way is to reset all

processors at the same time from the I/O processor, so that when Helios starts up all processors are in a reset state. The second way is to give the root processor **subsystem control**. This means that the root processor can perform a global reset, resetting all other processors in the network. For Helios, neither approach is particularly useful. If the network is to have any reasonable degree of fault tolerance, the networking software must be able to recover from crashed processors by isolating them and then rebooting them. Isolating is still possible, if the network connectivity has not been broken. However, rebooting is impossible without the ability to reset individual processors. The networking software cannot recover until all processors in the network, apart from the root processor, have crashed and should be reset anyway. Before this happens the user is likely to have lost patience and rebooted the whole network from the I/O processor.

The second problem with the reset scheme is the way in which it is asserted. This is done by poking a 32-bit integer 0 into address 0x00000000. Unfortunately, consider the following piece of poor quality but fairly typical C code.

```
char *pointer = malloc(128);
memset(pointer, ' ', 128);
```

Most of the time this piece of code will work fine. However, if the processor happens to be short of memory the call to **malloc()** will return **NULL**, which is the same as address 0x00000000. Hence the call to **memset** will activate the subsystem reset, if it is running on the root processor, and reset every other processor in the network. This will be rather confusing for the average user, at least until it has happened half a dozen times and the symptoms can be recognised instantly. Furthermore the circumstances which caused the problem (running out of memory when the program executes on the root processor) may not occur very often. These problems could have been avoided very easily, although at a slight additional hardware cost, by choosing a different address such as 0x70000000.

The lack of an individual processor reset facility and the ease with which the reset can be asserted accidentally make the TRAM system an unlikely choice for large networks. However a small number of TRAM modules can be combined to produce a workstation, with or without an I/O processor. A typical collection would be a SCSI module, an ethernet module, a graphics module, and possibly one processor module for the root processor. For a standalone workstation it would be necessary to add a ROM bootstrap module and probably an RS232 module with two serial ports, one for a mouse and one for a keyboard. Such a workstation could be connected into a larger network such as a Parsytec SuperCluster or a Telmat T.Node, to achieve the required processing capability. The various processors within the workstation would run mainly or only system software, such as the filing system or the X window system server. This reduces or eliminates the possibility of a crash on one of these processors, which would require the rebooting of the whole workstation (but not the larger network).

2.4.3 Parsytec

Parsytec GmbH have been working on Transputer systems since 1985. They supply two main systems: the MultiCluster, aimed mainly at industrial applications; and the SuperCluster, aimed more at the supercomputer market. In fact the two systems are

hardware compatible, and it is possible to take MultiCluster boards and plug them into part of a SuperCluster.

The MultiCluster series involves one or more heavy duty racks linked together, and a range of plug in boards. These include processing boards such as the MTM-2, with two Transputers each with one megabyte; I/O boards such as the GDS graphics display and the MSC mass storage board with its SCSI interface; and interface boards such as the BBK-V2 VME bus bridgehead. Host interface boards are available for a range of machines, including PCs and Suns. RS422 link buffering is supported as standard, for medium to long distance communication between processors. In the context of an industrial application, the interconnections between the processors are usually hard-wired using cables plugged in to the MultiCluster backplane. Once the network is up and running it should stay up and running for a long time without changing the software or the configuration. Cables tend to be more reliable than a crossbar switch, reducing the possibility of an error. A typical SuperCluster system is shown in Figure 2.19.

The basic unit of a SuperCluster is known as a **computing cluster**. This contains 16 processors, usually T800s because the system is aimed at supercomputing use which needs floating point arithmetic. Each cluster also contains a **network configuration unit** which has the link switches needed to configure its part of the network. The 16 processors in a cluster have a total of 64 links, which can be connected in any way. In addition, 32 of these links can be taken outside the cluster, to a higher-level configuration unit which allows the clusters to be interconnected. The smallest commercially available SuperCluster has four of these computing clusters, giving a total of 64 processors. Several of these can be combined to produce a larger network.

In addition to the computing clusters, a SuperCluster machine contains a **system services cluster**. Any of the MultiCluster boards, including the MSC with its SCSI disc interface, can be

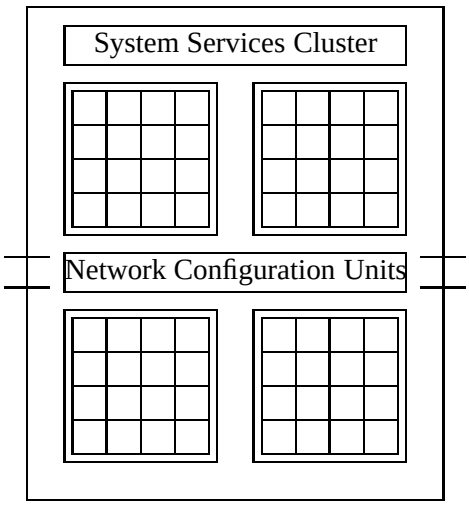


Figure **2.19** A Parsytec SuperCluster

plugged into this cluster to provide the required I/O facilities. External workstations and host processors can be connected here as well.

An important aspect of the Parsytec architecture is the reset scheme. Unlike the Inmos TRAM scheme, every processor can be reset individually. However, unlike the Telmat T.Node and the Meiko Computing Surface this is achieved without having a central control bus. Instead every processor is able to reset any of its four neighbours. This reset facility is supported by all the processors in the SuperCluster and by all MultiCluster boards, giving a consistent way of booting up any network built from Parsytec hardware. This distributed reset scheme is, in theory, ideal for fault tolerant networks since error recovery can start from anywhere. A critical requirement for fault tolerant networks is the duplication of all critical software components, and the network server is one of these. With Parsytec hardware it should be possible to run several network servers in different parts of the network: even if a processor running one network server is crashed, for any reason, the other network servers can recover from this. At the time of writing, this facility is not yet supported.

The Parsytec distributed reset scheme does, however, have disadvantages. It is not particularly secure. Any reasonably competent first-year student or similar hacker can produce a worm program that resets the processors on all four links and duplicates itself down all four links. Such a worm could spread through any network within seconds, wiping out all the networking software before the latter knows what is happening and can attempt to recover. For some applications this makes the machine more suited to the ‘Several single-user systems’ network described in section 2.2, where user networks are physically isolated from each other and hence cannot interfere with each other.

The Parsytec hardware supports a wide range of I/O facilities and networks of an arbitrary size, with a consistent reset scheme throughout the network. This makes it satisfactory for most applications. In theory it is ideal for fault tolerant applications although at present not much software makes use of this. The disadvantages are a lack of security, which may be important for some multi-user networks, and reset problems when mixing Helios and native nodes.

2.4.4 Telmat

The Supernode architecture provides a building block for producing large processor networks. It was developed under project P10850 funded by the Commission of the European Community Esprit program. The Telmat T.Node is a realisation of this architecture, manufactured by Telmat Informatique.

Every T.Node building block contains a reconfigurable link switch mounted directly on the backplane. There are seven plug in card slots per block. Two of these slots are used for worker modules, each with eight T800 processors and memory. A third slot is used for a controller card, responsible for configuring the building block and resetting processors. Two additional slots can be used for connections to other T.Node building blocks, or may be used to house additional processor cards. The remaining two slots are for special cards. Possibilities include a memory card, with one processor and up to sixteen megabytes of memory, and a disc card with a SCSI interface. There is a Control Bus for resetting processors and various other functions. A typical network of these building blocks is shown in Figure 2.20.

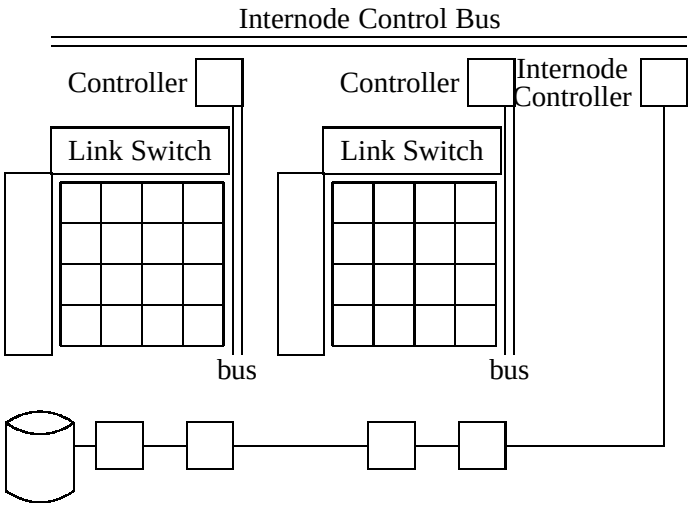


Figure **2.20** T.Node building blocks

Two T.Node building blocks can be combined to produce a Tandem.Node whose workers are fully interconnected through the link switches. Tandem.Nodes can have up to 64 worker Transputers. Larger machines based on the T.Node building block are known as Mega.Nodes. These can have up to 1024 processors in a fully reconfigurable network. This involves a control hierarchy, in particular an Internode Control bus in addition to the Control bus inside every T.Node building block.

Unlike other hardware, the T.Node link switch is not a full crossbar switch. It is possible to realise any desired network connectivity, but not every topology. For example, the link switch always allows processors `05` and `10` to be connected, but might not allow link 0 of `05` to be connected to link 1 of `10`. This does not affect the inner workings of Helios, which depend only on the connectivity. It has a side effect on the network specification contained in the resource map, **/helios/etc/default.map** which usually defines the topology. When running on a T.Node the internal representation of the resource map is modified to give the same connectivity but an achievable topology, before any attempt is made to boot the network.

In addition to the link switch, every processor in a building block is connected to a **control bus**. This bus is used by the controller card for resetting processors within the block, and gives an individual reset facility over all processors. The bus also provides a fairly low-bandwidth communication path between the processors and the controller card, and several other facilities. In a network of building blocks the controller cards are connected through an **internode control bus**, with a supervisor **internode controller**. Essentially this internode controller has complete reset and configuration control over the entire network, and the Helios network server interacts with it to give the required functionality.

Like all hardware, the T.Node has advantages and disadvantages. It is very suitable for building medium-sized and large networks of processors, forming the network backbone. Individual reset of all processors is available, and furthermore access to the internode controller can be restricted so that ordinary users do not have any way of activating the resets directly. This makes the network more secure in a multi-user

environment. The resets go through a bus, so there are no problems resetting native nodes. There is a negative side. Having centralised control makes the network less suitable for fault tolerant applications, because there is no sensible way of having multiple network servers and recovering if one of the network servers is crashed.

If the software on the internode controller goes wrong the whole network will have to be rebooted. Also, the size of a typical T.Node building block, with at least 18 processors, makes it less suitable for producing workstations. Instead a typical network would have other workstations or I/O processors connected to the T.Node building blocks.

2.4.5 Meiko

The final of the four major systems is the Meiko Computing Surface. Meiko's Computing surface has been developed since 1985. It is a scalable multiprocessor architecture with a performance range from workstation to supercomputer.

The Meiko Surface differs from the Parsytec SuperCluster and the Telmat T.Node in that there is no basic building block. Instead a Surface is built up from one or more **modules**, essentially racks capable of holding different numbers of boards which can be interconnected to produce larger networks. There is a wide choice of boards including computing elements, display cards, mass storage cards with a SCSI interface, and so on. Various different host interface boards are available, the most important being for the Sun3 and Sun4. All the Transputer links go into the module backplane, and may be hardwired or connected through a link switch. A heuristic algorithm is used to attempt to achieve the required topology, and this should succeed for nearly every topology.

Like the T.Node the Meiko Computing Surface is based around a control bus, the **Supervisor Bus**. This supports individual reset of processors and low bandwidth communication. Every module should contain a **Local Host** board, providing control over all the processors in that module. The various local hosts in a network should be chained together in another bus, controlled by the network **Module Master**. The Helios network server interacts with this Module Master to achieve the required functionality.

Unlike the Parsytec and Telmat machines, the Meiko Surface comes with its own system software **CS Tools**. Helios is an optional extra, which runs alongside CS Tools. The normal way of using CS Tools is to develop the software on the host machine, usually a Sun, including cross-compiling on the host. When it is time to run the application a **domain** of processors is obtained, and the software is booted into these processors. The system is integrated into the host environment. For example, when the application in the Surface opens a file, a request is sent to the host to perform this operation. The integration extends even to debugging facilities. For example, it is possible to compile the Surface application with debugging enabled and then use the **dbx** program on the Sun. This approach differs significantly from Helios, where the network of processors is continuously running the operating system, and applications execute under the operating system. In particular, all software development including compiling and debugging happens under Helios.

CS Tools does impose a number of restrictions. In particular, a domain of processors obtained from the system cannot overlap with other domains so it is not possible

to build a multi-user Helios network. Instead a Computing Surface network behaves like the 'Several single-user systems' network described in section 2.3.9, with the additional possibility that some of the single-user systems may not be running Helios.

The Meiko Computing Surface has a centralised individual reset like the T.Node, and hence it has the same advantages and disadvantages when it comes to security and fault tolerance. In theory the minimum size of a network is two processors, one local host and one computing element, so the Surface could be used to build a workstation with a small network. In practice the minimum machine involves four processors. Arbitrary topologies are available, unlike the T.Node. The other criterion, which may or may not be important depending on the user's requirements, is that the network is not controlled entirely by Helios.

2.4.6 Handling different hardware

This section has described the four most important commercial architectures. Helios supports all four architectures, with an additional but limited capability for mixing different hardware, using one set of networking software. Other hardware can be supported as well, usually without modifying the networking software.

Any network is either homogeneous (which means that all the hardware has the same control facilities) or consists of a number of subnetworks with different control facilities. The network server achieves control over a homogeneous network or subnetwork by loading a **device driver**. A Helios device driver is a piece of code loaded dynamically, usually to provide an interface between the hardware and the hardware-independent software. For example, the X window system server loads a device driver to manipulate video memory and colour look-up tables. The network server loads a device driver or several device drivers to manipulate the network control hardware. At present there are two different types of network device drivers, one to control the reset hardware and one to control the link configuration hardware. The following device drivers are available:

- **null_ra.d** a reset/analyse driver for when no reset hardware is available.
- **tram_ra.d** the reset/analyse driver for the TRAM reset scheme.
- **pa_ra.d** the reset driver for the Parsytec scheme.
- **telmat_r.d** the Telmat T.Node reset driver.
- **telmat_c.d** the Telmat T.Node link configuration driver.
- **rte_ra.d** the Meiko computing surface reset driver.
- **rte_c.d** the Meiko computing surface link configuration driver.

The network resource map **/helios/etc/default.map** defines the device driver or drivers to use for the current network. Please note that some of these drivers are actually owned by the appropriate hardware manufacturers and are not shipped as standard with Helios.

For Helios to run on networks which are not based on one of the four architectures described above, it is usually necessary to produce a new device driver or drivers. In

fact, for a small network it is possible not to specify a device driver and merely ensure that the whole network is reset before starting the networking software. Obviously without a device driver the networking software's ability to recover from errors will be limited.

For a homogeneous network a single device driver will usually suffice. For mixed networks device drivers can be used within the homogeneous networks, but there is a problem at the boundaries. The resource map syntax allows the user to specify reset facilities over and above what is provided by the device driver. For example, consider the network in Figure 2.21.

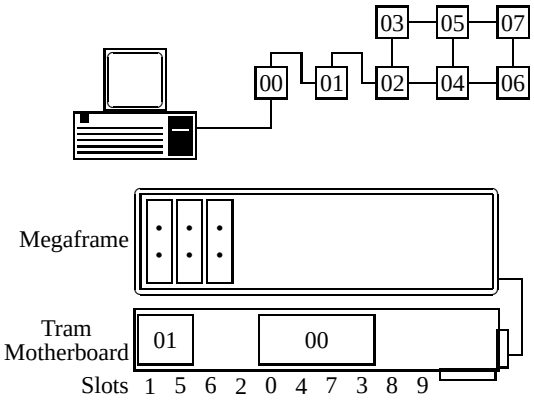


Figure **2.21** A mixed network

Processors `00` and `01` are TRAM modules on a suitable motherboard, and the remaining processors are part of a Parsytec MultiCluster. The TRAM reset is passed on to the first processor in the megaframe, so that whenever the global reset is asserted on processor `00` this affects processor `01`, the other TRAM module, and processor `02`, the first MultiCluster processor. The Parsytec reset scheme can be used on the remaining processors. Clearly such a network is 'bootable', but describing it in a resource map is difficult. Section 2.5 describes how it can be done for many networks.

2.4.7 Mapping task forces onto a network

In a task force each component task can execute on a separate processor, with the communication going over the processor links. A typical application might be a farm, with a master program (M), a number of worker programs (W*n*), and a load balancer (lb) to distribute the workload. Such a farm is shown in Figure 2.22.

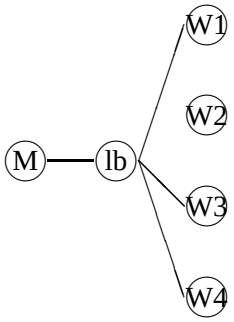


Figure **2.22** A farm application

The ideal way to run such a task force is to assign a separate processor to every component task, with the links between the processors matching the communication between the components. For the farm, this would require a network topology as shown in Figure 2.23.

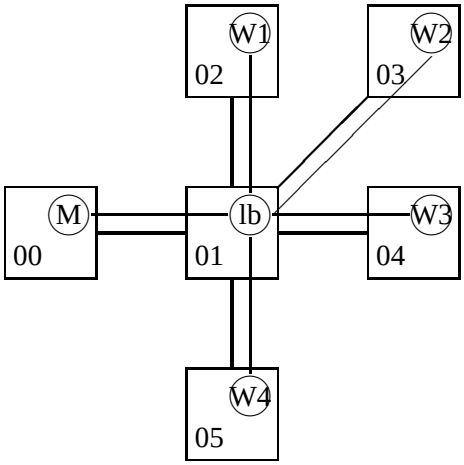


Figure **2.23** The ideal network topology for a farm

However, there is a small problem with this network. Processor `01` is used for the load balancer, which communicates with five other programs, and hence in an ideal network the processor would need five links. This is difficult with the current generation of Transputers. Instead the application must be mapped onto a real network, in such a way as to minimise the communication overheads. This means reducing the number of processors that messages have to be routed through, since this affects both the communication bandwidth and the CPU time available on the intermediate processors. Consider the network in Figure 2.24.

The right hand mapping is significantly better than the left hand mapping, because the average ‘distance’ between the load balancer and the worker components is reduced from two links to 1.5.

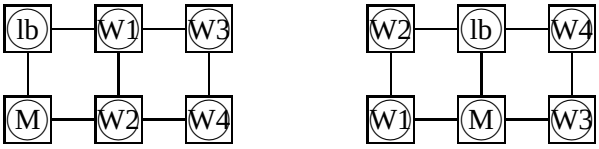


Figure **2.24** Two different mappings

2.4.8 Possible topologies

The networks that can be achieved depend mainly on the number of available links per processor. With just two links the choice is very limited: either a pipeline or a ring, as shown in Figure 2.25. With three links the choice widens. (See Figure 2.26.)

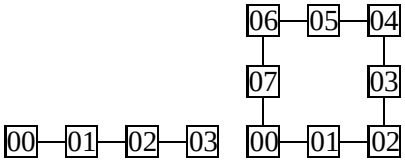


Figure **2.25** Two connections per processor

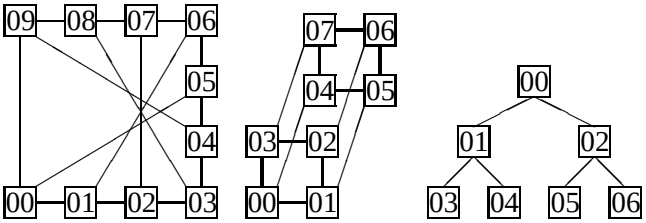


Figure **2.26** Three connections per processor

Both the chordal ring and the tree can be expanded to arbitrarily sized networks. In the cube all the available links are used, so it is impossible even to add an I/O processor. The chordal ring and the cube offer a degree of fault tolerance, in that the failure of any one node will not break the connectivity of the network. In the tree topology, any failure except in the bottom or leaf nodes will disconnect part of the network.

With four links, the number of possible network topologies becomes very large. Common networks include a simple mesh or a mesh with wrap around and a four dimensional hypercube, as shown in Figure 2.27.

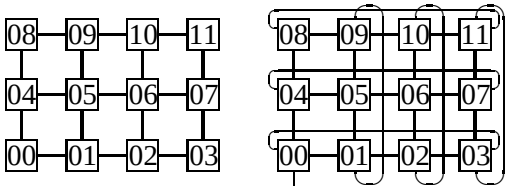


Figure **2.27** Four connections per processor

There are many other topologies such as cube connected cycles, lenses, and hypertrees, all with strengths and weaknesses. Given six links, it would be possible to build three-dimensional meshes, six-dimensional hypercubes with 64 processors, and systolic arrays. The choice of network topologies is very large, but does not help with the basic question: which topology is suitable for the application ?

2.4.9 Task force connectivity

Many applications have a fairly low connectivity. For example, a pipe line of tasks has just two communication channels between every component task. Such a task force can be mapped efficiently onto many different network topologies, as shown in Figure 2.28.

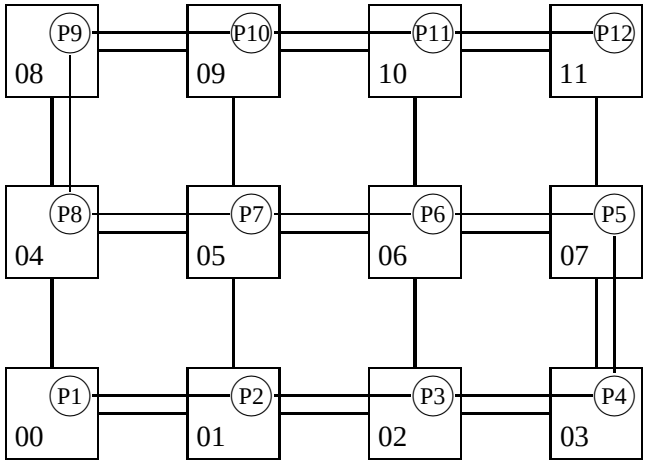


Figure **2.28** Mapping a pipeline of tasks

Other applications have a high connectivity. A farm with 64 workers requires a load balancer communicating with 65 other tasks. For such applications the exact network topology may not matter much, assuming all the processors' links are used, and it is far more important to place the load balancer and master components in a sensible place in the network. A mesh, preferably with partial or full wrap around, will give satisfactory results for most applications.

In between these extremes is the case where the connectivity of the task force is close to the connectivity of the network. For example, in an image processing application, the task force often takes the form of a two dimensional array of component tasks and a mesh with the same dimensions as the task force is usually the optimal network topology. For such applications the question arises as to whether or not the communication overheads are a significant factor in the efficiency of the task force. If every task spends 99 per cent of its time calculating rather than communicating, it will be more cost effective to attempt to optimise the calculation rather than modify the network topology to reduce the communication costs.

2.4.10 Other considerations

In addition to the nature of the task force, there are other practical considerations before deciding on a network topology. The actual hardware must not be forgotten. If the links are hard wired by tracks on a printed circuit board, nothing can change the topology. If the links are hard wired by sections of cable, wiring up a complex topology such as a hypercube cannot be recommended. If the hardware contains link switches, which allow the network to be configured to any desired topology, matters become more manageable but are still complicated.

With a large network, producing a resource map can be difficult. This is greatly simplified if the network has a very regular structure. For regular structures, and particularly for meshes, resource maps can be generated automatically or semi-automatically. The expected gains in performance when using a less standard topology may have to be considerable to warrant the effort of producing the required resource maps.

In a single-user network it is fairly easy to boot up the network with the required topology, by substituting a different resource map. This is not true in a multi-user network, where the underlying topology is decided by the network administrator and users merely borrow processors from the system pool.

2.4.11 Summary

In an ideal world networks would always match the topology of the application. In an almost ideal world, networks could be made to match the topology of the application. There is a wide range of possible topologies, offering hours of fun, but in the real world a user has to consider several questions before changing the network topology to match the application.

- Does the task force topology map ‘satisfactorily’ on to the existing network? The user has to define ‘satisfactorily’ in this context.
- Is communication a bottleneck for the application?
- Can the communication costs be reduced significantly by changing the network?
- Can the existing hardware cope with the desired connectivity?
- Must the application run unchanged on a different network, with different and less flexible hardware?
- In a single-user system, will the application be run often enough or for long enough to make the production of a new resource map worthwhile?

Experience has shown that a simple mesh, preferably with partial or complete wrap around, is satisfactory for nearly all cases and is easy to use.

2.5 Network commands

The bulk of the Helios networking software consists of three programs: the network server to administer the network, the Session Manager to handle all users and the Task

Force Manager, which handles a single user's session. None of these commands are run directly by the user. However there are various commands which interact with these programs and which allow users to exert greater control over the network. This section describes these commands.

The commands divide into a number of categories.

- **startns** must be used to start networking software.
- **findns**, **findsm** and **findtfm** may be used to locate a particular server in a network.
- **rboot**, **pa_rboot**, **clink**, **pa_reset**, and **tr_reset** are used when booting networks by hand.
- **dlink**, **elink**, **plink** and **lstatus** can be used to examine and change link modes. These commands are used mainly to connect and disconnect networks.
- **joinnet** is used to connect into a backbone of processors.
- **domain** is used to administer a user's domain of processors.
- **newuser** starts up a new user session.
- **rmgen** is used to compile network resource maps.
- **stopio** and **rebootio** are used to interact with I/O servers.
- **write** and **wall** allow communication between users.
- **who** and **users** list the users currently logged in.
- **whoami** displays the current user name.
- **diag_tfm** and **diag_ns** control debugging options inside a Task Force Manager and the Network Server.
- **uptime** shows how long the network has been running.
- **ps** gives information about which programs are running on processors in the network.
- **loaded** gives information about what code is loaded into processors in the network.
- **network** can be used to examine the current state of the network.
- **login** is used to start a new session.

Full information on all these commands can be found in *The Helios Encyclopaedia* or by using the online help facility.

2.6 Configuration files

The number of Helios configuration files which affect networking is considerable. They tend to be rather confusing to a user, particularly to a user new to Helios, not least because of the differences in syntax. This section attempts to eliminate some of the confusion, by giving details of the following configuration files:

- The **host.con** configuration file is read by the I/O server when it starts up. The file contains a list of options for the I/O server.
- The **initrc** file is read by the **init** program on the root processor, during the initial bootstrap. The file contains a list of commands using a format specific to **init**.
- The **nsrc** file is read by the networking software. It contains a list of options, not commands.
- The **.login** file is read by a user's login shell, the first shell run on behalf of that user, which is usually started by the Helios Session Manager. The file contains shell commands.
- The **.cshrc** file is read by all shells started by a user, whether directly or indirectly. The file contains shell commands.
- The **.logout** file is read by a user's login shell when it terminates, which happens when the user logs out. It contains shell commands.
- Resource maps are used to describe a network of processors. They are written in a language specially designed for this purpose.

In addition to these configuration files there are a number of programs and device drivers. Some of the programs, the network server, the Session Manager and the Task Force Manager, are run by other programs or in response to events. Others such as **startns** and **newuser**, are commands executed by the user, possibly interactively through a shell, possibly as part of a file of commands such as **initrc** or the various shell resource files. The device drivers such as **tram_ra.d** and **pa_ra.d**, complement the networking software by separating hardware-specific code for controlling the reset and link configuration hardware from the hardware-independent networking code. These hardware-specific device drivers may require additional information from other configuration files.

2.6.1 host.con

The **host.con** file is read by the Helios I/O server when it starts up, and contains a list of options for the I/O server. Typical entries might look like this:

```
helios_directory = c:\helios
Server_windows
# This is a comment
link_base = 0x100
```

Lines beginning with a # are comments, and are ignored. Other lines can contain a flag, for example the **Server_windows** line is a flag enabling windowing in the I/O server; alternatively they may specify a string or a number, for a particular option; for example, the first line specifies the string **c:\helios** for the option **helios_directory**.

Four options in the **host.con** file are important when configuring the network :

1. **root_processor**
2. **io_processor**
3. **bootlink**
4. **no_bootstrap**

The first two control the initial processor names. Under Helios processors have names just like other objects, for example `/00` and `/IO`. Processors must be given these names when they are booted (as soon as they 'exist' in the Helios world). Since the I/O server is responsible for 'creating' the I/O processor within the Helios world and for booting up the root processor, it must supply these names. The default names are `/00` and `/IO`, but alternatives can be provided in the **host.con** file.

```
root_processor = /maryRoot
io_processor = /maryPC
```

When booting a processor, that processor must also be supplied with an initial link configuration, specifying which links are not connected, which are connected to active Helios nodes, and so on. Usually the network server will provide this information. However, the root processor must be booted up with a link configuration by the I/O server. The assumption is made that all but one of the links will be not connected, the exception being the link to the I/O processor which must be active. On nearly all Transputer hardware this link is link 0, and hence the I/O server will default to this. For the few exceptions, the **bootlink** option can be used to specify an alternative, for example:

```
bootlink = 2
```

When the I/O server is used to initiate the bootstrap of a network, it is important that the resource map describing the network matches the options used by the I/O server. If the resource map indicates that the root processor is called `/00`, and the I/O server has called it `/maryRoot`, the network will not boot up correctly or at all.

The final option, **enable_link**, is used if the I/O processor is not attached to its own private Transputer. Instead it is equipped merely with a link adapter, and this link adapter is used to connect into an existing network. The **enable_link** option prevents the I/O server from booting or rebooting a Transputer, and forces it instead to enable the link connecting it to the network.

A complete description of the **host.con** options can be found in chapter 8, *The I/O server*.

2.6.2 initrc

All installations of Helios involve booting a Nucleus into one processor, as the first stage. What should happen next is not so clear. A large network used to control a factory floor has different needs from a scientific supercomputer, which is also different from a single-user single-processor workstation. One way to get around this is to use a different Nucleus for different applications, and sometimes this has to be done. For example, in a single-processor workstation the Nucleus must incorporate the Helios filing system, but in a network with an I/O processor and no additional hard disc this would waste memory. The alternative and more flexible approach is to read a textual resource file. On the root processor, and only on the root processor, the Nucleus will start up a separate program **/helios/lib/init**, which reads and executes commands from the text file **/helios/etc/initrc**.

In theory having a separate **init** program with its own parser, its own command syntax, and so on, is unnecessary. A shell could have been used instead. There are a number of reasons why this approach was not taken in Helios.

- Using a shell can be overkill. A shell provides a great many facilities not needed during the bootstrap stage, such as interactive command line editing.
- All of the shell would have to be loaded into memory, including the bits that are not needed. Also, the shell requires the C and Posix libraries, so these would have to be loaded as well. This would be rather inefficient and could cause memory fragmentation problems. The current **init** program is less than 3K in size, and does not need these libraries.
- A shell requires a fairly stable environment, in terms of a console window and reliable file I/O. This may not be available during the Helios bootstrap stage, for example a console window might not exist until the X window system has been started up.
- The requirements of a bootstrap stage are different from those of a shell. In particular it is rather important to have support for detecting the presence and absence of servers or other objects, and for waiting for such an object to appear.

A typical **initrc** file might look like this.

```
#
# This is a comment line
#
# First, set up the windowing system
ifabsent /window run -e /helios/lib/window window
console /window console
#
# Then start the networking software
run -e /helios/bin/startns startns -r /helios/etc/default.map
#
# Wait for the Session Manager to be active
waitfor /sm
#
# And start a user session
run -e /helios/bin/newuser newuser mary
```

As with the **host.con** file, lines beginning with a # are interpreted as comments. Otherwise the file contains a list of commands, with one command per line. The **init** program understands the following commands:

- **run** to execute another program.
- **ifabsent** to check for the absence of an object.
- **ifpresent** to check for the presence of an object.
- **waitfor** to suspend the **init** program until an object exists.
- **auto** to enter a name into the name table.
- **console** to specify the current console.

In order to run, most programs need various pieces of information in their environment which are sent by the parent program, in this case the **init** program.

1. Standard I/O streams **stdin**, **stdout**, **stderr** in C, or units 5 and 6 in Fortran.
2. A current directory.
3. A vector of arguments, possibly empty.
4. A set of environment strings which may be used to store any additional information.

When **init** runs other programs the standard streams are initially set up to be the error logging server **/logger**, which is usually provided by the I/O server. The current directory is set to **/helios**, which must be present because the **init** program is **/helios/lib/init** and the **initrc** file is **/helios/etc/initrc**. The arguments are provided by the **run** command, and the set of environment strings is empty. The syntax of **run** is as follows:

```
run [-e] [-w] <command name> [argument 0] [argument 1] ...
```

There are two optional arguments which must come before the command name. The first, **-e**, causes **init** to send an environment to the specified program. It is possible to produce programs which do not require an environment, for example the Helios ram disc, but these are the exception rather than the rule. The **-w** option causes **init** to wait until the program has terminated. By default **init** will continue as soon as the program starts running, and **init** itself will terminate as soon as the last statement in the **initrc** file has been executed. Following these optional arguments comes the command name, which must be a complete path name.

If the **-e** option is used to indicate that an environment should be sent, then the command name must be followed by one or more arguments. The zeroth argument is conventionally the program's name. However, some programs such as **login** use this argument to determine that the program was started by the bootstrap process rather than from a shell. The **login** program checks that this argument is '-' rather than **login**, for example. The zeroth argument may be followed by additional arguments if desired. All programs run by **init** run on the root processor. To execute programs on remote processors the **remote** command can be used, for example:

```
run -e /helios/bin/remote remote -d 01 /helios/lib/fs raw
```

This would run the **remote** command on the root processor, sending it an environment because it is an ordinary program rather than a special system program. Argument zero is **remote**, quite reasonable since that command like most others does not distinguish between running during the bootstrap phase and running in a user session. The additional arguments are: **-d** for detach, to indicate that **remote** should not wait for the program to terminate; `01` for the target processor; **/helios/lib/fs** for the program to execute; and **raw** as an argument for that program.

The reader should be aware that when the network is running in a protected mode using **remote** may fail. The remote processor will be protected such that no user other than the current owner can access it, and the **init** program does not have any special privileges for executing programs. Whether or not a network is running in protected mode is controlled by the **nsrc** file. Usually it is better to make use of **run** commands inside the resource map, which avoids these problems. The remaining commands understood by **init** are rather more simple. The **console** command is used to specify alternative standard streams for subsequent **run** commands. The exact syntax is as follows:

```
console <server name> <window name>
```

First **init** will attempt to locate the specified server. It is assumed that this provides a terminal window interface, but that is not essential. Next it will attempt to create the specified window if any, and if successful this window will be used for standard streams from now on. Typical ways of using the command are:

```
console /window mywindow
console /termserver console
```

The first command creates a new window **mywindow** within the server **/window**. The second creates a new window **console** within the server **/termserver**.

The **ifabsent** and **ifpresent** commands are the only conditions which can be used in an **initrc** file. Both commands take as their first argument the name of an object, which might be a server name, a file name, a processor name, or any other object within the Helios world. This is followed by another **initrc** command, usually but not always a **run** command. In the case of **ifabsent** this second command will be executed if and only if the specified object does not exist. In the case of **ifpresent** the command will be executed if the specified object does currently exist. Typical ways of using these commands are:

```
ifabsent /fs run -e /helios/lib/fs fs raw
ifabsent /lock auto /lock
ifpresent /helios/lockfile run -e /helios/bin/rm rm /helios/lockfile
```

The **waitfor** command is used to suspend the **init** program until an object exists. **init** will attempt to locate the object at intervals of one second. The command is usually used to wait for a server to start up or for a processor to be booted, but is not restricted to this. For example, a previous program might create a particular file when it has done a certain amount of work, and the **waitfor** command can be used to wait until that file exists.

```
waitfor /sm
waitfor /Cluster/07
waitfor /helios/lockfile
```

The **auto** command is used to create an entry in the name table. Certain Helios servers such as the ram disc and the null device are loaded automatically as soon as an attempt is made to access them. This is achieved by creating a suitable entry in the processor's name table, and the **auto** command can be used to do this. **auto** takes a single argument, the name of the server. For example:

```
auto /lock
```

would enter the name **/lock** in the root processor's name table, and cause the program called **/helios/lib/lock** to be run automatically when any attempt is made to access the lock server. The server is started up without an environment, so it is unlikely to be particularly complicated. Helios does this automatically for the **/ram**, **/pipe**, **/fifo** and **/null** servers on every processor.

2.6.3 .login, .cshrc, and .logout

The normal user interface used with Helios is a shell, or a number of shells in separate windows. These shells read in a set of files containing shell commands, which may be useful when configuring the networking software. The **.cshrc** file is read by every shell when it starts up, and users can start up any number of shells either explicitly or implicitly as the result of other commands. Hence the **.cshrc** file is not very useful for networking purposes. However the **.login** file is read only by the first shell to be started for a user, the login shell. Similarly the **.logout** file is read only by the login shell. Hence these provide a fairly simple way of executing networking commands on a per-session basis. In a single-user environment it may be desirable to obtain all the processors in the network as soon as the bootstrap process has been completed and a user session has been started. This can be achieved by a call to the **domain** program in the **.login** file.

```
domain get /00 /01 /02 /03 /04 /05 /06 /07
```

Even in a multi-user environment, it is often desirable to pre-allocate a small number of processors.

```
domain get 2
```

This command would attempt to obtain two processors satisfying the default requirements. The **.login** file is particularly useful for the special user `shutdown`. When a user logs in with that user id, in order to shut down the network, Helios will start up a Task Force Manager and a shell for that session as usual. Hence the **.login** file will be executed in the directory `/helios/users/shutdown`. For a simple network, this file might contain a single line.

```
stopio /maryPC
```

This runs the **stopio** program, making it send a terminate message to the I/O server running on processor /maryPC. If there is only a single I/O processor in the network then this suffices for shutting down the network, and the I/O processor will return to the host operating system. For more complicated multi-user networks it may be desirable to have a more complicated file.

```
wall << end
The system is going down in five minutes.
end
sleep 240
wall << end
One more minute until the system goes down.
end
sleep 60
wall << end
The system is now going down
end
sleep 5
stopio /jonPC
stopio /nickPC
termfs /fs
stopio /BootPC
```

Alternatively, it may be desirable to start up a normal shell session, execute some of these commands interactively to give other users a chance to request a delay before the shutdown occurs, and perform the final shutdown commands in the **.logout** file.

2.6.4 nsrc

The **nsrc** file contains a list of options for the networking software, like the **host.con** file which has a list of options for the I/O server. The **nsrc** file is read by the **startns** program when networking software is started up, and passed in the environment to the network server and/or Session Manager. By default **startns** reads the file **/helios/etc/nsrc**, but an alternative filename can be specified on the command line. A typical **nsrc** file might look like this:

```
#
# This is a comment
#
single_user
#password_checking
#processor_protection
#no_taskforce_manager
share_root_processor
#root_processor =/06
waitfor_network
preload_netagent
```

Again, the # symbol can be used to indicate a comment. It is also rather useful when disabling or re-enabling an option, because it means that there is only one character to be added or deleted. The various options have the following meanings.

no_taskforce_manager

In a single-user system with a network of just one or a small number of processors, having a Task Force Manager for that user may not be necessary. However it may still be useful to have a Session Manager to allow multiple users to make use of the network at different times, or to enforce password checking. With the **no_taskforce_manager** option the Session Manager will not start up a Task Force Manager when the user logs in. Instead it will execute the default command from the password file, usually a shell, on the root processor.

In a multi-user system, using this option allows several users to log in and share the same processor. Hence a multi-user system is possible even if there is only one processor in the network, but this is not recommended.

password_checking

In a given network it may or may not be desirable to force users to quote passwords when they start a session. If passwords are in use then the **password_checking** option must be enabled. Please note that password checking must be enabled or disabled on a global basis. When an ordinary user logs in, they may not need a password if there is no entry in the password file. When a new user is added to the system, the system administrator could decide whether or not to give them a password.

preload_netagent

The network server frequently needs to perform a complicated operation on various processors in the network, such as cleaning out unnecessary libraries from the Loader when a processor is returned to the system free pool by a Task Force Manager. To do such jobs the network server will run a little program, **/helios/lib/netagent**, on the required processor. For small networks there is very little overhead in loading this program off disc every time it is required. For large networks loading off disc is inefficient and it is better to keep the network agent permanently loaded in memory. To do this, the **preload_netagent** option should be used.

processor_protection

Helios can run either in a protected mode or in an open mode. In a protected mode users will be completely unable to access each other's processors, or processors in the system pool, unless the current owner explicitly gives access. In an open mode users can access each other's processors explicitly, unless the owner has denied access to all other users. However, users must force programs to run on each other's processors. The difference between the two modes can be illustrated with the **remote** command. In protected mode attempts to remotely access a processor will fail, unless that processor is currently in the user's domain or the user has been given a capability for that processor by its owner. In open mode attempts to execute commands remotely will succeed, unless the owner of the processor has explicitly denied access to the processor. The **domain** command can be used to set protection modes on processors.

root_processor = /Net/ClusterA/06

For very complicated networks the networking software may occasionally have difficulty working out on which processor it is supposed to run the network server and Session Manager. Should this happen, it may be necessary to give the full name of the root processor in the **nsrc** file. This option need not be used unless the networking software produces an error message that it cannot determine the root processor.

share_root_processor

In a single-user system the question arises as to whether the Task Force Manager can run on the same processor as the network server and Session Manager (the root processor) or must run on a different one. Note that the login shell will also run on this processor. The only reason for not sharing the root processor is a shortage of memory. In a multi-user network the network server always reserves its own processor, and never allows users to access this processor.

single_user

By default a Helios network is assumed to be a multi-user network. Multi-user networks are more restrictive than single-user ones, because in a multi-user environment the networking software has to take care to protect users from each other. Hence, for example, the network server cannot be made to exit in a multi-user environment. The **single_user** option can be used to put the network into the less secure mode. The option must be enabled by the user.

waitfor_network

When initialising a network there are two important stages: booting up the network; and starting user sessions. It is not possible to start a user session until the network has been fully booted. Attempting to log in before then will give an error message, **insufficient network resources available**. For small networks the time taken for booting the network is comparable to the time taken to start up the Session Manager, register the current window, and run login. Hence the initialisation process can continue while the network server is booting up the network, and everything is ready at about the same time. For large networks this is not true. The time taken to boot up a network can be considerable, and hence there can be a significant delay between the networking software starting up and the time when a user is able to login. To provide synchronisation, the **waitfor_network** option can be used. This option delays the startup of the Session Manager until the whole network has been booted, and hence no login prompts will appear until the network is fully booted.

2.6.5 Network resource maps

A network resource map is a text file describing the available network hardware. Networks can be very complicated, and hence a special language is used to allow users to specify their networks. Helios comes with a resource map compiler **rmgen** which parses the resource maps, performs validation checks, and produces a binary object file which is used by the networking software. By convention the textual form is given the suffix **.rm** and the binary form the suffix **.map**.

In theory producing textual resource maps is not the only way to specify a network. Other possible techniques are: a graphical editor which allows users to draw the network; a worm program that explores an existing network; and network generators that can specify standard topologies automatically. Unfortunately all these approaches have disadvantages. A graphical editor will be tied to a particular graphics system, probably the X window system, that may not be available on the user's hardware. A worm program can fail if the network supports link configuration, because most processors cannot be accessed until the networking software has set up the links. Also with distributed reset schemes such as the Parsytec one, triggering a worm in a multi-user environment can be disastrous. Generating network topologies automatically is fine, but does not supply the required information about reset and configuration facilities, nor can it specify that say a file server should be run automatically on a particular processor that is equipped with a SCSI interface. Textual resource maps, though perhaps more difficult to use, provide greater functionality. Consider the network of Figure 2.29.

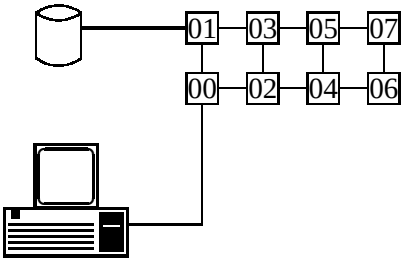


Figure **2.29** A simple network

A resource map for this might be:

```
# This is a comment
network /Cluster {
  Reset { driver; ~00; tram_ra.d}

  processor 00 { ~IO,    , ~01, ~02; }
  processor 01 { ~00,    ,    , ~03; run -e /helios/lib/fs fs scsi; }
  processor 02 {    , ~00, ~03, ~04; run /helios/lib/lock; }
  processor 03 { ~02, ~01,    , ~05; }
  processor 04 {    , ~02, ~05, ~06; }
  processor 05 { ~04, ~03,    , ~07; }
  processor 06 {    , ~04, ~07,    ; }
  processor 07 { ~06, ~05,    ,    ; }
  processor IO { ~00; IO }
}
```

Resource maps may contain the following information:

- The network name, or a hierarchy of network names.
- Descriptions of the processors.

- Specification of the reset driver.
- Specification of the configuration driver.
- Additional reset facilities that might be available.

In addition lines beginning with a # are treated as comments and ignored. Blank space is also ignored, and resource maps are not case sensitive except when specifying names.

Network names and hierarchies

A resource map must contain the following:

```
network <name> { <network description> }
```

Any data following the closing curly bracket is ignored. The keyword **subnet** is an alias for **network**. A name can consist of any combination of letters, digits, and the underscore character (_). Names can be up to 31 characters long, and are case sensitive. Hence network `/Cluster` is different from network `/cluster`. Names must not match with any of the resource map syntax keywords. For most networks there is no need for a hierarchy of network names. However, with mixed networks containing different reset schemes it may be useful. For example,

```
network /Cluster {

    processor 00 { ~IO,    , ~01, ~02; }

    subnet /subnetA {
      reset { driver; ~00; tram_ra.d }
      processor 01 { /Cluster/00,    ,    , ~03;
                     run -e /helios/lib/fs fs scsi; }
      processor 02 {    , /Cluster/00, ~03, /Cluster/subnetB/04;
                     run /helios/lib/lock; }
      processor 03 { ~02, ~01,    , /Cluster/subnetB/05; }
    }

    subnet /subnetB {
      reset {driver; ; pa_ra.d }
      processor 04 {    , /Cluster/subnetA/02, ~05, ~06; }
      processor 05 { ~04, /Cluster/subnetA/03,    , ~07; }
      processor 06 {    , ~04, ~07,    ; }
      processor 07 { ~06, ~05,    ,    ; }
    }

    processor IO { ~00; IO }
}
```

In this network processors `00` and `IO` are at the top level. Processors `01`, `02`, and `03` are in `/Cluster/subnetA`, and controlled with one reset driver. The remaining processors are to be found in `/Cluster/subnetB`, and are controlled with a different reset driver. The naming tree for such a network is shown in Figure 2.30.

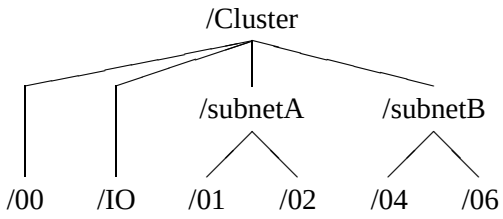


Figure **2.30** Hierarchical network names

Inside the network server, the **/ns** directory would contain two objects `00` and `IO`, and two subdirectories `clusterA` and `clusterB`. These subdirectories would contain the appropriate processor objects. In fact it is unnecessary to give unique network names to the subnetwork names. The subnets can be left unnamed, simply by using curly brackets.

With this second syntax the naming tree is straightforward again. There is one network level, one processor level, and a server level within the processors. Similarly, the network server's **/ns** directory would contain all processors at the top level, without any subdirectories.

```
network /Cluster {

    processor 00 { ~IO,    , ~01, ~02; }
    {
      reset { driver; ~00; tram_ra.d }
      processor 01 { ~00,    ,    , ~03;
                     run -e /helios/lib/fs fs scsi; }
      processor 02 {    , ~00, ~03, ~04;
                     run /helios/lib/lock; }
      processor 03 { ~02, ~01,    , ~05; }
    }
    {
      reset {driver; ; pa_ra.d }
      processor 04 {    , ~02, ~05, ~06; }
      processor 05 { ~04, ~03,    , ~07; }
      processor 06 {    , ~04, ~07,    ; }
      processor 07 { ~06, ~05,    ,    ; }
    }
    processor IO { ~00; IO }
}
```

The two resource maps have exactly the same effect, but the first syntax gives longer processor and server names. Hence for most networks the second syntax is preferable, if only because it involves less typing. For both resource maps there would be only one network server in the network, usually running on processor `00`. This network server would have two device drivers loaded, `tram_ra.d` and `pa_ra.d`.

Processor connectivity

Inside a given network or subnetwork the resource map should contain one or more processor descriptions, and possibly details of the networking hardware. A processor

description takes the following form:

```
processor <name> { <connections>; <options> }
```

An example might be:

```
processor 00 { ~IO, ,~02,~03; System }
```

The keyword **terminal** can be used as an alias for **processor**. Again, the name can be a combination of letters, digits, and the underscore and character up to 31 characters long. The name is case sensitive. A typical list of connections would be:

```
~01, /Cluster/subnetB/02, Ext, ;
```

This indicates that link `0` of the processor is connected to a processor `01` at the same level of the network hierarchy. The `~` character is short for the current network or subnetwork name. If the processor has a full name `/Cluster/xx`, then this means that `~01` is equivalent to `/Cluster/01`. If the network has a hierarchy of network names and the full processor name is `/Cluster/subnetA/00` then this means that the shorthand `~01` will be equivalent to `/Cluster/subnetA/01`. It is always possible to use the fully expanded form in place of the shorthand form. Link 1 is connected to processor `/Cluster/subnetB/02`. That processor has a different network base name from the current processor, so the `~` shorthand cannot be used. Link 2 is an **external** link. This means that there may be another processor running Helios or a Helios network at the other end of the link, now or at a future stage. Hence the network server puts the link into pending state, waiting for the other side to connect in. This can happen when a user in the remote network uses the **elink** or **clink** commands. Links not specified in the resource map as external ones can be put into pending mode explicitly, using the **plink** or **clink -p** commands. Link 3 is shown as not connected. Following the link connection there may be an optional number enclosed in square brackets.

```
~01[1], ~01[2], Ext[12], ;
```

There are two connections between the current processor and processor `01`. The extra numbers in the square brackets indicate the destination link on the remote processor, so link 0 of this processor is connected to link 1 of the remote processor. In the case of external links the meaning is different. The number is used only with certain machines such as the Telmat T.Node, to indicate which external connector on the backplane of the machine should be used. Since this connection has to be made through the electronic switch it is important to specify exactly which connector to use.

If there is an I/O processor at the other end of a link with just a link adapter, then when the I/O server is started up on that processor it will merely enable the link. The network server must distinguish this case from another Helios network enabling the link, because the action required is different. Hence the I/O processor must be specified in the resource map, instead of leaving the link as an external one.

For most hardware the network resource map specifies the actual topology of the network. If the resource map indicates that link 0 of a processor is connected to another processor, then that connection really exists. It may be hard-wired, or it may be set up by the networking software using a link switch. However, given hardware with restricted link switching such as the Telmat T.Node this may not be true. If the resource map indicates that link 0 of a processor is connected to another processor then a link will be connected to that processor, but it does not have to be link 0.

Processor options

Following the link connections in a processor description there can be a number of additional options. The various options should be separated by semi-colons and terminated with the curly bracket that finishes the processor description. The following options are available.

- A **mode** field. This can be used to specify one of four processor modes.

  1. **Helios**. The default. This is a normal processor which can be allocated to users for running applications.
  2. **IO**. An I/O processor. I/O processors cannot be used for running applications. Also, I/O processors are never booted by the network server. There is usually one I/O processor responsible for performing the first stage of the network bootstrap, and additional ones connect into the network.
  3. **System**. The processor is reserved for use by the system. It cannot be allocated to users, and hence it is usually impossible to run applications there. In an unprotected network programs can be placed explicitly using the **remote** command. System mode is usually used with the **run** option to run just one program such as a file server on that processor.
  4. **Native**. This processor should never be booted, and will not be used by Helios. It may be necessary to incorporate it into the resource map in order to make the link connections, if the hardware includes a link switch.

  The processor mode can be specified simply by listing it.

```
processor 00 { ~IO, ~01, , ; System; }
processor 01 { ~00 , , , ; Helios; }
```

- A processor **type**. This is used to control the default Nucleus to be booted into the appropriate processor, and the bootstrap mechanism. Once the processor is up and running the network server will verify that the processor type specified is correct, and if necessary it will give warnings. The real processor type rather than the specified one will be used when allocating processors, so if a user requests four T800s that is what will be supplied, no matter what the resource map says. The processors recognised by **rmgen** are:

  1. **T800, T805, T414, T425, T400**, these are actually equivalent because the same bootstrap mechanism and the same default Nucleus is used for all of them.
  2. **T212**, for 16-bit processors. Helios cannot run on a 16-bit processor, so these processors must always be native ones.
  3. **680x0**, used with Helios running on any of the 680x0 family.
  4. **ARM**, for Helios running on any version of the ARM[7].
  5. **i860**, for Helios running on any version of the i860.
  6. **T9000**, in preparation for the Inmos T9000.
  7. **320C40**, in preparation for the Texas Instruments[8] TMS 320C40

[7] Trademark of Acorn Computers Ltd
[8] Trademark of Texas Instruments, Inc.

The **ptype** keyword should be used to specify the processor type. In this context **processor** can be used as an alias. Typical examples are:

```
processor 00 { ~IO, ~01, , ; ptype T800 }
processor 01 { ~00, , ,  ; ptype T400 }
```

- A **memory size**. With most hardware Helios is perfectly capable of working out how much memory there is on a processor, and the network server will obtain this information when the processor has been booted. However, with some hardware the memory map may be arranged strangely. In particular there are graphics boards with one or several megabytes of normal processor memory, immediately followed by a megabyte or so of video memory. It is extremely difficult for software to distinguish between the types of memory, so Helios will use video memory for its memory allocation. The resulting display can be very interesting but is not usually what is desired. Helios can be made to skip the phase determining the memory size, by specifying the actual amount of memory in the resource map. For the root processor this must be done in the **host.con** file, using the **transputer_memory** option. Memory sizes can be specified in hex, decimal or octal.

```
processor 01 { ~00, , ~02, ; memory 1048576 }
processor 02 { ~01, ,    , ; memory 0x100000 }
```

- A **Nucleus**. In most networks the standard Helios Nucleus, **/helios/lib/nucleus**, should be booted into every processor in the network. This Nucleus should be present already on the processor doing the booting, so there is no need to fetch it off disc every time. However, in very special cases it may be necessary to boot a different Nucleus into the processor and hence the networking software provides an option.

```
processor 01 { ~00, , , ; nucleus /helios/lib/nucleus.fs }
```

  For processors other than Transputers the argument is an arbitrary string interpreted by the appropriate bootstrap software. Usually, but not always, this will be a file name.

- **Programs** to run on that processor. The network server can be made to run software automatically on particular processors, using the **run** option. Once the programs are up and running the network server ignores them, so it does not matter whether or not they exit. Hence the facility is useful for 'once only' initialisation programs and for permanent servers. If the processor has to be rebooted, the program will be run again. Any number of programs can be run in this way. The syntax is the same as used by the **initrc** file.

```
processor 01 { ~00, , , ~03; System;
               run -e /helios/lib/fs fs raw }
processor 02 { , ~00, ~03, ~04 ; run /helios/lib/lock }
```

This facility provides a fairly simple way of starting up a network such that all the software required runs as soon as possible. However, the **initrc** file is used to run programs on the root processor whereas the resource map makes it easy to run programs on particular processors. The programs are executed as soon as the processor has been booted, so they can be used by network device drivers if required. This is useful when booting mixed networks.

- Additional **attributes**. The options described so far should suffice for most networks. However, to give users maximum flexibility it is possible to define arbitrary string attributes as well. These strings are not used directly by the networking software. However, it is possible to request processors with a specific attribute and the networking software will try to find one. Typical examples might be:

```
processor 01 { ~00, , , ~03; attrib 30Mhz }

domain get "{ attrib 30Mhz }"
```

Reset and configuration drivers

Performing resets in a homogeneous network is fairly easy. **Device drivers** are available for the most common hardware architectures, and these can be specified in the resource map. Link configuration drivers can be specified in the same way.

```
Reset { driver; ~00; tram_ra.d }
Configure { driver; ; telmat_c.d }
```

Drivers are specified by the keywords **Reset** or **Configure**, depending on the driver purpose. By convention, reset drivers end with **_ra.d** or **_r.d**, and configuration drivers end with **_c.d**. Following the keyword are three arguments, enclosed in curly brackets. The first argument should be the keyword **driver**. In the case of **Reset** this first argument may be a list of processors. The second argument is a string of some sort, that will be passed to the device driver. Usually this string is the processor in the network that has the actual reset hardware attached, but device drivers are free to interpret the string in any way. The final argument is the device driver file name. This can be an absolute file name, for example `/c/drivers/myrst_ra.d`, but by default refers to a file in the **/helios/lib** directory. Within a given network or subnetwork there may be only one device driver, and the network server will invoke this device driver for the processors in this network. For example,

```
Network /Cluster {
     Reset { driver; ; pa_ra.d }

     subnet /subnetA {
       Reset { driver; ~06; tram_ra.d }
     }

     subnet /subnetB {
       Reset { driver; ; telmat_r.d }
     }
}
```

Processors at the top level are controlled using the Parsytec reset driver. Processors within **subnetA** are controlled using the TRAM reset driver, and the network server will never attempt to reset these using the Parsytec scheme. The TRAM reset driver will be passed the string **/Cluster/subnetA/06**, presumably the processor equipped with the TRAM subsystem control hardware. Similarly processors in **subnetB** are controlled only using the Telmat scheme.

The following device drivers are available at present.

1. **tram_ra.d**, the reset driver for the Inmos TRAM scheme. The only facility supported by this driver is a global reset of all processors under its control. The driver can take an optional argument specifying the processor with the subsystem control hardware, defaulting to the root processor. In mixed networks it may be necessary to specify a processor other than the default.

2. **pa_ra.d**, the reset driver for the Parsytec reset scheme. This supports an individual reset for all processors that currently have active Helios neighbours. No argument is required.

3. **telmat_r.d**, the reset driver for the Telmat T.Node which is supplied by Telmat Informatique. It supports an individual reset for all processors.

4. **telmat_c.d**, the configuration driver for the Telmat T.Node, again supplied by Telmat Informatique.

5. **rte_ra.d**, a reset driver for use on the Meiko Computing Surface. This driver does not require any additional arguments.

6. **rte_c.d**, a configuration driver for the Meiko Computing Surface. Again this driver does not require any additional arguments.

Reset drivers only work within a subnet. This causes problems in mixed networks. If the Parsytec scheme is used within one subnet then it is necessary to reset one processor within this subnet in order to reset and boot the rest. This processor must be reset without using the Parsytec reset scheme, since the processors outside the subnet do not support it. Hardware can usually be rearranged to give reset on this processor, possibly with a bit of soldering, but the network server needs to be informed about this. The user can specify commands which, when run on a particular processor, reset one or more other processors to support such mixed. networks.

Mixed networks and additional resets

To support mixed networks, networks containing hardware supplied by more than one manufacturer and using different reset schemes, users can specify additional reset facilities in the resource map. The syntax is similar to that for reset drivers, but specifies one or more processors, instead of the keyword **driver**, as the first argument. The second argument gives the processor on which the reset command is to be executed. If omitted the command will be executed on the root processor. The third argument is the actual command, using the same syntax as the **initrc** file and the **run** option in a processor description.

```
Reset { ~01, ~02, ~03; ~00; run -e tr_reset tr_reset}
```

This line specifies that running the **tr_reset** program on processor `00` will reset processors `01`, `02`, and `03`. Similarly,

```
Reset { ~05; ~04; run -e pa_reset pa_reset 3 }
```

specifies that it is possible to reset processor `05` individually by executing the **pa_reset** command on processor `04`. To illustrate the way this can be used in practice, consider Figure 2.31.

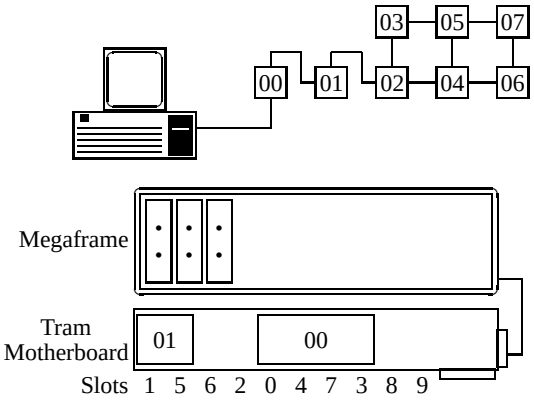


Figure **2.31** A mixed network

Processors `00` and `01` are TRAM modules on a suitable motherboard, and the remaining processors are part of a Parsytec MultiCluster. The TRAM reset is passed on to the first processor in the megaframe, so that whenever the global reset is asserted on processor `00` this affects processor `01`, the other TRAM module, and processor `02`, the first MultiCluster processor. The Parsytec reset scheme can be used on the remaining processors.

A possible resource map for this is:

```
Network /SlightlyUnusual {
   Reset { ~01, ~02; ; run -e tr_reset tr_reset }

   processor 00 { ~IO, , ~01, ; }
   processor 01 { , ~00, ~02, ; }
   { Reset { driver; ; pa_ra.d}
     processor 02 { , ~01, ~03, ~04; }
     processor 03 { ~02, , , ~05; }
     processor 04 { , ~02, ~05, ~06; }
     processor 05 { ~04, ~03, , ~07; }
     processor 06 { , ~04, ~07, ; }
     processor 07 { ~06, ~05, , ; }
   }
   processor IO { ~00; IO }
}
```

No reset driver is specified for the top level. In fact the **tram_ra.d** driver could have been specified but this would not have given any greater flexibility. Using the

tr_reset program on the root processor will reset processors `01` and `02`. There is an unnamed subnet containing processors `02` to `07`, and the Parsytec reset driver can be used within this subnet. This resource map describes the available reset hardware, and allows the network server to boot up such a network reliably.

Another common mixed subnet would be a Telmat T.Node as the network backbone, but with a TRAM based workstation as the front-end. Assuming a similar topology to the above, the resource map would be:

```
Network /Possibility {
   Reset { ~01, ~02; ; run -e tr_reset tr_reset }

   processor 00 { ~IO, , ~01, ; }
   processor 01 { , ~00, ~02, ; }
   { Reset { driver; ; telmat_r.d }
     Configure { driver; ; telmat_c.d }

     processor 02 { , ~01, ~03, ~04;
                    run -e /helios/lib/tcontrol tcontrol }
     processor 03 { ~02, , , ~05; }
     processor 04 { , ~02, ~05, ~06; }
     processor 05 { ~04, ~03, , ~07; }
     processor 06 { , ~04, ~07, ; }
     processor 07 { ~06, ~05, , ; }
   }
   processor IO { ~00; IO }
}
```

Now the Telmat reset and configuration drivers will be used within the unnamed subnet, and in addition the **tcontrol** program will be executed on processor `02` as soon as it is booted. This **tcontrol** program is a server to interface to the **internode controller**, and is accessed by the device drivers.

A third network might have perhaps ten Parsytec Transputers making up the workstation and some additional processors, attached to a Telmat T.Node. The resource map for that would be something like:

```
Network /VeryConfused {
   Reset { ~10; ~09; run -e /helios/netbin/pa_reset pa_reset 2 }

   { Reset { driver; ; pa_ra.d }
     processor 00 { ... }
     processor 01 { ... }
        ...
     processor 09 { ... }
   }
   { Reset { driver; ; telmat_ra.d }
     Configure { driver; ; telmat_c.d }
     processor 10 { ...; run -e /helios/netbin/tcontrol tcontrol }
     processor 11 { ... }
     processor 12 { ... }
   }
   processor IO { ~00; IO }
}
```

There is an individual reset available for processor `IO`, using the **pa_reset** program. Processors `00` to `09` are controlled using the Parsytec reset driver. Processors `10` onwards are controlled using the Telmat reset and configuration drivers. Provided that adequate hardware reset facilities are available it should be possible to define them in the resource map. The standard hardware reset programs supplied with Helios will suffice for most of the networks, but users can write their own if needed.

Formal syntax

An outline of the formal syntax of resource maps is shown below. Lexical tokens are enclosed in quotes, and are not case sensitive. Optional items are enclosed in square brackets.

```
<Resource Map>  ::= 'network' <address> '{' <network>  |
                    'subnet'  <address> '{' <network>

<network>       ::= '}' |
                    'reset' <reset> <network>         |
                    'configure <configure>  <network> |
                    'processor' <processor> <network> |
                    'terminal'  <processor> <network> |
                    '{' <network> |
                    'network' <address> '{' <network> |
                    'subnet'  <address> '{' <network>

<reset>         ::= '{' 'driver' ';' [string] ';' <file> '}' |
                    '{' <list>   ';' <proc_id> ';' 'run' <command>

<configure>     ::= '{' 'driver' ';' [string] ';' <file> '}'

<processor>     ::= <name> '{' <list> ';' <description>

<description>   ::= '}' |
                    ';' <description> |
                    'helios' <description> |
                    'system' <description> |
                    'native' <description> |
                    'IO'     <description> |
                    'ptype'  <ptype>    <description> |
                    'processor' <ptype> <description> |
                    'memory' <size>     <description> |
                    'nucleus' <file>    <description> |
                    'run' <command>     <description> |
                    'attrib' <string>   <description>

<ptype>         ::= 'T800' | 'T414' | 'T425' | 'T400' |
                    'T212' | 'T222' | 'M212' |
                    'ARM'  | 'i860' | '68000' | 'T9000' | '320C40'

<list>          ::= <proc_id> [ ',' <list> ]
```

```
<proc_id>       ::= '~' <name> | <fullname>

<address>       ::= '/' <name>

<fullname>      ::= '/' <name> [ <fullname> ]

<file>          ::= <fullname>

<command>       ::= [-e] <string> [string] [string] ...

<string>        ::= a sequence of characters

<size>          ::= a number in hex, decimal, or octal

<name>          ::= a sequence of letters, digits, and underscores
                    not exceeding 31 characters
```

2.7 Configuring networks

Section 2.3 described many different types of processor networks that can be used with Helios, giving an outline of what is required but no details of the commands or configuration files. Section 2.4 explained why networking can be difficult, because of the range of hardware available. Section 2.5 described the various networking commands that can be used. Section 2.6 gave details of the configuration files. This section will repeat most of the networks of section 2.3, this time giving details of how to configure all the networks. Where a network's configuration is similar to a previous one only the differences will be given. It is hoped that the reader will recognise at least one of the networks as the appropriate one, given the available hardware and the user's requirements.

2.7.1 Single-processor workstation

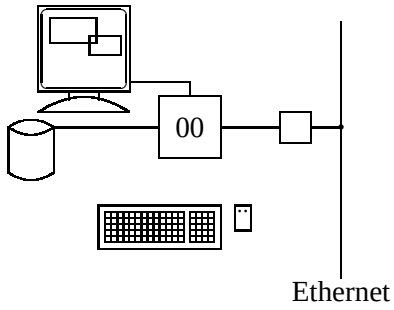


Figure **2.32** A single-processor workstation

A single processor may be equipped with a ROM bootstrap mechanism, a hard disc, a graphics display, an ethernet connection, keyboard, and mouse, to give a complete single-processor workstation. There is no need for a network server, since there is only one processor, and there is no need for a Task Force Manager to administer the user's domain of processors and run programs within that domain.

There is no I/O processor, and hence no I/O server, so the **host.con** file is not used. The next configuration file is the **initrc** file. As a result of the ROM bootstrap the filing system must start up and interact with the hard disc, which will be followed by the Nucleus running the **init** program. A possible **initrc** file might be:

```
#
# Get the X server and Terminal emulator up
run -e /helios/bin/xhelios -newXrc=/helios/etc/Xrc
run -e /helios/lib/window window
# Direct output to one of these windows
waitfor /window
console /window console
#
# The user now has a chance to see what is happening
#
# The mouse and keyboard are attached to serial ports
run -e /helios/lib/rs232 rs232 com1.8250.10000000 com2.8250.10000008
run -e /helios/lib/keyboard keyboard /rs232/com1
run -e /helios/lib/mouse mouse /rs232/com2
#
# Ethernet software should be next. This involves the TCP/IP
# server and the internet daemon
run -e /helios/lib/tcpip tcpip jon 91.0.0.111
run -e /helios/lib/inetd inetd
#
# Start up a Session Manager, but not a network server
run -e /helios/bin/startns startns -nons
waitfor /sm
#
# and create a user session
run -e /helios/bin/newuser newuser
```

The hardware is started up step by step, in order of importance. The hard disc must be running already, or the system would not have got this far. Screen output is the next most important because until a terminal system is up there is no way to output diagnostics to the user. This requires the X server, with a specification of the **Xrc** configuration file to use. If the **initrc** file is changed such that the X server is not run then the machine will not be able to display any output. This is unfortunate, because the machine is now unusable. Usually the only good reason for changing the **initrc** file is to support additional hardware, an infrequent occurrence, and the user will have to be careful.

In addition to the X server it is necessary to start up the terminal emulator, which is a client of X. By default the X server simply initialises and clears the screen, displays a mouse cursor, and waits for clients to connect in. The terminal emulator is a Helios server that installs itself in the name table as **/window**, and waits for its clients. The **console** command creates a new terminal window, and redirects the **initrc** output to this window. The terminal emulator interacts with the X server to make this window visible. It is now possible for the user to get diagnostic information.

In addition to a graphical output device the X server needs keyboard and mouse inputs. These devices could be plugged into serial ports attached to the Transputer. The

initrc file runs an rs232 server to control these serial ports, and mouse and keyboard servers which interact with the rs232 server. Next come the commands to start up the ethernet software, including all the TCP/IP support and the required daemons. The ethernet software will read some configuration files of its own to specify appropriate options.

All the hardware has now been accounted for, so it is possible for a user to log in. Logging in requires a Session Manager, so the **startns** command is used. The **-nons** option suppresses starting up the network server, since there is nothing for it to do. Without a network server there is no need to worry about the **-r** option or the resource map. The Session Manager will start up after a short delay, and then the **newuser** command is used to create a new session. No name is specified, so the user has to type in a name. Depending on the **nsrc** file, a password will be required as well. If desired the **initrc** line could read:

```
run -e /helios/bin/newuser newuser mary
```

If password checking is not enabled a session will be created for user `mary`. If password checking is enabled the **login** program will echo this name and prompt for the password. One of the user ids should be `shutdown`, to terminate all the software, synchronise the hard disc, and allow the workstation to be powered down without loss of data.

The next file to consider is the **nsrc** file. This would be something like:

```
#
# This is a comment
#
single_user
#password_checking
#processor_protection
no_taskforce_manager
share_root_processor
#root_processor = /00
#waitfor_network
#preload_netagent
```

The network is a single-user network. There is only one processor and every user in a network needs at least one processor. Password checking is disabled. Some options, such as the **waitfor_network** and **processor_protection** options, are only interpreted by the network server, and no network server is run in this network. The **no_taskforce_manager** line forces the Session Manager to start up a shell on the local processor, rather than to start up a Task Force Manager for the user and create a shell within its **/tfm** directory.

It may be desirable to separate the window used for hardware diagnostics from the first window used for the user session. This can be done very easily. The first **console** statement should be replaced by:

```
console /window diagnostics
```

and just before the **newuser** command there should be a line

```
console /window console
```

to create a second window for the user's session. Please note that under the X window system it is necessary to run a separate **Window Manager** program to allow positioning and repositioning of windows. The Ultrix[9] Window Manager or **uwm**, and the TAB Window Manager or **twm**, are shipped with the Helios X window system, but various other Window Managers exist.

There is no need to write a network resource map, since this is used only by a network server and no network server is started up. The other important file to consider is the **.logout** file for user `shutdown`. This must contain commands to shut down the whole network cleanly. For this machine shutting down means synchronising and terminating the file server.

```
termfs /fs
echo Disks synched
echo The system may be powered down.
```

2.7.2 Workstation with I/O processor

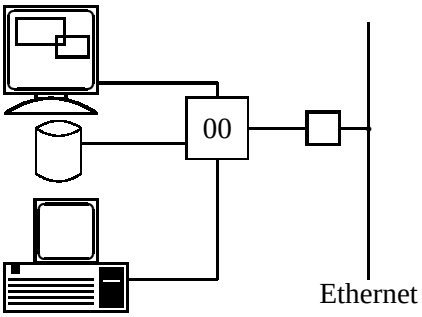


Figure **2.33** A workstation with I/O processor

It is possible to build a Transputer workstation based around an I/O processor such as a PC. The network can start off with just the PC and one processor, and can be expanded gradually. An I/O server must be run on the I/O processor, so a **host.con** file is required. Amongst the options might be:

```
# The host.con file
Server_windows
logging_destination = both
#root_processor = /tom
#io_processor = /pc
#bootlink = 2
#no_bootstrap
```

The **Server_windows** option causes the I/O server to provide a **/window** server, so this does not have to be run on the root processor. The error logger is configured to send all its output to a file and to the I/O processor's screen. The default processor names `/00` and `/IO` are used, and the I/O processor is connected to link 0 of the root processor. The I/O server must boot up the root processor rather than attempt to connect into a running network. The first **initrc** file might be something like:

[9] Trademark of Digital Equipment Corporation

```
#
# Run a Session Manager
run -e /helios/bin/startns startns -nons
# Create a console window
console /window console
# And start a session
run -e /helios/bin/newuser newuser mary
```

A Session Manager is started up, using the I/O server's error logger for its diagnostics output. Then a window is created, and a user session is started up. The same **nsrc** file can be used as with the standalone workstation. As the network is expanded the **initrc** file can be changed to allow for it. For example, when a graphics display is added the following lines could be added before creating the console window.[10]

```
ifabsent /window run -e /helios/bin/xhelios xhelios
ifabsent /window run -e /helios/lib/window window
```

If the **Server_windows** option in the **host.con** file is enabled a **/window** server will already exist, so there is no need to run X. If the option is disabled then the above lines would start up the X server and the terminal emulator. At times it may be useful not to run X, if an application needs a lot of memory, and a single line change to the **host.con** file achieves this. Please note that with some I/O processors, notably PCs, it will be necessary to enable the **Xsupport** option of the **host.con** file as well.

When a SCSI interface is added a file server could be started up by adding the command

```
run -e /helios/lib/fs fs scsitram 3
```

indicating that the file server should interact with a SCSI TRAM module on link 3. It might be necessary to run the **tr_reset** command first to reset this TRAM module. When an ethernet interface is added the **tcpip** and **inetd** commands could be added to the **initrc** file. The user is unlikely to add serial ports for the mouse and keyboard, since these can usually be provided by the I/O processor.

The **nsrc** file for this configuration is the same as for a stand-alone workstation. The presence or absence of an I/O processor has no significant effect on the configuration of the Session Manager, which is the only part of the networking software that is running.

Given that there is an I/O processor, there are two possible things to do when a user logs out. The first is to put up another login prompt. In this case logging in as `shutdown` would cause a terminate message to be sent to the I/O server, as well as synchronising the hard disc and disconnecting the ethernet. The second is to shut down the system as soon as the user logs out, including terminating the I/O server, which means that the code in `shutdown`'s **.login** file is moved to the user's **.logout** file. Exactly the same work must be done to shut down the system.

```
termfs /fs
stopio /IO
```

[10] In practice adding a graphics display would require a second processor rather than plugging more hardware into the root processor

2.7.3 Workstation for developing parallel software

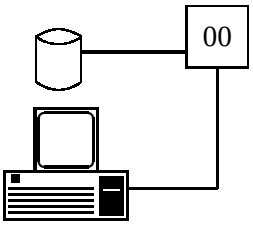


Figure **2.34** A workstation for developing parallel software

Usually there is no point in running all the networking software on a single-processor system. The Session Manager is required to get a user session started, but the network server and Task Force Manager are redundant. However, with some parallel programming systems including the Helios CDL it is possible to develop software on a single processor and run it unchanged on multiple processors. To test the software it is necessary to start up a network server and run a Task Force Manager. The **initrc** line used to start the networking software should be changed to:

```
run -e /helios/bin/startns startns -r /helios/etc/default.map
```

This will cause **startns** to run a network server as well as a Session Manager. The **-r** options is used to inform the network server that it is running in a network that has not yet been booted. Strictly speaking it is redundant in this single-processor system because there is nothing else to boot, but as soon as another Transputer is added it would be essential. The resource map is held in the file **/helios/etc/default.map**. The text form of this might look like the following:

```
Network /Cluster {
    processor 00 { ~IO, , , ; run -e /helios/lib/fs fs scsitram 1 }
    processor IO { ~00; IO }
}
```

If desired the T222 on the SCSI TRAM module could be specified in the resource map as a native processor, but there is little point in doing so. It is possible to use names other than `00` and `IO`, provided the **host.con** names match the ones in the resource map.

```
root_processor = /tom
io_processor   = /pc

Network /Cluster {
    processor tom { ~pc, , , ; run -e /helios/lib/fs fs scsitram 1}
    processor pc  { ~tom; IO }
}
```

There is no need to specify a reset driver or a configuration driver, since there are no other processors to boot up. It will be necessary to change one line in the **nsrc** file. Because a Task Force Manager is required to test the parallel software, the option

no_taskforce_manager should be disabled. This will stop the Session Manager simply running a shell on the root processor, as happens in the previous two networks. The network is still a single-user system, with no need for password checking. The time taken for the network server to initialise the two processors is very small, so there is no need to wait for the network before the Session Manager is run and sessions can be started.

The options for shutting down the system are the same as before. It is necessary to synchronise and terminate the file server, if it is running in the network, and to send a terminate message to the I/O processor. This can be done either in a user's **.logout** file or in the **.login** file for user `shutdown`.

2.7.4 A small network

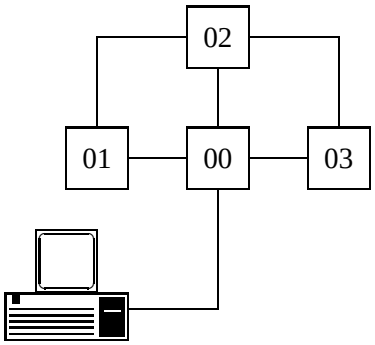


Figure **2.35** A small network

As with the previous two networks, the I/O processor will boot up the root processor and the **init** program will start running. Then the remaining three processors should be booted. There are two ways to do this.

1. Do the booting 'by hand.'

2. Run the networking software.

The network can be booted manually from the **initrc** file if desired. The three processors may or may not have to be reset, depending on whether or not resetting the root processor from the I/O processor acts as a global reset. If no reset is required the following commands can be used. [11]

```
run -e /helios/lib/rboot rboot 1 /00 /01 0032
run -e /helios/lib/rboot rboot 2 /00 /02 2203
run -e /helios/lib/rboot rboot 3 /00 /03 0220
```

This will automatically enable the cross links. Alternatively the **clink** command could be used.

```
run -e /helios/bin/remote remote 01 clink 2 -p
run -e /helios/bin/remote remote 02 clink 1 -e
```

[11] See *The Helios Encyclopaedia* for a fuller explanation of manual booting using **rboot**.

The **clink** command has to be run on the processor with the link that has to be changed. Hence the **remote** command is required. If not all processors are attached to the root processor it is still possible to perform a manual bootstrap, again by using the **remote** command.

```
run -e /helios/lib/rboot rboot 1 /00 /01 0002
waitfor /01
run -e /helios/bin/remote remote 01 rboot 2 /01 /02 3200
waitfor /02
run -e /helios/bin/remote remote 02 rboot 3 /02 /03 0320
run -e /helios/lib/clink clink 2 -e
run -e /helios/lib/clink clink 3 -e
```

Note that the root processor is booted up with all but one of its links on a 'not connected' setting, the exception being the link to the I/O processor. Hence it is necessary to enable the cross links from processor `00` to `02` and `03`, after these have been booted.

If the processors are not automatically reset, more work must be done. If the TRAM reset scheme is in use the **tr_reset** program should be run before attempting the bootstrap of the other three processors.

```
run -e /helios/lib/tr_reset tr_reset
run -e /helios/lib/rboot rboot 1 /00 /01 0032
 ...
```

If the Parsytec scheme is in use the **pa_reset** program can be used. It is desirable to use the **pa_rboot** program instead of **rboot**, since the former is specifically designed for booting Parsytec hardware.

```
run -e /helios/lib/pa_reset 1
run -e /helios/lib/pa_reset 2
run -e /helios/lib/pa_reset 3
run -e /helios/lib/pa_rboot 1 /00 /01 0032
run -e /helios/lib/pa_rboot 2 /00 /02 2203
run -e /helios/lib/pa_rboot 3 /00 /03 0220
```

Even when booting by hand it is still necessary to run a Session Manager to create a user session. This can be done using **startns** and the **-nons** option, as before. In the **nsrc** file the **no_taskforce_manager** option should be enabled, because there is no network server and hence the Task Force Manager cannot obtain a domain of processors. This prevents the user from running parallel software automatically. However, the **remote** program can be used to run programs explicitly on specific processors.

For booting by hand, the **nsrc** file and the various ways of shutting down the system are the same. The options need to be changed only if it is intended to run a network server. If booting by hand is considered too difficult, it is possible to run a network server instead. It may or may not be desirable to force this network server to exit.

2.7.5 A fairly small single-user network

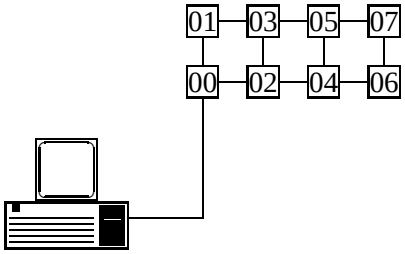


Figure **2.36** A fairly small single-user network

Adding more processors actually makes it easier to decide how to configure the system. For a network this size booting by hand, although still possible, becomes excessively tedious. Hence a network server must be used to boot up the network. The **initrc** file for such a network might be:

```
#
# Run the network server and Session Manager
run -e /helios/bin/startns startns -r /helios/etc/default.map
#
# Run X windows if necessary
ifabsent /window run -e /helios/bin/xhelios xhelios
ifabsent /window run -e /helios/lib/window window
ifabsent /window waitfor /window
#
# Create a console
console /window console
#
# And start a user session
run -e /helios/bin/newuser newuser
```

Both the network server and the Session Manager are started up, with their diagnostic output going to the error logger of the I/O server. If the I/O server does not contain a **/window** server then the X window system is booted up and the terminal emulator is started. A console window is created, and a session is run within that window. The **newuser** command is not given a user name, so the Session Manager will prompt for one. The network resource map for this might look something like this.

```
Network /Mynet {
    Reset { driver; ; pa_ra.d }

    processor 00 { ~IO,    , ~01, ~02; run /helios/lib/lock }
    processor 01 { ~00,    , ~03,    ; }
    processor 02 {    , ~00, ~03, ~04; }
    processor 03 { ~02, ~01,    , ~05; }
    processor 04 {    , ~02, ~05, ~06; }
    processor 05 { ~04, ~03,    , ~07; }
    processor 06 {    , ~04, ~07,    ; }
    processor 07 { ~06, ~05,    ,    ; }

    processor IO { ~00; IO }
}
```

The resource map is quite straightforward. The network is assumed to be homogeneous, using the Parsytec reset scheme. If it contained hardware supplied by different manufacturers using different reset schemes, giving a mixed network, the resource map would have to be more complicated. This was discussed in detail in section 2.6. The only 'complication' in this resource map is running a lock server on the root processor. This program is a simple Helios server not requiring an environment, so the **run** command is not given the **-e** option and no arguments can be passed. An alternative resource map would be:

```
Network /MyNet {
    Reset { driver; ; tram_ra.d }

    processor tom   { ~pc, , ~lisa, ~dick; run /helios/lib/lock }
    processor lisa  { ~tom, , , ~sarah; }
    processor dick  { , ~tom, ~sarah, ~harry; }
    processor sarah { ~dick, ~lisa, , ~susan; }
    processor harry { , ~dick, ~susan, ~fred; }
    processor susan { ~harry, ~sarah, , ~emma; }
    processor fred  { , ~harry, ~emma, , ; }
    processor emma  { ~fred, ~susan, , ; }

    processor pc { ~tom; IO }
}
```

With this resource map the **host.con** file must contain the following lines:

```
root_processor = /tom
io_processor   = /pc
```

or the network will fail to boot up. The **nsrc** file for this network might be something like the following.

```
single_user
#password_checking
#processor_protection
#no_taskforce_manager
share_root_processor
#root_processor = /tom
#waitfor_network
#preload_netagent
```

The network is put into single-user mode with a shared root processor. No passwords are required, possibly because the network does not have a separate hard disc for the Helios filing system and hence the password file cannot be protected in any case. Processors are not protected since this option is useful only in a multi-user environment. The network is still quite small, so there is little need for preloading the network agent or to delay sessions until the network is fully initialised.

A Task Force Manager is needed for running most parallel software, including Helios CDL. However, if the user can make do with the facilities provided by the **remote** and **wsh** commands, to run programs on particular processors, then the option **no_taskforce_manager** could be enabled. This would make the Session Manager start

a shell on the root processor, rather than start a Task Force Manager `/mary` for user Mary, and run the shell in `/mary/tfm` as a simple task force.

The **share_root_processor** option may be important if a Task Force Manager is started, otherwise the option is ignored. When the Session Manager creates a new session it needs to obtain one processor from the system pool for running that session's Task Force Manager. In a multi-user network the root processor is always reserved for use by the system, so another processor will be allocated. However in a single-user network it may or may not be desirable to allow the root processor to be allocated. If the **share_root_processor** option is enabled then the root processor will be allocated, otherwise it is reserved for use by the network server and Session Manager. Unless the root processor is low on memory, for example because the network server has to administer a very large network or because the X server or the filing system is running there, it is usual to enable this option. Assuming that a Task Force Manager is run, the user could pre-allocate all processors to that user's domain. Typically this is done in the **.login** file.

```
domain get /00 /01 /02 /03 /04 /05 /06 /07
```

This allocates all processors to the user's domain. The processors will be returned to the system pool automatically when the user logs out and the Task Force Manager terminates. Once the network grows past a certain size specifying all the processors in the **domain get** command becomes tedious, and it may be easier to use a template.

```
domain get 8
```

This would request eight processors with no restrictions on the processors, and the network happens to have exactly eight processors. Shutting down the network happens in much the same way as before. This can happen either in a user's **.logout** file or in the **.login** file for user `shutdown`.

2.7.6 A network with configuration hardware

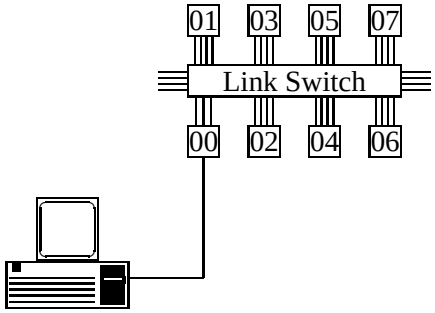


Figure **2.37** A network with configuration hardware

Adding a link switch to the hardware makes very little difference to the network configuration. All of the previous section is still relevant, and one line should be added to the network resource map to specify a device driver for controlling the link switch. Only the network server needs to know about the presence or absence of a link switch, because it is responsible for programming the switch. All other software can ignore it.

```
Network /Cluster {
    Reset { driver; ; telmat_r.d }
    Configure { driver; ; telmat_c.d }

    processor 00 { ~IO,    , ~01, ~02; run /helios/lib/lock }
    processor 01 { ~00,    , ~03,    ; }
    processor 02 {    , ~00, ~03, ~04; }
    processor 03 { ~02, ~01,    , ~05; }
    processor 04 {    , ~02, ~05, ~06; }
    processor 05 { ~04, ~03,    , ~07; }
    processor 06 {    , ~04, ~07,    ; }
    processor 07 { ~06, ~05,    ,    ; }

    processor IO { ~00; IO }
}
```

2.7.7 A single-user supercomputer

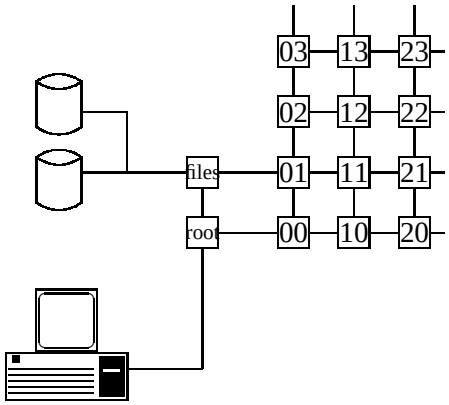


Figure **2.38** A single-user supercomputer

Adding large numbers of processors to the network does not affect the **initrc** file in any way. A typical resource map for this network would be:

```
Network /net {
    Reset { driver; ; pa_ra.d}
    Configure { driver; ; pa_c.d}

    processor root  { ~IO, , ~files, ~00; System }
    processor files { ~root, , , ~01; System;
                      run -e /helios/lib/fs fs MSC 2 }
    processor 00 { , ~root, ~01, ~10; }
    processor 01 { ~00, ~files, ~02, ~11; }

      ...       ...     ...

    processor IO { ~root; IO }
}
```

The first two processors are given the mode **System** instead of the default mode **Helios**. This means that the processor cannot be allocated to the user's domain, and hence the

user's Task Force Manager will not run programs there. In an unprotected network, a single-user system does not require protection of processors. The user could still explicitly run programs on these processors with the **remote** and **wsh** commands. With very large networks the network server will need a considerable amount of memory, and hence there may not be much left on the root processor. The second processor is only used to run the filing system. Treating it as a system processor means that the file server can use all available memory as a cache, and cannot be crashed, or its cache corrupted by a user program.

Several of the options in the **nsrc** file are affected. First, the user will almost certainly want a Task Force Manager to distribute programs, so the **no_taskforce_manager** option must be disabled. Second, the **share_root_processor** option will be ignored because the root processor is reserved for use by the system. This means that the network server will not allocate the processor to any user, not even for running a Task Force Manager. For large networks it is highly desirable to pre-load the network agent, to reduce disc accesses. Also it will take time to boot up a very large network, so it is desirable to wait for the network. The **nsrc** file should look something like this.

```
single_user
#password_checking
#processor_protection
#no_taskforce_manager
#share_root_processor
#root_processor = /tom
waitfor_network
preload_netagent
```

In the **.login** file it is still desirable to obtain all processors in the network. A suitable command might be:

```
domain get 64
```

which would get any 64 processors. Shutting down the network will be the same as before.

2.7.8 Several single-user systems

Given a large array of processors, users can be allocated their own smaller networks, without overlap. This is a safe way of administering the system because users do not interfere with each other's networks. However, it can be an inefficient use of resources. There will be an underlying administrative system, usually not controlled by Helios, to allocate processors to users' networks. Management of this underlying system is hardware-dependent.

Configuring such a system involves separate sets of configuration files, each similar to one of the previous single-user networks. For example, the user in the bottom left of the diagram could have the following resource map.

```
Network /Net {
        Reset { driver; ; rte_ra.d }
        Configure { driver; ; rte_c.d }
```

```
        processor 00 { ~IO, , ~01, ~02; }
            ...    ...
}
```

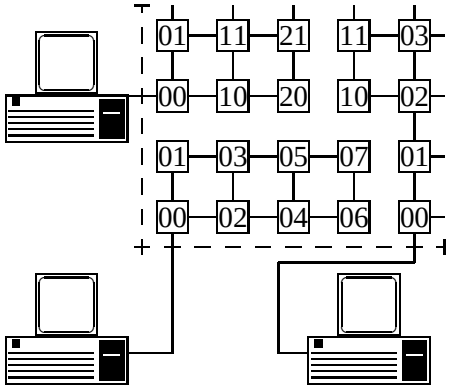


Figure **2.39** Several single-user systems

2.7.9 A process control system

The networking requirements of a process control system are very different from any of the previous networks. It is necessary to run a network server to boot up the network, and possibly to run various control programs on the different processors. The network server should continue running, monitoring the network, rebooting crashed processors, and running the control programs again on these rebooted processors. A typical **initrc** file might look like this.

```
#
# Run the rs232 server, and a terminal server to talk to it
run -e /helios/lib/rs232 rs232 com1.8250.10000000
run -e /helios/lib/terminal terminal /rs232/com1
waitfor /terminal
#
# Create a console window on that terminal for network diagnostics
console /terminal NetDiagnostics
#
# Run the network server only
run -e /helios/bin/startns -r -nosm /helios/etc/floor1c.map
#
# And run a monitor program in another window
console /terminal Monitor
run -e /helios/process/monitor monitor floor1c
```

To get console output a terminal server is started up, interacting with a serial line. The network server is run in one window, and a process monitor program is run in another. There is no need to run a Session Manager, since no user session is required. The monitor program may be an output only device, or it may allow interaction through a keyboard or quite possibly another input device. A resource map for this network might look something like this.

```
Network /floor1c {
    Reset { driver; ; pa_ra.d }

    processor 00 { ext, ext, ~01, ~03; }
    processor 01 { ~00, ~02, ~04, ~03;
         run -e /helios/process/arm.Mk4 arm.Mk4 job72 }
    processor 02 { ~03, ~01,    , ~04;
         run -e /helios/process/TempGauge TempGauge mon12 }
    processor 04 { ~01, ~02,    , ~03;
         run -e /helios/process/arm.Mk4 arm.Mk4 job89 }
    processor 03 { ~00, ~01, ~02, ~04;
         run -e /helios/process/PressGauge PressGauge mon43 }
}
```

Two of the root processor's links are declared as external ones, giving the option of having a larger network interacting with this small one to give remote monitoring and control facilities. The Network is made to run one program on every processor, representing the various jobs to be done by this network. In fact the network server could be made to start several jobs on every processor, simply by giving more **run** commands. Most of the **nsrc** options are redundant, because there are never any users in this network. A suitable **nsrc** file might be:

```
#single_user
#password_checking
#processor_protection
#no_taskforce_manager
#share_root_processor
#root_processor = /tom
#waitfor_network
preload_netagent
```

With no user sessions shutting down, the network must be arranged differently. It is no longer possible to put suitable commands into a **.logout** file or in the **.login** file for user `shutdown`, because these files are never used. Instead there must be an alternative way to shut down the network, for example through the monitor program. In a typical factory environment it is usually necessary to consider very carefully the exact order in which to shut down the network and hence the machinery, to avoid accidents.

2.7.10 A small multi-user network

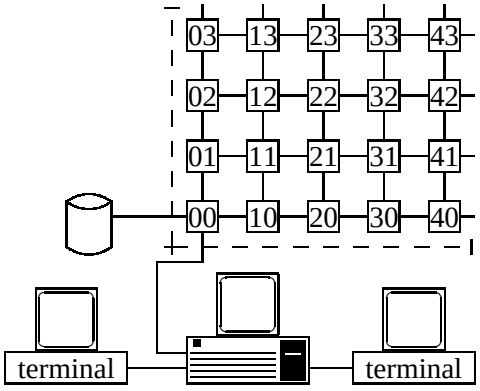


Figure **2.40** A small multi-user network

Configuring a small multi-user network involves several changes to the initial bootstrap. Usually the Helios file server must be started up as soon as possible so that the networking software can use a protected file system. This can be done by using a special Nucleus which incorporates a file server, so that the file server runs as soon as the first processor is booted. It may be desirable to suppress the **/helios** server that normally runs inside the I/O server, using **host.con** options in the I/O server. Chapter 8, *The I/O server*, should be consulted for more details. Suitable entries might include:

```
system_image = ~/lib/nucleus.fs
no_helios
```

The first command in the **initrc** file will usually start up the networking software.

```
run -e /fs/bin/startns startns -r
```

Both a network server and a Session Manager are required. While the network is being booted, it is possible to start terminal servers to cope with the two dumb terminals.

```
run -e /helios/lib/terminal terminal Term1 rs232 /IO/rs232/com1
run -e /helios/lib/terminal terminal Term2 rs232 /IO/rs232/com2
```

It is now necessary to wait for the Session Manager to be ready. Once that happens it is possible to create and register suitable windows.

```
waitfor /sm
console /Term1 User
run -e /helios/bin/newuser newuser
console /Term2 User
run -e /helios/bin/newuser newuser
```

If desired it is also possible to run a session on the I/O processor, but this is not necessarily safe. In particular, if the I/O processor crashes or is rebooted for some reason, the users logged in through the dumb terminals are also affected.

The resource map for this multi-user network is slightly different than for the single-user network, because a special Nucleus is running on the first processor. This Nucleus should not be used on any of the other processors, because these do not have the required hardware to run the file server.

```
Network /Net {
        processor 00 { ... ; nucleus /helios/lib/nucleus.fs }
        processor 01 { ... }
}
```

Since the system has changed from single-user to multi-user, the network configuration file **nsrc** needs important changes. A suitable **nsrc** file might look like this:

```
#single_user
#password_checking
processor_protection
#no_taskforce_manager
#share_root_processor
#root_processor = /tom
waitfor_network
preload_netagent
```

The network is no longer single-user and this affects the configuration. The **single_user** option must be disabled, or the Session Manager will refuse to start more than one session. Password checking is still optional. Processor protection is now desirable, to stop users accessing each other's processors. Use of the **no_taskforce_manager** option is conceivable but unlikely: this would cause both users' shells to run on the root processor, which is dangerous, and there would be no easy way to exploit the network facilities. Sharing the root processor is no longer possible: in a multi-user network the network server and Session Manager run on a reserved processor, inaccessible to users. It is desirable to wait for the whole network to be booted before starting sessions, since the dumb terminals may have no obvious way to work out when the network has been booted. Depending on the size of the network, it may be desirable to pre-load the network agent. Individual users may wish to pre-allocate some processors when they login, by using the **domain** command in their **.login** file.

```
domain get 2
```

Shutting down the network should involve logging in as user `shutdown`, so that all users can log out first. The terminals connected to Helios may be in different rooms, so it may not be easy to work out who is logged in and where. A **.login** file for user `shutdown` might be:

```
wall << end
The system is going down in five minutes.
end
sleep 240
wall << end
One more minute until the system goes down.
end
sleep 60
wall << end
The system is now going down !
end
sleep 5
termfs /fs
stopio /PC
```

2.7.11 Two connected single-user networks

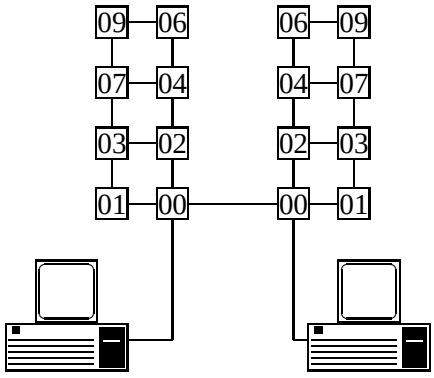


Figure **2.41** Two connected single-user networks

It is possible to connect together two or more networks merely to allow an exchange of data, rather than to share processors which is rather more difficult. The two networks should be configured separately as either single-user or multi-user networks, similar to the ones described previously. The only difference with these previous networks is in the resource map: the connecting link should be declared as external.

```
Network /jonNet {
    Reset { driver; ; tram_ra.d }

    processor 00 { ~IO, ~01, ~02, ext; }
      ...       ...
}

Network /maryNet {
    Reset { driver; ; pa_ra.d }

    processor 00 { ~IO, ext, ~02, ~01; }
      ...       ...
}
```

Given this resource map user Jon could enable the connecting link with the command

```
elink /00 3
```

if there is a network server running in that network, or with the command

```
remote 00 clink 3 -e
```

User Mary could enable the connecting link in much the same way. The link could be disabled again with the command.

```
dlink /00 3
```

It is very important that the two networks have different names. If both networks are called `/Cluster` network names would become ambiguous: there would be two processors called `/Cluster/00`, and so on. Hence in the resource maps given above the two networks are called `/jonNet` and `/maryNet`. Given a connected network the users will be able to access each other's resources subject to any protection that may

be installed. For example, user Jon could access the file `/maryNet/IO/c/test.c`, a file on the remote hard disc. Whether or not it is possible to execute programs in the remote network depends on the **processor_protection** option of the **nsrc** files. If processor protection is disabled then user Jon could execute a command remotely using:

```
remote /maryNet/02 ls
```

However, the networking software will never place a program in a remote network automatically. For many networks, allowing this remote execution facility is desirable, because it gives the users greater flexibility. When shutting down a network it may be desirable to include another command to disable the connecting link. For example, the **.login** file for user `shutdown` might now look something like this:

```
dlink /00 3
stopio
```

2.7.12 A large multi-user network

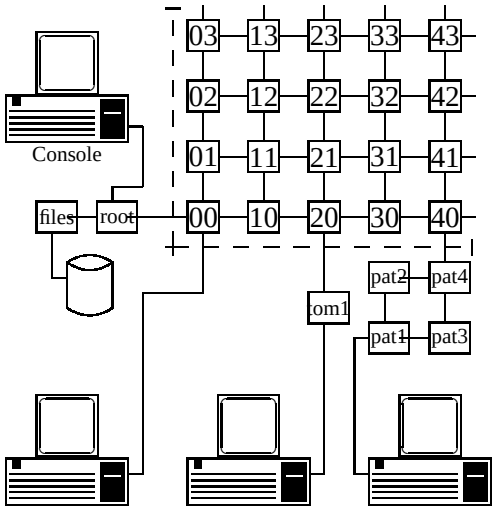


Figure **2.42** A large multi-user network

Building a large, reliable multi-user system usually requires a reliable backbone of processors, with its own system console. Several I/O processors or workstations are connected to this backbone. The network backbone is booted up and should not be rebooted during normal operation. It may have to be taken down occasionally for essential system maintenance or to add new system software or new hardware support. In extreme cases the network may have reached a state where the networking software cannot recover and the whole network has to be rebooted, for example when a worm program has flooded the network and crashed the network server.

There are various different sets of configuration files for such a network. First, there is a set for the network backbone. Then there are separate sets for every I/O processor and workstation connected to this backbone. The backbone's **initrc** file is fairly simple.

```
#
# Run the network server and Session Manager
run -e /helios/bin/startns startns -r /helios/etc/backbone.map
# Create a console window
console /window console
# And start a session
run -e /helios/bin/newuser newuser operator
```

The networking software is started up from the console, probably an I/O processor, and diagnostic output is sent to the I/O server's error logger. It is assumed that the **host.con** file has the following entries.

```
Server_windows
logging_destination = both
root_processor      = /root
io_processor        = /console
bootlink            = 2
```

Multiple windows are enabled inside the I/O processor, because the console does not need any fancy graphics output. Diagnostic output sent to the error logger will be recorded in a file and displayed on the I/O processor's screen. The root and I/O processors are given appropriate names. According to the diagram, link 2 of the root processor is connected to the I/O processor, but this is mainly for artistic reasons. Once the networking software has been started a session is created for the user **operator**. This is a normal shell session, but should be used for system maintenance rather than for running applications.

There is no particular reason why the system console should be a standard I/O processor. It could be a standalone workstation with a graphics display, booting from ROM. Alternatively it might a Transputer with a serial port, and a dumb terminal attached to this port. For these two cases the root processor will need its own hard disc, and the filing system should be started during the ROM bootstrap. Part of the resource map for this backbone would look something like this:

```
Network /Network {
    Reset { driver; ; telmat_r.d }
    Configure { driver; ; telmat_c.d }

    processor root { , ~files, ~console, ~00; System }
    processor files { , , ,~root; System;
              run -e /helios/lib/fs fs scsi }
    processor console { ~root; IO }

    processor 00 { ~pc1, ~root, ~01, ~10; }
        ...
    processor pc1 { ~00; IO }
        ...
    processor 20 { ext[3], ~10, ~21, ~30; }
        ...
    processor 40 { ext[4], ~30, ~41, ~50; }
        ...
}
```

Typically the backbone would consist of a Parsytec SuperCluster, a Telmat T.Node, a Meiko Computing Surface, or a mixture of these. The resource map has to specify the appropriate drivers. The root processor and the filing system processor run as **System** processors, so that they will not be allocated to users. Processor `00` is shown as connected to another I/O processor, `pc1`. When an I/O server starts up on that I/O processor it should enable the connecting link. Usually the Nucleus on `00` will detect this and send a message to the network server. When an I/O processor connects to a network the network server will automatically locate a **/window** server in that processor and start a new session. Processors `20` and `40` are listed with external links, indicating that at a future stage there may be a processor or a network at the other end of the link. The **nsrc** file for the backbone would be something like this:

```
#single_user
#password_checking
processor_protection
#no_taskforce_manager
#share_root_processor
#root_processor = /tom
waitfor_network
preload_netagent
```

This is the same **nsrc** file as for other multi-user networks. The size of the network has little or no effect on the **nsrc**, only on the resource map. Typically the **.login** file for the operator would start up one or more monitoring programs, in different windows. Also, there would be one or more interactive shells for system maintenance. The **.logout** file could contain commands to shut down the network, since running a network without an operator may not be a good idea. This could replace the work normally done in the **.login** file for user `shutdown`.

In addition to the network backbone the diagram shows three ways of connecting into the network. On the left is a single I/O processor with just a link adapter, no Transputer. When the I/O server runs on that processor it should not attempt to boot up a Transputer. Instead it should enable the link into the network. To achieve this, the **enable_link** option should be enabled in the **host.con** file. Some time later, in about a second or two, the network server detects this, locates a **/window** server inside the I/O processor, creates a window, and creates a new session within that window. The Session Manager prompts for a login name and password. When the user logs out another prompt is displayed. If the I/O server terminates the network server detects this and takes appropriate action, stopping the Session Manager from running **login** inside the I/O server's window, and possibly aborting a session that might still be running from inside that I/O server.

In the middle is an I/O processor with a single processor. Alternatively it could be a standalone workstation with a hard disc and graphics display. This processor is booted up normally, by the I/O server or from ROM. On the right is a small network of processors. Both machines start up in much the same way. The **initrc** file might look like this.

```
#
# Initrc file for connecting into a larger network
#
# Run X windows if necessary
```

```
ifabsent /window run -e /helios/bin/xhelios xhelios
ifabsent /window run -e /helios/lib/window window
ifabsent /window waitfor /window
console /window console
#
# Start a network server, but no Session Manager. Wait for the
# network server to perform its initialisation
run -e -w /helios/bin/startns startns -r /helios/etc/outside.map
#
# Now join the larger network
run -e -w /helios/bin/joinnet joinnet tom1 2
#
# And register the window with the backbone's network server, to
# start a session
run -e /helios/bin/newuser newuser tom
```

The resource maps used to boot the two small networks are fairly standard, although care has to be taken with the names used. The resource map for Tom's network might be:

```
Network /TomNet {
        Processor tom1 { ~TomPC, , ext, ; }
        Processor TomPC { ~tom1; IO }
}
```

Pat's resource map might be something like this:

```
Network /PatNet {
        Processor pat1 { ~PatPC, , ~pat2, ~pat3; }
        Processor pat2 { ~pat1, , , ~pat4; }
        Processor pat3 { , ~pat1, ~pat4, ; }
        Processor pat4 { ~pat3, ~pat2, , ; }
        Processor PatPC { ~pat1; IO }
}
```

The **nsrc** file for both networks might look like this.

```
#single_user
#password_checking
processor_protection
#no_taskforce_manager
#share_root_processor
#root_processor = /tom
waitfor_network
preload_netagent
```

Since no Session Manager is run inside the small network, options like **single_user** and **password_checking** are ignored. These options are useful only for the Session Manager, and the only Session Manager in the network runs in the main backbone. Once the network server has booted up and initialised the small external network it is necessary to connect into the main network. This is the purpose of the **joinnet** command, which takes two arguments, describing the processor and link connected to the backbone. It contacts the local network server, then it tries to enable the link to the network backbone, and searches the backbone for another network server. The local

network server is then made to surrender control of its processors to the remote one, and will exit. The main network server in the backbone now knows about the external processors, and can allocate these to users logged in through the external network. For safety reasons these processors will not be allocated to other users. Once the external network has been joined with the main network it is possible to register windows and start sessions with the **newuser** command.

2.7.13 A mainframe computer

There is no essential difference between a 'mainframe' computer and the large multi-user network of the previous section. Once a multi-user network reaches a certain size and has a sufficient amount of attached I/O hardware, thinking of it as a mainframe rather than an ordinary processor network gives the right frame of mind for administering it. For example, a traditional mainframe requires one or more full-time or part-time operators with responsibility for the day to day running of the machine, including making tape backups. Also, a traditional mainframe requires a room of its own with suitable air conditioning. A large processor network will generate considerable heat, just like a mainframe, so a separate room may be appropriate.

The only difference between configuring this mainframe and the multi-user network of the previous section is the amount of software to be started up to handle the varied hardware. The mainframe has multiple discs and a tape drive, rather than a single disc drive. It has dumb terminals attached to serial ports, so the networking software must start up **/rs232** servers and **/terminal** servers. There is ethernet hardware, so run low-level software to interact with this hardware and higher-level daemons, to allow for file transfer, remote logins, and so on. Usually this software can be started up conveniently with **run** commands in the resource map.

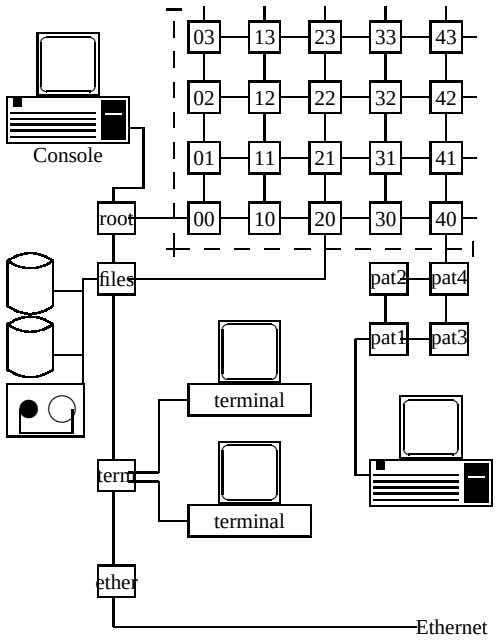


Figure **2.43** A mainframe computer

2.7.14 Networked mainframe computers

To network mainframes together, simply specify the connecting links as external links in the resource map, and enable them automatically at the end of the **initrc** file. Mainframes are normally networked together on a permanent basis, rather than enabled and disabled when required, because the networking software will not know when a user needs access to a remote facility. When shutting down a machine it is polite to disable the links, but not essential.

Chapter 3

Programming under Helios

The purpose of this chapter is to provide a description of the mechanics of programming under Helios. It makes no attempt to teach programming itself, and the reader is assumed to be familiar with concepts such as stack, calling conventions, program modules, and the like. Instead this chapter describes how existing programs can be compiled under Helios to produce executables.

This chapter concentrates on programs written in the C language, because this is the language used for most Helios applications. Most of the chapter should be applicable to other languages such as Fortran, Pascal, and Modula 2. Since these languages are not part of the standard Helios package the language specific documentation should also be consulted.

Section 3.1 gives a basic introduction to the programming tools. Experienced programmers may find it tedious, but the information should suffice for most users.

Section 3.2 gives more detailed information about the programming tools, and in particular it describes the underlying programs. In addition this section describes libraries, what they are for and which ones are available. A distinction is made between Scanned and Resident (Shared) libraries, and an example is given on how to produce a Scanned library. This section also describes some of the other tools available under Helios to help programmers, and a brief description of the actual compilation process.

Section 3.3 gives a description of some of the servers available under Helios, starting with a general description of how to interact with different servers and giving a description of some of the more common ones.

The final section of this chapter, section 3.4, is a tutorial. It explains how to use the Helios protection mechanism to protect your files from other users, and how you can then use it to give other users limited access to your files.

3.1 Simple programming

This section describes the basic tools available under Helios to support programming. First the compiler driver is introduced, and is used to compile some simple programs. For non-trivial programs it is desirable to let the system perform the administrative side of compilation, and the **make** utility is useful for this. Finally there is a summary of the various types of file likely to be encountered during programming, and how they can be compiled.

3.1.1 A simple program

Consider the following C program.

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

int main(int argc, char **argv)
{
  puts("Hello world");
  return(EXIT_SUCCESS);
}
```

Before anything can be done with this program it has to be typed in and written to a file. Usually this involves invoking an editor of some sort. Helios comes with the Micro Emacs editor as standard, and this is described in detail in *The Helios Micro Emacs Guide*.[1] Various other editors are also available under Helios.

When the program has been typed in it must be given a filename; for example, **hello.c**. The first part, **hello**, reflects the purpose of the program. The suffix, **.c**, specifies the type of the file, in this case a C program. This suffix is used by tools such as the compiler driver to work out what to do with the file.

Given a typed-in program, it is necessary to turn it into an executable binary by compiling and linking it. Compiling translates the machine-independent C program into a machine specific intermediate file. Linking means taking this intermediate file, adding some start-up code and various libraries provided by the system, and producing an executable file. Doing this can be tedious, so Helios provides a compiler driver program to do all the complicated bits. To invoke this compiler driver the following command line could be used.

```
c hello.c
```

c is the compiler driver. It is similar to the command **cc** on most Unix systems, and the options are usually identical. Given a single argument ending in a **.c** suffix it assumes that this argument refers to a file containing a C program. This program is passed through the C compiler to produce an intermediate assembler file. This assembler file is then linked with some initialisation code, the C library, and various other libraries needed by the program. Finally a binary executable is written to the file **a.out**, and typing in the command **a.out** to the shell will execute this file and cause the text `Hello world` to appear on the screen. During the compilation process the compiler will display a copyright message, giving amongst other things the version of the compiler, and often the compiler will give some warnings about your program. For the example program the compiler will warn you that the variables **argc** and **argv** are not used. A quick examination of the actual program will show that this is correct, the variables are not used, but for this program it does not matter.

3.1.2 Driver options

There is rather more to the compiler driver than just compiling a “Hello world” program. Some of the more useful command line options are given below.

[1] Published by Distributed Software Ltd.

-help Simply typing in the command **c -help** will cause the compiler driver to list the current options. These options include the type of source files recognised, for example `.c` for C programs and `.f` for Fortran programs. Next there are a number of command line options, with almost every letter of the alphabet used for one option or another, both the upper case and the lower case version of the letter. Finally there is a set of environment strings which can be used to override some of the defaults built into the compiler driver.

-o It is possible to specify a particular file to hold the final binary program. For example, given the command line

```
c -o hello hello.c
```

the compiler driver will compile and link the program **hello.c** as before, but the binary executable will be written to the file **hello** instead of to **a.out**.

-g This option is used to compile the program for debugging. It is useful only if you have a copy of the Helios debugger. Please consult the Helios debugger manual for more information.

-D is used to pre-define some options for the C preprocessor. For example, consider the following command line.

```
c -DTesting -Ddebugflags=17 hello.c
```

This is equivalent to having the following two lines at the start of the C program.

```
#define Testing
#define debugflags 17
```

-I can be used to make the C compiler search a particular directory for the C header files. By default the compiler will search the current directory, followed by the main include directory **/helios/include**. Suppose that there is an **/include** server running somewhere in the network, which can be used to read header files without accessing a disc. To use this include disc, the following command line can be used.

```
c -I/include -o hello hello.c
```

-O is used to enable optimisation within the compiler and the linker. When optimisation is used the final binary program is likely to run faster. In addition the program may be smaller than it would otherwise be, but that is not guaranteed. Producing an optimised binary will take longer, possibly a lot longer, than producing an unoptimised binary.

-T is used to specify the type of processor on which the binary object is meant to run. Options include **-T4** to compile for a T414 Transputer, and **-T8** to compile for a T800 Transputer. The processor types supported are subject to change at any time, but **c -help** should list the currently supported processors.

The above list accounts for less than a quarter of the command line options available in the compiler driver. In addition there are a number of environment variables used to override defaults built into the compiler driver. For example the default name of the binary executable is **a.out**, if the user does not specify another file with the **-o** option. There is an environment variable **OBJNAME** which can be used to change the default from **a.out** to something else. Consider the following two commands:

```
setenv OBJNAME binary
c hello.c
```

This causes the compiler driver to generate the file **binary** as the executable binary program, instead of **a.out**. Typing the name **binary** would execute the program and display the text `Hello world`.

The exact options understood by the compiler driver are subject to change at any time, as Helios development continues and more processors are supported. Also, many of the options are of little or no interest to the majority of users. Hence this chapter does not give a complete list of all options, merely the ones most likely to be needed by a typical user. Instead the **-help** option can be used to determine the options understood by the current version of the driver, or the *Helios Encyclopaedia* can be consulted.

3.1.3 Multiple modules

The "Hello world" example used earlier consisted of just one source file. This is fine for simple programs, but many programs are so large that they should be split into a number of different source files or **modules**. Having multiple modules makes the linking process more complicated and hence slows that down. However, compiling a small file takes less time than compiling a large one. Deciding when and how to split a large program into separate modules should always be left to individual programmers. To show how to use the compiler driver with separate modules, consider the following two files.

main.c

```
#include <stdlib.h>

extern void say_hello(void);

int main(int argc, char **argv)
{
  say_hello();
  return(EXIT_SUCCESS);
}
```

io.c

```
#include <stdio.h>

void say_hello()
{
  puts("Hello world.");
}
```

The simplest way to compile these two modules together is to specify both of them on the command line, for example:

```
c -o hello main.c io.c
```

The compiler driver will put module **main.c** through the compiler, generating two warnings about unused variables as before. Then it will put module **io.c** through the compiler. Finally it will link the resulting object files together to produce the binary executable **hello**. This is simple. A problem occurs when just one of the source files is changed, because using exactly the same command line will cause the compiler driver to put both source files through the compiler again.

To avoid this problem it is necessary to make use of intermediate files. If you examine the directory after the above command you would find two extra files, **main.o** and **io.o**. These are intermediate object files which can be passed to the linker. To rebuild the binary executable using the two object files, the following command line can be used.

```
c -o hello main.o io.o
```

Since all the file names passed as arguments to the compiler driver end with the **.o** suffix, the compiler driver can work out that none of the files need to be compiled and hence it will invoke the linker. Suppose that one of the files needs to be recompiled but the other one does not. A command line to do this is:

```
c -o hello main.c io.o
```

One of the file arguments ends with **.c** and hence the compiler driver will invoke the compiler for this file. The resulting **main.o** object file will be linked with the other object file and various standard libraries to produce the binary executable. This extends naturally to any number of source and object files.

When there are several source files it may be easier to recompile one file to the intermediate object form, and then link all the **.o** files together in a separate command. For example if the programmer changes just one file out of six then it is inefficient to recompile all six. The compiler driver has a **-c** option to produce intermediate object files. For example, the following command lines may be used to rebuild the program from scratch.

```
c -c main.c
c -c io.c
c -o hello main.o io.o
```

3.1.4 Make

Using the compiler driver to build programs is fine for simple programs with just one of a small number of modules. For more complicated programs it becomes tedious, inefficient because the user has to remember to recompile files when appropriate, and dangerous because it is easy to forget one of the files. However, the basic job required is fairly simple: given a set of sources, recompile any that have changed since the last compilation; then link together all the objects that have been changed. All this

administration is tedious, and can be left to the computer. A tool which can be used for this is the **make** utility.

Consider an example. There is a program `teatime`, comprising the sources `assam.c`, `water.c`, `sugar.c`, `cream.c`, `scones.c`, and `jam.c`. In addition there is a header file `teapot.h` which is used by the source files `assam.c` and `water.c`. The **make** utility needs to know what the target program is, what it depends on, and similar information. To do this it reads a separate file, **makefile**. For this application a suitable **makefile** is:

```
teatime: assam.o water.o sugar.o cream.o scones.o jam.o
        c -o teatime assam.o water.o sugar.o cream.o scones.o jam.o

assam.o: assam.c teapot.h
        c -c assam.c
water.o: water.c teapot.h
        c -c water.c
sugar.o : sugar.c
        c -c sugar.c
cream.o : cream.c
        c -c cream.c
scones.o: scones.c
        c -c scones.c
jam.o: jam.c
        c -c jam.c
```

These lines all have the same format. First there is a **target**, indicating something that the **make** program should produce. This target is usually, but not always, a file. Following the target is a colon **:**, and then a list of **dependencies**. These indicate the objects that must exist before the target can be made, and each such dependency is usually listed in the makefile as another target. On the next line, and starting with a tab character, there is a command which the **make** program should execute to build the target. These commands are ordinary commands, as you would type in at the shell prompt or possibly put into a shell script. The first target in the makefile is special, it is the **default** target which will be made unless **make** is specifically instructed to build some other target. The combination of target, dependencies, and commands is usually referred to a **rule**.

Referring to the example makefile, the default target is `teatime`. Before this target can be made the **make** program has to build the targets `assam.o`, `water.o`, and so on. Once all the object files have been made it is possible to build the `teatime` program by invoking the compiler driver, as per the command line. The second target `assam.o` is required to build the default target. This second target depends on two files, `assam.c` and `teapot.h`, and can be built by invoking the compiler driver on the `.c` file.

So how does this work in practice? Assume that the source files exist, that the makefile has been typed in, but that nothing has been compiled yet. To get the job started, just use the **make** command without any arguments. **make** will read the makefile, determine the default target, and note that all the object files need making first. There are rules for making all the object files. Hence **make** would invoke the compiler

driver for all the source files, one by one, and then the compiler driver would be invoked a last time to link the objects together and produce the executable program. If something goes wrong halfway through the make, for example if one of the files could not be compiled because of a typing mistake, then the **make** would be aborted at that point.

Now suppose that the programmer makes a small change to the module `scones.c` and needs to rebuild it. All that the programmer has to do is type **make** again. The program reads the makefile again, works out the default target, and hence the dependencies. The first dependency is on `assam.o`. This object file in turn depends on two other files, `assam.c` and `teapot.h`. However, these two sources have not changed since the last time that `assam.o` was created: **make** examines the file system to work out which files were changed and when, so it can work out such things. Since the sources have not changed there is no need to recompile them. When `scones.o` is examined **make** will discover that the dependency file `scones.c` has changed since `scones.o` was last updated, and hence this file will be recompiled. This will result in a file `scones.o` which has changed since the last time `teatime` was produced, so it is necessary to remake the default target by relinking the objects.

If the programmer changes the file `teapot.h` everything becomes a bit more complicated. Since `teapot.h` is a dependency for both `assam.o` and `water.o`, both these targets will be remade by executing the appropriate commands. Subsequently, the default target, `teatime`, has to be remade by executing the linking command.

Variables

The **make** utility is more powerful than described so far. The first useful extra facility is the use of variables. In the first rule of the makefile there are two identical lists of object files. This is inefficient, because the details must be typed in twice. It is also slightly dangerous, because the two lists might get out of step as the software is being developed. To avoid this problem a variable could be used.

```
objects = assam.o water.o sugar.o cream.o scones.o jam.o

teatime: $(objects)
        c -o teatime $(objects)
```

The first line declares a variable `objects`, and this variable is assigned a text string containing the six object names. Variables are always text strings, as the **make** utility does not have the concept of variable types such as integer, double precision numbers, or anything like that. This first line is not a **make** rule, because it does not follow the syntax for rules: target, colon, dependencies, commands.

The next line is a rule. It defines the default target `teatime` as before, and then uses the objects variable for the dependencies. The syntax `$(x)` means “Insert the value of variable **x**”. This is different from depending on target **x**: if the rule said that `teatime` depended on **x** without the brackets, **make** would assume that it had to create target **x** first; since there is no rule for making target **x**, this would produce an error message. After the dependencies comes the command used to rebuild the current target, and again this can use the objects variable.

In addition to your own variables, **make** pre-defines a number of useful variables for you. These are used mainly to make it easier to write the commands needed to build the targets. All pre-defined variables consist of a `$` character followed by something else.

`$@` is equivalent to the target of the current rule. For example, the first rule in the makefile can be written as follows.

```
teatime: $(objects)
        c -o $@ $(objects)
```

When the **make** program comes to execute the command it will substitute the current target, `teatime`, in place of `$@`, and then it will substitute the user's variable `objects` to give the list of objects files.

`$^` is equivalent to all the dependencies for the current rule. For example, the rule to rebuild the object file `scones.o` can be written as:

```
scones.o: scones.c
         c -c $^
```

In this case there is only a single dependency file, `scones.c`, so the **make** program will substitute this name for `$^`.

`$<` is similar to `$^`, but refers to the first dependency only. This is useful for the file `assam.o`, which has a dependency on a header file as well as the C source file: using `$^` would result in an attempt to compile the header file. A suitable rule for building `assam.o` would be:

```
assam.o: assam.c teapot.h
        c -c $<
```

`$*` stands for the target name without its suffix. For example, if the target of the rule is called `assam.o` then the variable `$*` is equivalent to `assam`. This variable is occasionally useful when manipulating suffixes in a fairly unusual way. Details of such suffixes are given later on. For example, the following rule can be used to build an object file together with its associated assembler file, should this be required for some reason.

```
scones.o: scones.c
         c -S $<
         c -c $*.s
```

The first command is used to compile the C program to give the corresponding assembler file, `scones.s`, without producing the object file. The second line is used to produce the object file. Since the current target is `scones.o` the variable `$*` is set to `scones` and hence the second line invokes the compiler driver with argument `scones.s`.

`$?` is a variable defining which of the current target's dependencies had changed. It is used mainly for reporting during the progress of a make, or when the makefile is not working as expected. For example, consider the following:

```
assam.o: assam.c teapot.h
	echo Rebuilding $@ because $? has changed
	c -c assam.c
```

Complicated lines

Building programs can get very complicated, so the lines in a makefile can become equally complicated. For this reason makefiles can have comments, just like ordinary sources. In a makefile comments are introduced by a hash symbol, just like in shell scripts. Comment lines are ignored completely by the **make** program.

```
#
# Makefile for the subsystem "teatime"
# This is component 16.30 of project "daily schedule"
# Author: A. Programmer
#
```

For large systems it is possible that some text does not fit into one line of the makefile. **make** uses the same approach as the shell, a backslash character \ indicates that the current line really continues on to the next one. For example, the following lists some objects for a more complicated system.

```
high_tea = darjeeling water honey cream lemon cakes biscuits \
           scones jam crumpets silver_spoons china \
           cucumber_sandwiches and lots of other things
```

Note that the last line does not use a backslash character, because there is no point in continuing the line on to the following blank line. In the commands section of a rule, it is possible to give several different commands.

```
assam.o: assam.c teapot.h
	c -S $<
	c -c $*.s
```

The first command compiles the source file called `assam.c` to produce an assembler file `assam.s`. The second command takes this assembler file and turns it into an object file called `assam.o`. Note that there is no backslash character between the commands. If a backslash character were used then **make** would merge the two command lines into a single line, giving `c -S assam.c c -c assam.s`: this will not have the desired effect, instead it will cause the compiler driver to produce an error message and abort.

Default rules

Taking the original makefile of some pages back, but using the facilities described so far, we would get a file like the following.

```
#
# There will be some comments at the start of the file.
#
```

```
# These are the modules required for the teatime program
objects = assam.o water.o sugar.o cream.o scones.o jam.o

# The default target is the program teatime
teatime: $(objects)
        c -o $@ $^

# These rules recompile the various modules needed by teatime
assam.o: assam.c teapot.h
        c -c $<
water.o: water.c teapot.h
        c -c $<
sugar.o: sugar.c
        c -c $^
cream.o: cream.c
        c -c $<
scones.o: scones.c
        c -c $^
jam.o: jam.c
        c -c $<
```

Note that `$<` can often be used instead of `$^`, but not the other way around. Most of the commands in this makefile are identical, and it is silly to duplicate the commands every time. **make** has a mechanism for defining default rules, for example the default way to create an object file from a C source file is to invoke the compiler driver with the **-c** option. More specifically, **make** can be given rules on how to turn a file with one suffix into a file with a different suffix. The first step is to specify which suffixes should be recognised by the **make** program.

```
.SUFFIXES:
.SUFFIXES: .c .f .o
```

Most **make** programs require two lines in the makefile for this. The first line eliminates any suffixes which might be built into the **make** program, because it is not usually clear whether the built-in ones are correct, or even what they are. The second line defines the suffixes used by this makefile. In the example, **make** is told to recognise three suffixes, for C and Fortran sources and for the object files, but any number of suffixes can be given. Default rules involve a special target name constructed from the two suffixes. There should not be a dependency, and the command name should use the various built-in variables where possible. For example, the following defines a default rule for changing a file with a **.c** suffix into one with a **.o** suffix.

```
.c.o:
        c -c $*.c
```

Constructing similar rules for other types of compilation is relatively easy, and a list of such rules is given at the end of this section. To tell **make** to use the default rule to build a target, just specify the target and the dependencies without a command.

```
assam.o: assam.c teapot.h
```

```
water.o: water.c teapot.h
sugar.o: sugar.c
cream.o: cream.c
scones.o: scones.c
jam.o: jam.c
```

In fact makefiles can be even simpler than this. The last four lines are redundant because **make** will automatically use default rules under certain circumstances. Suppose **make** needs to build the target `X.o`. The following conditions must be met:

1. There is no specific rule for building target `X.o`
2. There is a default rule for building `.o` files from `.c` files
3. There is a file `X.C`

If all conditions are met then **make** will automatically use the default rule. The whole makefile now looks something like this.

```
#
# Simplified makefile for building the teatime system
#
.SUFFIXES:
.SUFFIXES: .c .o

.c.o:
        c -c $*.c

objects = assam.o water.o sugar.o cream.o scones.o jam.o

teatime: $(objects)
        c -o $@ $(objects)

assam.o: assam.c teapot.h
water.o: water.c teapot.h
```

It is still necessary to have rules for the two object files `assam.o` and `water.o` because these depend on a separate header file as well as on the source file. If these dependencies were not part of the makefile then the two objects would not be remade if the header file were changed, which was one of the reasons for using **make** in the first place.

Multiple targets

Usually there is only one makefile per directory. If a directory is to contain multiple programs then it must be possible to build multiple targets with a single makefile. For example, suppose a given directory is used to build three programs: `coffee_break`, `lunch`, and `teatime`. Unless told otherwise, **make** treats the first ordinary rule in the makefile as the default target it is supposed to make. This default target need not refer to a real file. For example, the following makefile achieves the required results.

```
default: coffee_break lunch teatime

coffee_break: $(coffee_objects)
      c -o $@ $(coffee_objects)

lunch: $(lunch_objects)
      c -o $@ $(lunch_objects)

teatime: $(teatime_objects)
      c -o $@ $(teatime_objects)
```

The first target in the makefile is now something called `default`. To build this target, **make** must first build three subsidiary targets. Since **make** is not given any rules for making the main target from its three components, it will never generate a file `default`. This is, in fact, exactly the behaviour that is required.

On the **make** command line it is possible to specify exactly which target is supposed to make. For example suppose that the programmer needs to rebuild program `lunch`, but not the other two programs. Using the command line `make lunch` causes **make** to ignore any targets except `lunch`, and whatever subsidiary targets are needed to build that one. It is also possible to have some special targets: consider the following makefile entry.

```
clean:
       rm $(coffee_objects) $(lunch_objects) $(teatime_objects)
```

This should not be the first rule in the makefile, since the default action should not be deleting the intermediate files. Also this rule should not be a subsidiary target of any other rule, so that it does not get invoked by mistake. Instead the rule will be ignored completely unless the user types `make clean`. Similar extra rules are commonly used for backing up, installing software, and other administrative chores.

Arguments to make

The **make** program can take various command line options, as shown below. The *Helios Encyclopaedia* and the on-line help system may give further information if necessary.

-f <*filename*> allows the user to specify a makefile other than the default, which is **makefile** in the current directory. This is used mainly when experimenting with the makefile, by copying the working makefile to a temporary one and changing and using this temporary one. To use this option the following command line can be used.

```
make -f makefile.tmp
```

-i causes **make** to ignore errors produced by the various commands it runs. Normally when a command in the makefile fails, for example when a compiler encounters a serious error, the **make** program detects this and aborts the whole job. The **-i** option would allow **make** to continue in spite of such errors. This can be dangerous, and the option should be used with care.

-n causes **make** to list the commands it would execute to build the target, without actually executing them.

-q is used mainly with shell scripts. The **make** program reads the makefile and checks the default target. If any commands must be executed then **make** would return an error. Otherwise **make** would exit with success. No commands are actually executed, and no output is produced. For example the following shell script checks whether or not a make is required, and if so it generates some messages first. The shell variable **cwd** is used here to display the current directory.

```
make -q
if ($status == 1) then
 echo Make in $cwd, some work is required
 echo Please go and drink a cup of coffee
 make
else
 echo Make: the target is up to date
endif
```

-s runs the **make** program in silent mode. By default the program will display the commands it is because to execute before actually running them. This option prevents this. It is not very useful because commands like the C compiler will generate a considerable amount of output anyway.

-t is used to **touch** a target, rather than build one. The time stamp associated with the default target or the one specified is changed as if the target had just been rebuilt. This is used mainly when debugging makefiles to avoid excessive recompilations.

Any other arguments will be interpreted as the targets which are supposed to be produced instead of the default target. For example, the following command line

```
make -n coffee_break lunch
```

will cause **make** to show the commands it would execute in order to build the targets called `coffee_break` and `lunch`, without actually executing these commands.

Different make programs

Helios comes with two different **make** programs. The first one, **/helios/bin/make**, is a conventional version of the utility which supports the features described so far and nothing else. This version suffices for most programming needs. The second program is a port of GnuMake, a much more powerful utility suitable for very big applications. This second program is held in the file **/helios/local/bin/gmake**.

The facilities provided by **GnuMake** come at a price. The program is almost eight times larger than the simple **make** utility and it needs a lot more memory at run-time. Hence, on machines which have fairly small amounts of memory use of **GnuMake** should be avoided.

3.1.5 Common suffixes

Programs written in C are conventionally given the suffix **.c**. The suffixes used for other languages and for intermediate files are as follows.

.a is used for source files to the assembler macro preprocessor **AMPP**, used mainly for advanced programming such as building device drivers and Resident libraries.

.a is also used for some libraries. For example, the file **/helios/lib/libX11.a** is the main X Window System library. Such libraries always start with **lib** and are used only for linking: they are not passed as sources to the compiler driver. Hence there is little possibility of confusion.

.bcp is used for programs written in the BCPL language.

.c is used for C programs.

.cpp is commonly used for programs written in C++. Other common suffixes for this include **.cxx**, **.c++**, and **.C**.

.d suffixes refer to device drivers, which are special types of program.

.def is the suffix for another type of library.

.f refers to programs written in Fortran.

.h is a C header file. The standard header files can be found in the directory **/helios/include** and its subdirectories.

.i is used for C files which have been passed through the C preprocessor but have not been compiled. This can be achieved with the **-E** option of the compiler driver. It is also used for special binaries that can be embedded in the Nucleus.

.lib is another way of describing libraries. The name **xyz.lib** is equivalent to **libxyz.a**, but the file name is more likely to fit into the naming limits imposed by certain filing systems.

.m is a macro include file used by **AMPP** programs. The standard macro include files can be found in the directory **/helios/include/ampp**.

.mod is used for Modula-2 programs.

.o files are object files produced by the assembler, which can be passed through the linker to give executable programs.

.p files are another type of object file usually generated when building device drivers and Resident libraries.

.pas files are Pascal sources.

.s is the input to the Helios assembler. Such files are rarely written, but are produced by the C compiler and by **AMPP** and then passed through the assembler to produce **.o** files.

The table below shows the most useful commands for compiling programs.

From	To	Command
X.c	X.o	c -c X.c
X.o	X executable	c -o X X.o
X.f	X.o	c -c X.f
X.a	X.o	c -c X.a
X.mod	X.o	c -c X.mod
X.c	X.s	c -S X.c
X.s	X.o	c -c X.s
X.c	X.i	c -E X.c
X.c	X.p	c -m -c -o X.p X.c
X.a	X.p	c -c -oX.p X.c

The corresponding makefile rules are:

```
.suffixes:
.suffixes: .a .c .f .mod .i .s .o .p

.a.c:
        c -S $*.a
.a.o:
        c -c $*.a
.a.p:
        c -c $*.p
.c.s:
        c -S $*.c
.c.o:
        c -c $*.c
.c.p:
        c -m -c -o $*.p $*.c
.f.o:
        c -c $*.f
.mod.o:
        c -c $*.mod
.c.i:
        c -E $*.c
.s.o:
        c -c $*.s
```

3.2 More advanced programming

This section describes three things. Firstly, it describes libraries: what they are for; what types there are; the main ones available under Helios; and how to produce your own. Secondly, it describes some of the other tools available to help programmers. Thirdly, it gives a more detailed description of the compilation process, indicating the work that has to be done by the compiler driver and explaining some of the less

obvious options that are available. This section is aimed at more advanced users who need facilities not described so far.

3.2.1 Libraries

The purpose of libraries is to make programming easier. For example, a typical application program usually needs to read and write some files. If the application programmer had to worry about individual disc blocks or about the exact hardware registers which must be poked to access a particular kind of disc, then very few programs would be written. Instead, the operating system provides some library routines to perform file I/O, allowing the programmer to concentrate on the application. Application programs must be linked with these libraries after the compilation stage, in order to produce the final executable program.

A considerable number of libraries are available for Helios, either as part of the standard product or as optional extras. The following list is not exhaustive – new libraries are added regularly as Helios development continues – but it contains the more common ones. Some of the descriptions refer to specific routines, and further details of these can be found either in the *Helios Encyclopaedia* or in the on-line help system.

1. The C library is used by most C programs. It contains a wide range of routines varying from **fopen()** to access a file to **strtol()** to manipulate strings.

2. The Fortran library is another language library, like the C one. There are also libraries for Pascal, Modula2, BCPL, and so on. Language libraries are usually mutually exclusive, in other words it is not possible to link a program with both the C library and the Fortran library.

3. The Posix library is an implementation of the IEEE standard 1003.1-1988 Standard Operating System Interface for Computer Environments. The standard defines an operating system interface for Unix-style systems. It includes routines like **execve()** to run another program, and **getpwentry()** to check the contents of the system's password file. For a variety of reasons concerning the architecture of the Transputer and other processors without memory management facilities, the Posix library cannot be fully conformant to the standard for these processors. More details can be found in chapter 5, *Compatibility*. The Posix library is at a lower level than the language libraries. In fact most language libraries are built on top of the Posix library, so for example any C program can access Posix library routines automatically.

4. The System library exists at a lower level still. Helios is based on the client-server model: for an application to do anything other than pure computation it must interact with a server; to read a file it must interact with a file server; to create a lock it can interact with a lock server; to display graphics on a suitable display it must interact with a graphics server, usually the one supplied with the X window system. The System library is used to perform standard interactions with the majority of servers. Additional libraries may exist for specific interactions, for example to display graphics. The System library is part of the Helios Nucleus.

5. The Kernel is the lowest level accessible to application programmers. It provides routines which inherently need to interact closely with the hardware. For example the Kernel has routines **Wait()** and **Signal()** to act on semaphores and provide synchronisation between threads. The Kernel also provides the message passing routines used by the higher-level software, particularly the System library, but use of these routines should be avoided by application programmers. The Kernel is part of the Nucleus.

6. The Utility library is also part of the Nucleus, and provides miscellaneous routines needed inside the Nucleus that did not logically belong anywhere else. These routine include **strlen()** and similar string operations, **Fork()** to start a new thread within the current program, and **IOdebug()** for very low-level debugging.

7. The Server library is the final library embedded in the Nucleus. Its purpose is to facilitate the writing of Helios servers, and it is described in more detail in chapter 12, *Writing servers*. In addition to these four libraries the Nucleus contains two programs, the Processor Manager and the Loader. This is illustrated in Figure 3.1.

Figure 3.1: The library hierarchy

8. The X library provides a programmer's interface to the X server. It is complemented by various other libraries for the X toolkit, widgets, Motif[2], and so on. This is the main graphics facility supported by Helios.

9. The PC graphics library provides some basic graphics facilities, using a VGA or similar display on a host PC. It is a cheap alternative to X, but only offers a fraction of the functionality.

10. The Windows 3 library provides an alternative windowing system and graphics library for use with Microsoft Windows version 3.0. The graphics being displayed on the host PC's screen.

11. The BSD[3] compatibility library contains some routines provided by BSD 4.3 Unix systems that are not part of the Posix standard. These routines are provided to assist in porting programs to Helios.

12. Similarly, Helios contains curses and termcap libraries to improve Unix compatibility. Existing Unix systems need to cope with many different terminal types, for example VT100 terminals attached to serial ports or Xterm windows on an X display. All these terminals need different control sequences to clear the screen,

[2]Trademark of Open Software Foundation, Inc.
[3]Berkeley Software Distribution

move the cursor to a specific location, and so on. To achieve hardware independence for application programs Unix provides the curses and termcap libraries. Under Helios these are redundant, since Helios ensures that all terminals accept exactly the same sequences. Nevertheless, these libraries are provided to cope with existing programs that use them.

13. The Debugger library is linked with programs that have been compiled for debugging, provided the Helios debugger is part of your system. This library interacts with the debugging server **tla**. Its routines are never called from user code, as compilers generate the calls automatically.

14. The Fault library is used for interpreting Helios error codes. For communication between clients and servers Helios uses 32-bit integers to encode requests and replies. For example, the integer `0xCA06800C` is an error code generated by the I/O server indicating that a file is missing. The Fault library provides various routines to interpret such numbers, the most important being **Fault()** which takes a number and turns it into a string that contains a description of the error in English.

15. The Floating Point libraries are used to perform certain floating point operations. There are different versions for the different types of processor. For example **fplib.t4** is used for T414, T400, and T425 Transputers which do not have a built-in floating point unit, and hence these libraries must do the arithmetic the hard way.

16. The Resource Management library provides an application programmer interface to the network of processors. It allows programmers to write applications that examine the network, manipulate processors, execute parallel applications, and so on. It is described in more detail in chapter 7, *The Resource Management library*.

Types of library

The usual place for holding libraries is in the directory **/helios/lib**. Examining this directory can be somewhat confusing, because there are rather a lot of files with rather a lot of different suffixes.

The first type of library is the **Resident** or **Shared** library. A Resident library is a separate piece of code which is loaded into memory upon demand. For example, a typical C program is linked with the C library and the Posix library amongst other things. These two libraries could be embedded into the binary of every C program, which means that every C program would have about 50K of code inside it. Since the directory **/helios/bin** contains over 100 such programs the system would use five megabytes just to hold duplicate copies of the C and Posix libraries. This is nonsensical.

Resident libraries provide an alternative. When a program is linked with a Resident library the library code does not get embedded in the binary object. Instead the binary object contains a description of the Resident library, its name and how to use it. When the program is executed the system detects that it uses one or more Resident libraries, and these libraries are loaded into memory. If there are several programs running on

the same processor needing the same Resident library then only one copy of the library will be loaded, and this will be shared by the various programs.

For example, suppose that the user runs a shell on an empty processor using the command **wsh 01**. The shell is linked with the C library and the Posix library, both of which are resident. Hence when the shell is executed the system detects that both of these libraries are now needed, and they will be loaded into memory automatically. The shell can now use these libraries as if they were embedded in the binary object, just like any other piece of code.

Helios Resident libraries have a **.def** file associated with them. For example, the **/helios/lib** directory contains files **clib.def** and **posix.def**. This **.def** file defines the library and contains all the information needed to link with the library. In addition there are files **clib** and **posix**. These are the library objects themselves, in other words they are the pieces of code loaded by the system when needed. Only the library definition files are needed for linking.

There are several **.def** files which do not appear to have a corresponding object file. They are as follows: **Kernel.def**, **syslib.def**, **util.def** and **servlib.def**. These four libraries are part of the Helios Nucleus and hence they are always loaded in memory, so there is no need to have separate object files.

Resident libraries can be very useful, particularly for system programmers. However, building them is rather complicated (a full explanation of how to do so can be found in chapter 16, *Program representation and calling conventions*). Hence Helios also supports a different type of library: the **Scanned** library. When a program is linked with a Scanned library the linker extracts the parts of the library needed by the program, and adds these to the final binary program. The Scanned libraries shipped with Helios include **bsd.lib**, **curses.lib**, and **termcap.lib**. For example, suppose an application program uses the **popen()** routine. This routine is in the BSD compatibility library **bsd.lib**, so the program has to be linked with this library. This library, however, contains over 50K of code of which only a small part is needed for **popen()**. Hence during the linking process most of the library will be discarded as unnecessary, and the final program does not contain all of the code. The code that implements **popen()** itself **is** embedded in the final program, and is not part of a Resident library loaded dynamically.

Consider an example. There is a Helios command **network** which can be used to examine the current state of a network of processors. This program contains the following parts:

```
Program   network
ResRef    Kernel
ResRef    SysLib
ResRef    ServLib
ResRef    Util
ResRef    FpLib
ResRef    Posix
ResRef    Clib
Module    network.c
Module    popen.c
Module    string.c
Module    signal.c
Module    nuprtnet.c
ResRef    RmLib
```

At the start of the binary object there will be a header identifying the program and containing some information needed by the system when the program is being loaded, such as its stack size. The program contains Resident library references or ResRefs for eight libraries, and the system has to ensure that all of these are in memory when the program starts up, loading them off disc if necessary. The libraries themselves are not part of the binary object, only references to them. The next part is the module **network.c**, which forms the main part of the program. This is the code actually written by the programmer, and there may be several such modules. Then there are three modules **popen.c**, **string.c** and **signal.c** which are part of the BSD compatibility library. That is a Scanned library, so the linker extracted the bits it needed and discarded the rest. The module **nuprtnet.c** also comes from a Scanned library, a private one written by the programmer.

Suppose that at some moment in time two programs are being executed on the same processor: **network** and **domain**. Figure 3.2 illustrates the bits of code loaded into memory.

Figure 3.2: Code in memory

There is only one copy of the C library and of all the other Resident libraries, shared by the two programs. There are two copies of module **popen.c** in memory because that module comes from a Scanned library, and hence the module is actually part of the binary program.

Linking with libraries

Given all these libraries it is necessary to know how to link programs with them. By default programs are linked automatically with the libraries they are likely to need. For example, every C program is linked automatically with the C, Posix, floating point, fault, server, utility, system, and Kernel libraries. However, the linker is intelligent enough to only link libraries that are referenced by other code modules into the executable file it produces. So, if a program does not make use of any Server library calls, the binary program will not contain the corresponding Resident library reference. Some libraries, for example the Kernel, are needed by higher-level libraries so these are nearly always included.

Linking with most other libraries is fairly easy. The following command line links a program with both the X and the Resource Management libraries.

```
c -o drawmap -lX -lRm drawmap.c
```

The **-l** argument specifies a library to be linked with. The compiler driver will automatically search for something that matches with the library name. For example, `-lRm` would match with **Rmlib.def**, **Rm.def**, **Rm.lib**, **libRm.a**, and so on. In other words the compiler driver ensures that the library is found and the user does not need to supply the full name.

The BSD library is a special case. It is not possible to combine BSD compatibility with the full Posix standard, so the user has to decide whether to use BSD or not. Typically, the BSD library would be used when porting existing programs, but not when writing new programs from scratch. To use the BSD library the option **-D_BSD** should be used, for example:

```
c -oprogram -D_BSD program.c -lX
```

Below is a table of the various libraries and how to link with them. The first column gives the name of the library. The second column gives the option for the compiler driver which should be used. The third column gives a brief description of the library's purpose.

Library	How to link	Purpose
Kernel	automatic	basic part of Nucleus
Syslib	automatic	client-server interaction
Util	automatic	miscellaneous Nucleus routines
Servlib	automatic	building Helios servers
Fault	automatic	interpreting Helios error codes
Posix	automatic	main Unix library
Floating Point	automatic	arithmetic on different processors
Language	automatic	C, Fortran, Pascal etc. libraries
X	`-lX`	interaction with X graphics server
PC graphics	`-lPCgraph`	simple graphics library
BSD	`-D_BSD`	BSD Unix compatibility
Curses	`-lcurses`	Unix-style screen control
Termcap	`-ltermcap`	terminal characteristics
Debugger	`-g`	use the Helios source-level debugger
Resource Management	-lRm	control of the processor network

If a programmer builds his or her own Scanned libraries then these should not normally be exported to **/helios/lib**. The compiler driver can be made to search directories other than the default **/helios/lib** with a command line option. For example, the following line can be used to link with the library **matrix.lib** held in a separate subdirectory `../mathlibs`.

```
c -o calc -L../mathlibs -lmatrix calc.c
```

Building Scanned libraries

Libraries can be very useful things so experienced programmers will normally want to produce their own. Building Resident libraries is rather complicated (see chapter 16, *Program representation and calling conventions*). Such libraries are normally produced only by system programmers. On the other hand, building Scanned libraries is easy.

Suppose a maths library has four different modules: **fourier.c**, **matrix.c** and also **integral.c** and **simuleqn.c**. (Remember that a module normally equals a source file.) The makefile to turn these into a library would look something like this.

```
# Makefile for the maths library

.suffixes:
```

```
.suffixes: .c .o

.c.o:
c -c $*.c

objects = fourier.o matrix.o integral.o simuleqn.o

maths.lib : $(objects)
c -o$@ -j $^
```

The **-j** option to the compiler driver instructs it to take the specified object files and turn them into a Scanned library. This library can now be used as any other library, with **-lmaths**.

Some guidelines should be observed when designing Scanned libraries. Most importantly, parts of Scanned libraries are included on a per module basis. If the application program needs just one routine in a module of a Scanned library then all of that module gets included. Hence Scanned libraries consisting of a small number of large modules tend to be inefficient. In theory every single routine should have its own module, but maintaining large numbers of small source files is difficult. Hence the approach normally taken is to split the library into closely-related modules, where all of the routines relating to a particular area are put into the same module.

For example, the BSD compatibility library contains the following modules, amongst others.

1. **getopt.c** to parse program arguments.
2. **inetaddr.c** for manipulating internet and socket addresses.
3. **popen.c** holding the **popen()** and **pclose()** routine.
4. **syslog.c** for writing to the system log.
5. **fileio.c** for certain file I/O operations.

Suppose a program uses **popen()** and **pclose()**. This means the binary object would include the third module listed above, but not any of the others. If **pclose()** had been put into a different module then the final binary object would need to incorporate two BSD modules instead of one, and hence it would be larger. For most libraries, the programmer need not be too concerned about the above, because the libraries will split quite sensibly into modules anyway. However, occasionally a small amount of effort in library design will result in significant improvements in the library usage, and produce smaller binary objects.

3.2.2 Other tools

So far the only tools described have been the compiler driver **c** and the two **make** utilities. Various other tools are available to help programmers, and this subsection describes a few of them.

CDL

The CDL compiler can be used when developing parallel programs. It is described in detail in chapter 4, *CDL*, and will not be discussed further here.

RCS

The RCS system can be used for controlling complex software systems. It can keep track of when source files were changed, who changed them, and why the change was necessary. Hence if a system suddenly stops working the programmers can find out what has changed to cause this, and find out whether the changes introduced a bug or simply revealed a bug that had been lurking in the software. A typical software system goes through many different versions and releases during its lifetime, and RCS allows the programmers to work out exactly which files were used to build a specific release.

For fairly simple systems implemented by just one programmer RCS is not usually worthwhile. If multiple programmers are involved in producing the system then some sort of control mechanism is essential, and RCS serves this need.

AMPP

The **AMPP** program is a macro pre-processor. It takes a piece of text and transforms it to a different piece of text using certain rules, which can be defined dynamically. Its main purpose is for writing programs in assembler language, because it can take care of tedious jobs such as putting the right return instruction at the end of every routine. However, it could be used more generally for any system that needs to transform text files.

include

Compiling programs can take a long time. Various factors affect the amount of time taken:

1. The compiler and other tools may need to be loaded off disc. See the description of **cache** below to avoid this.

2. The source code has to be loaded off disc, and the binary has to be written to disc. This cannot be avoided.

3. The code has to be compiled, and this involves some computation by the compiling processor. This cannot be avoided.

4. Any header files needed by the program have to be loaded off disc. For many programs the header files are actually significantly larger than the program itself, and reading in the header files controls the speed of the compilation.

To speed up compilations Helios has an **include** disc, a server somewhat like the RAM disc which contains all the system header files. These files are read-only. This means that the header files are permanently in memory, and hence disc I/O is avoided. The compiler driver will automatically use the include disc if one is loaded somewhere in the network of processors. It should be installed by the system administrator, and

typically it gets started automatically on a system processor by running it from the network resource map. For more information, please consult the help system for **include** and **buildinc**.

cache

The other problem with compiling software is having to load programs off disc. To avoid this, it is possible to cache useful programs on a processor, typically from the resource map. For example, the system administrator might put the following into the resource map.

```
    Processor 07 { ~05, ~06, ~08, ~09; System;
run -e /helios/bin/cache cache cc asm make ampp emacs ls more;
```

Cached programs will not always be used automatically. To force the system to use the cached versions of the program the cache should be added to the shell's search path in the user's **.cshrc** file.

```
    set path=(/07/loader . /helios/bin /helios/local/bin)
```

If all the useful tools are cached, and if an include disc is loaded, then compilation times can be greatly reduced. Obviously this does involve a cost in terms of processor memory, so the techniques are useful only if one or more processors in the network has enough spare memory.

map

map is a simple processor monitoring utility. It can be used to display a processor's memory map, cpu usage, link statistics, and message port usage. When things go wrong it provides a simple way of determining whether the problem is lack of memory or something else. It can also be used to get a rough idea of a program's performance and what bottlenecks may exist. **map** is useful only on a single processor, and there are other utilities such as **domain** and **network** to examine what is going on in a network of processors.

bison

A common requirement in programming is to read in a text file of some sort, where the text file has a specific syntax. A compiler is the obvious example, where the syntax is that of the programming language, but it is not the only example. A raytracing application might read a description of the various objects in the picture from a text file. A text processor reads plain text interspersed with control sequences. Much of a typical parser can be generated automatically, and **bison** is a public domain tool which can help with this. In theory the programmer merely specifies the syntax used in the file, and **bison** generates the parser program. In practice everything is, of course, a bit more complicated but for many jobs using **bison** can save time.

flex

bison only forms half of a parsing system. The parser produced by **bison** accepts tokens. For example, in the C language **if** and **while** are keywords which generate specific tokens. Taking a sequence of bytes as found in a file and turning this into a sequence of tokens is the responsibility of a lexical analyser, and the **flex** program can be used to generate such lexical analysers automatically.

cc

The compiler driver **c** is a relatively simple program. It does not compile programs itself. Instead it works out what has to be done, which files must be compiled to produce which assembler files, and which of these assembler files must be linked with the right libraries to produce an executable program. The actual compilation is done by a separate program **cc**, which takes a text file containing a C program and produces an assembler output file. Later on in this subsection there is a description of how to use **cc** directly, avoiding the compiler driver.

asm

The assembler **asm** is also invoked automatically by the compiler driver to do certain jobs. It can take an assembler source file, generated by a compiler, by **AMPP**, or very occasionally by a user, and turns this into a binary version. The assembler file usually has a **.s** suffix, and the output produced has a **.o** suffix. The source file is significantly larger than the binary file, and hence keeping these **.s** files is costly in terms of disc usage.

On Transputer based systems the **asm** program serves a second purpose. It can take assembler files, either as text **.s** files or binary **.o** files, and it can link these with start-up code and the relevant libraries to produce an executable program. Hence **asm** acts as the Transputer linker as well as the assembler.

objed

objed is an object program editor which can be used to examine binary executables and change some of the characteristics. It can take various options plus the file name of the executable. The **-i** option can be used to obtain relevant information about the program.

```
% objed -i /helios/bin/ls
Image size = 5384
Object type is Program
Name is 'ls'
Stacksize = 20000
Heapsize = 4000
```

The image size is an indication of the size of the program. In addition there will be a small header at the start of the file, typically another 12 bytes. Typical types are program and module, module referring to a Resident library. Every program has a name embedded in it. When the program starts up the initial stack for the main program is set to 20000 bytes. If the program generates a stack overflow message then this has

to be increased. The initial heap size is set to 4000 bytes. Hence the program is given 4000 bytes for dynamic memory allocation when it starts up, for use by **malloc()** and similar routines. Should the program need more than these 4000 bytes then another 4000 bytes chunk will be allocated dynamically, if there is enough free memory in the processor. If the program should attempt to allocate more than 4000 bytes then a suitable chunk will be allocated directly from the system pool, bypassing the program's current heap. For applications that use a lot of dynamic memory allocation it may be desirable to increase the heap size. In addition, it may be possible to tune this heap size to match the actual allocation requirements of the application; this helps to reduce memory fragmentation. To change stack and heap size once a program has been compiled, **objed** can be used.

```
objed -s50000 -h100000 myprogram
```

At compile time the compiler driver can be given suitable options.

```
c -o myprogram myprogram.c -s10000 -h100000
```

In addition **objed** allows programmers to examine their programs and in particular which Resident libraries and which parts of Scanned libraries have been included. This is achieved with the **-m** option.

```
% objed -m /helios/bin/network
Program :    network slot   8 version 1001 size   120 datasize    0
ResRef  :     Kernel slot   1 version 2000
ResRef  :     SysLib slot   2 version 1000
ResRef  :    ServLib slot   3 version 1000
ResRef  :       Util slot   4 version 1000
ResRef  :      FpLib slot   5 version 1000
ResRef  :      Posix slot   6 version 1000
ResRef  :       Clib slot   7 version 1000
Module  :  network.c slot   9 version    1 size  5580 datasize  172
Module  :    popen.c slot  10 version    1 size   672 datasize   13
Module  :   string.c slot  11 version    1 size   700 datasize    7
Module  :   signal.c slot  12 version    1 size   424 datasize    5
Module  : nuprtnet.c slot  13 version    1 size  2640 datasize    6
ResRef  :      RmLib slot  24 version 1000
```

ResRef refers to a Resident library. Module refers to part of the user's code or to part of a Scanned library. For modules the size indicates the size of the binary code of that module, and datasize indicates the amount of static data used by that module in 4-byte words.

3.2.3 Manual compilation

In addition to the compiler driver **c** Helios has two other utilities, **cc** and **asm**, which are used to build programs. These two programs are invoked automatically by the compiler driver. **cc** is the actual C compiler, it takes some C source code and compiles it to produce intermediate assembler code. **asm** serves two purposes. Firstly, it can act as a simple assembler, taking textual assembler code as produced by the compiler or by some other means, and turning it into binary object files. Secondly, on Transputer

versions of Helios, it can take these binary object files and link them with start-up code and the necessary libraries to produce executable programs.

There are three reasons why explicit use of the compiler and assembler may be necessary.

1. In the early days of Helios the **c** compiler driver did not exist so all programming had to go through the compiler and assembler directly. Hence for historical reasons there are still makefiles that use the compiler and assembler directly instead of going through the compiler driver.

2. The compiler driver involves a small overhead. Typically it requires about 40K of memory to run, so if memory is tight then a compilation might fail because it goes through the compiler driver. Also, it is slightly less efficient to go through the compiler driver, because it involves running an extra program.

3. Some compiler and assembler options are not supported by **c**.

For these reasons, programmers may occasionally find themselves using the compiler and assembler directly, and this subsection explains how to use these programs.

The compiler

The C compiler **cc** takes a single C module and converts it to an assembler file. In the simplest case the command line would be something like this.

```
cc test.c -s test.s
```

The output file produced is an assembler text file and hence it has a **.s** suffix rather than the **.o** suffix for object files. The command line options include:

-d is used to pre-define macros, rather like **-D** option of the compiler driver. It can be used in two ways.

```
c test.c -s test.s -dSystem12
c test.c -s test.s -dSystem=12
```

The first defines the constant **System12** but does not give it a value. Hence it can be used by `#ifdef` and similar constructs but not by `#if`. The second defines the constant **System** and gives it the value **12**, so this can be used for both types of pre-processor test. By default the compiler driver automatically pre-defines three constants, and any makefile using the compiler directly should also pre-define these three.

```
cc test.c -s test.s -d__HELIOS -d__TRAN -d__HELIOSTRAN
```

Some of the Helios header files check for these constants and programs are unlikely to compile correctly if these constants are not defined. Another important constant is called **-d_BSD** to indicate whether or not the program is being compiled for Berkeley compatibility.

-i is used to specify the include file search path, in other words the directories to be searched for header files. With the compiler it is possible to specify different search paths for header files included by #**include "header.h"** and #**include**<**header.h**>. The **-i** option is used for include files inside double quotes. All the include directories should be listed as a single string.

```
cc test.c -s test.s -d__HELIOS -d__TRAN -d__HELIOSTRAN \
   -i,/include/,/helios/include/
```

When a header file is supposed to be included the C compiler works as follows. If the name specifies an absolute filename, for example

```
#include "/helios/include/stdio.h"
```

then the search path is ignored. Otherwise the compiler takes the search path and, for every entry, appends the specified name to the directory name. For example, if the header file to be included is `"header.h"` then the compiler would search for it in the current directory (the result of appending `header.h` to an empty string), then it would search for `/include/header.h`, and finally `/helios/include/header.h`. Note that it is important that all the real directories specified in the search path end with a / character, or appending the header name will generate gibberish.

-j is like **-i** but is used for include files enclosed in < > characters. It is used in exactly the same way.

-w, -e, and -f control various options in the compiler such as which warning and error messages are suppressed. These do not affect the actual code produced. More information can be found in the help entry for **cc**.

-t can be used to specify a particular Transputer processor. The recognised options include **-t8** to compile for T800 or similar processors with a built-in floating point unit, **-t4** for a T414, and **-t5** for a T425 or other processor without the built-in floating point unit but with the **dup**, **bytblt**, and similar instructions. The default is **-t4** because under Helios a program compiled for the T414 can run on any processor.

-s is used to specify the output file for the compiler. This will be an assembler text file.

-p is used to pass pragmas to the compiler. Pragmas are system specific options to the compiler. Usually they should not be put into source code because different compilers will have a different set of pragmas. The **-p** string should be followed by a letter and a number. For example, if the option **-ps0** is given then this is equivalent to the following line in the C code.

```
#pragma -s0
```

The most useful pragmas are:

1. **-ps1** can be used to disable stack checking. This will result in a small decrease in code size and a small speed-up. On Transputers there is no hardware facility for detecting stack overflows and the associated memory corruption. Hence the compiler puts extra code into the program to do the checks in software. Clearly, disabling these checks should be done only once a program has been fully debugged.
2. **-pf0** is used to disable the vector stack. This may be necessary when producing Resident libraries. A description of the vector stack mechanism is given in chapter 16, *Program representation and calling conventions*.
3. **-pg0** can be used to suppress the putting of names into the binary code. By default the C compiler will put the names of all routines in the code produced and this can be used by, for example, the stack error handling. Suppressing this will result in reduced code size and possibly a small speed-up, but again it should not be used until the program has been fully debugged.

-l is used when building Resident libraries and device drivers. It stops the compiler from outputting code for certain things including module headers, calling stubs, and static data declarations. More information is given in chapter 16, *Program representation and calling conventions*.

asm

The **asm** program can be used in two main ways. First it can take one or more assembler text files and produce the binary object files. A command line to do this is:

```
asm -p -o module.o module.s
```

The **-p** argument specifies that no linking should take place. The assembler text file **module.s** is transformed into a binary file **module.o** containing the same information, but using up rather less file space. Multiple source modules can be specified. If so then the assembler produces a single binary file containing the different modules. This is used by the compiler driver to build Scanned libraries.

```
asm -p -omaths.lib matrix.s fourier.o integral.o simuleqn.s
```

Note that the assembler can take a mixture of text **.s** files and binary **.o** files. The latter are unchanged, but are now incorporated into the single binary file. To invoke the **asm** program as a linker a command line like the following should be used.

```
  asm -o a.out /helios/lib/c0.o program.o -l/helios/lib/helios.lib \
-l/helios/lib/c.lib -l/helios/lib/bsd.lib
```

Since no **-p** argument is given the assembler will attempt to link the various parts together to produce an executable program. The first of these must be some start-up code. Unless there is some start-up code at a known fixed location within the binary file the system cannot start the program. For C programs the normal start-up code is held within the file **/helios/lib/c0.o**, which calls the routine **_main()** inside the C library. Once the C library has initialised itself it will call **main()** inside the user's program, and the user's code can now be executed. Languages other than C will have

their own versions of this file. In addition there is a file **/helios/lib/s0.o** for use by special programs which do not need the C library.

The start-up code is normally followed by the user's own code, as one or more modules. Any number of modules can be given. Finally the necessary libraries are included. **helios.lib** includes the Kernel, System library, Utility library, Server library, Fault library, Floating Point library, and Posix library. This is normally used for all programs irrespective of the language. **c.lib** contains the C library only, so this is used only when linking C programs. Again other languages will have their equivalents. The assembler has a number of other options.

-v puts the assembler into verbose mode. This causes it to report progress at regular intervals, and produce a summary at the end.

-f specifies a fast link. This means that the assembler should attempt to optimise the output produced, hopefully speeding up the code and reducing its size. This optimisation can take a long time for big programs. This facility is used by the **-O** option of the compiler driver.

s0.o

By default every C program is linked automatically with the C library, because in the vast majority of cases this is what is required. Occasionally it may be necessary or desirable to avoid using the C library. One reason would be if the target program is to run on a processor with very little memory, for example 512K. Another reason would be for writing simple Helios servers, which should use up as little memory as possible because they run continuously. Writing such programs can be difficult, as the application programmer must take care to avoid using any C library routines, which is a somewhat unusual way of programming. For example it no longer possible to use **stdin**, **stdout**, **fprintf()**, or anything similar. Only the following C routines can be used: **strlen()**, **strcpy()**, **strncpy()**, **strcat()**, **strncat()**, **strcmp()**, **strncmp()**, **memset()**, **memcpy()**, **setjmp()**, or **longjmp()**.

For such special needs Helios is shipped with an alternative piece of start-up code, namely **/helios/lib/s0.o**. Instead of calling **_main()** in the C library, this code calls **main()** in the user's program immediately. Note that the system cannot perform any initialisation on behalf of the program, so the program has to do more work than is usual. To link such a program the following command line can be used.

```
asm -v -f -olockserv /helios/lib/s0.o lockserv.o \
      -l/helios/lib/helios.lib
```

If the application programmer chooses to use the Posix library but not the C library the initialisation is relatively easy. The Posix library contains a routine **_posix_init()** which can do most of the work. The program should start with code something like this:

```
#include <unistd.h>

int main(void)
```

```
{ int  argc;
  char **argv;

  { char **argv1;
    argv = argv1 = _posix_init();
    for (argc = 0; *argv1 != (char *) NULL; argc++, argv1++);
  }

     /* The normal user program starts here. */
}
```

Using the Posix library but not the C library saves some overhead, but not all. It is possible to write applications that rely only on the libraries built into the Nucleus: the Kernel, System library, Server library and Utility library. This would save another 30K of memory, but leaves the programmer with even fewer library routines available. The start-up for such a program might look something like this.

```
#include <syslib.h>
#include <task.h>

int main(void)
{ Environ     env;

  if (GetEnv(MyTask->Port, &env) < Err_Null)
   { IOdebug("MyProgram: failed to receive environment");
     Exit(0x100);
   }
}
```

The environment block **env** will contain pointers to various vectors, including **Strv** for the program's standard streams, **Envv** for the environment strings, and **Argv** for the program's arguments. All of these vectors are terminated by a NULL pointer.

Occasionally, particularly for basic system servers such as the RAM disc, it may be desirable to start programs without an environment at all. This is possible from the **initrc** file or from the network resource map. For example the following two lines from an **initrc** file start up two programs, one with an environment and one without.

```
run /helios/lib/lockserv
run -e /helios/bin/startns startns default.map
```

The first line runs a lock server without an environment at all. Hence the lock server does not receive any arguments, environment strings, standard streams, or anything else: everything must be done the hard way. The second line runs the **startns** program with a full environment including the arguments specified on the command line, and environment strings and standard streams inherited from the **init** program. Note that if a program is written to not receive an environment at all then it cannot be started from a shell, because the shell will always attempt to send an environment.

Start-up for programs that do not accept an environment is very easy. **main()** is called without any arguments, and the program should not call **getenv()**. The program

is limited in what it can do; for example, it does not have a current directory so it cannot perform file I/O in this directory.

3.3 Servers

This section describes the various ways of interacting with servers. Essentially there are three ways of interacting with servers. The first is from the command line, for example:

```
myprog >& /logger
```

This would execute the program **myprog** and send the output to the logger server. Many Helios servers work in much the same way; for example, it is possible to use a command like this to redirect output to a file in a filing system or in a RAM disc, to a window, to the null server, to the error logger, and so on. However, it is not possible to redirect output to, for example, the mouse server because there is no reason for a mouse server to read data.

The second way is through C library or Posix library calls, from inside the user's application. Essentially this uses mechanisms defined by existing standards to perform I/O. For example, the following piece of code opens a file and writes to it.

```
FILE *str = fopen("hello", "w");
fputs("Goodbye\n", str);
```

The third mechanism involves using Helios specific facilities. For example, existing Unix standards do not describe how a mouse behaves, so a mechanism has been implemented which is suitable for the sort of hardware that typically runs Helios. If application programmers use these mechanisms then their code will not be portable to systems other than Helios. On the other hand, for some applications this is unavoidable.

This section begins with a summary of the Posix library I/O routines, which will suffice for the majority of applications. Next it gives a description of the System library routines, indicating the similarities and differences compared to the Posix library. Finally there are descriptions of some of the more common servers available under Helios: file systems; the **/window** server; the **/rs232** server; the **/centronics** server; the **/mouse** and **/keyboard** servers; the various networking servers; the Nucleus **/tasks** and **/loader**; the null server; the error logger; the real-time **/clock** server; the lock server; the raw disc server; X windows; and pipe and socket I/O.

3.3.1 Posix facilities

The Posix library provides a wide range of I/O facilities. In fact it has to support all the I/O facilities that Unix systems might use. Some Posix routines operate on named objects, for example the **unlink()** routine acts on one specific named object. Other routines operate only on open files and require a file descriptor, for example **read()** can be used to read data from an open file or server. File descriptors are simple integers starting at 0. Detailed information on specific routines can be found in the online help system or in the *Helios Encyclopaedia*. The most useful routines in the first category are:

open() is used to open a stream to a named file or server, creating/truncating the object depending on the exact open mode used. It returns a file descriptor that can be used by routines such as **read()** and **select()**.

opendir() is like **open()** but acts on a directory instead of a file.

creat() is similar to **open()**, and is used to create or truncate a file or server. It returns a file descriptor.

mkdir() is like **creat()** but creates a directory instead of a file.

unlink() is used to delete a file, provided the application has sufficient access to the file. The way it interacts with servers depends very much on the server. For example, deleting the error logger clears its memory, but deleting the mouse has no effect.

rmdir() is like **unlink()** but acts on directories rather than files. It is rarely useful for anything other than file systems.

rename() can be used to change the name of a file or object. Note that renaming a file is not the same as moving it. Renaming it is usually permitted only within one directory and cannot be used to move a file from one directory to another. Similarly renaming cannot be used to move a file from one file system to another. Such operations should be implemented by making a copy of the file and then deleting the original.

link() is used to create a symbolic link, in other words an entry in the naming tree which actually refers to some other object elsewhere in the naming tree. It is only useful inside file systems.

access() can be used to examine an application's current access rights to a file or server. Please note that the Helios protection mechanisms are not the same as those assumed by the Posix standard, so the information returned by this routine is not necessarily completely accurate. For more details please see chapter 5, *Compatibility*.

stat() fills in a **struct stat** data structure with various pieces of information about the file or server, such as its type, its size, and the time when it was last changed. This is used by, for example, **ls -l** to obtain additional information about a specific object.

getcwd() fills in a buffer with the name of the current directory. This allows applications to change the current directory with **chdir()** and restore it later on.

chdir() changes the current directory to that specified. After a call to **chdir()** any relative pathnames (that is, ones not beginning with a slash character /), are relative to the new current directory.

ctermid() puts the name of the current controlling terminal, usually the application's current window, into the specified buffer.

pipe() takes as argument an array of two file descriptors. The routine creates a new pipe, with one end allowing read-only access and the other end write-only access. File descriptors for these two ends are put into the array. Typically this routine is called just before starting another program with **vfork()** and **execve()**, to allow the new child program to interact with its parent.

socket() **bind()**, **accept()**, **connect()** and a considerable number of related routines are used for interacting with sockets, a Unix compatible mechanism for interaction between programs. Typically these routines are used for interaction between a client and a server on two different machines attached to the same internet, but the routines can be used more generally.

The most useful routines acting on existing file descriptions are:

close() terminates a stream connection to a file or server that was produced by **open()** or
create().

closedir() applies the same operation to an open directory.

readdir() and **rewinddir()** interact with an open directory to extract the data.

read() attempts to obtain data from an open file or server.

write() attempts to send data to an open file or server.

lseek() can be used to control the position within the file for the next **read()** or **write()** operation. Usually this routine has no effect on servers.

select() is available to determine whether various streams, either to files or to servers, have data to be read or can accept more data from a write.

dup() duplicates an existing file descriptor, returning a new integer.

dup2() is rather more useful. It attempts to take an existing file descriptor and open a second stream to the object. This second stream should use the file descriptor specified as the second argument. A typical use for this would be inside the child process produced by **vfork()**, to overwrite the standard input and output streams of the child with pipes to the parent.

isatty() takes a file descriptor as argument and checks whether or not the stream corresponds to an interactive stream such as a window or a serial line. This is particularly useful to check the nature of the streams inherited through the environment, for example to check that a particular stream really does refer to a window and has not been redirected to or from a file.

ttyname() can be used on interactive streams to determine the name of the stream. It is like **ctermid()** but can be used on any interactive file descriptor.

fstat() is like **stat()** but operates on an open file descriptor rather than on a named object. The information produced is usually the same, except that **fstat()** is more likely to give an accurate file size than **stat()** with some servers.

fileno() takes a C stream, in other words a **FILE** pointer such as **stdin**, **stdout**, or **stderr**, and extracts the underlying Posix file descriptor.

fdopen() performs the inverse operation to **fileno()**, taking a Posix file descriptor and turning it into a C **FILE *** stream.

cfgetospeed() cfsetospeed(), **cfgetispeed()**, **cfsetispeed()**, **tcgetattr()**, and **tcsetattr()** can be used to control window and RS232 servers.

3.3.2 System library facilities

The System library uses three different types of data. Like the Posix library it uses names of files and servers. However the System library routines do not operate on names in isolation. Instead the various routines always involve a context of some sort, and contexts are specified by **Object** data structures. Also, instead of using integer file descriptors to represent open streams the System library uses **Stream** data structures. The following code fragment indicates two ways of opening a stream to an existing file in the current directory.

```
int fd = open("data", O_RDONLY);

Object   *current_dir = cdobj();   /* get from environment */
Stream   *str = Open(current_dir, "data", O_ReadOnly);
```

In practice the Posix library always uses the current directory as the context Object. The System library is more flexible in that various different context Objects are possible. These contexts can be obtained dynamically by calls to **Locate()** or **Create()**, or they may be inherited through the environment through the **Objv** vector. A C program's environment can be obtained through the **getenviron()** call.

Both the **Object** and the **Stream** data structures are to be found in the header file called **/helios/include/syslib.h**. The names used in System library calls can be absolute, in other words beginning with a slash character /, or relative to the context Object. All of the main routines acting on names require a context Object, with the exception of **Locate()**. The purpose of the context being also to define the access rights of the application, and unless a suitable context is provided the application will have only default access to the file or server. Similarly if an absolute pathname is used then the application will have only default access. This default access may or may not suffice for the application's requirements.

The following code fragment illustrates the use of System library calls to create a new text window, as is done by the **run** and **wsh** commands.

```
Object *create_window(char *name)
{ Environ *env = getenviron();
  Object  *window_server = env->Objv[OV_CServer];
  Object  *new_window =Create(window_server, name, Type_Stream, 0,
                Null(BYTE));

  return(new_window);
}
```

The routine extracts a context Object for the current window server from the environment, and creates a new entry in the **/window** directory. This new entry is equivalent to a new window, so another window will actually pop up on the screen. A stream to this new window can now be opened in order to write to the window or read data from that window. Alternatively the window can be removed from the screen by a call to **Delete()**. The main routines in the System library acting on Objects and names are:

Locate() can be used in two ways. If it is given an absolute pathname, with or without a context, then it checks whether the file or server exists and returns a suitable Object. The application will have only default access, for example it might not be able to delete the file. Alternatively the routine can be given a context such as the current directory and a relative pathname, and it will return an Object with suitable access.

Create() returns an Object like **Locate()**, but it is used to create a new file or alternatively to truncate an existing one. The exact behaviour depends on the file system or server. For example, attempting to **Create()** a file that already exists is equivalent to truncating it. Attempting to **Create()** a window that has the same name as an existing window will succeed, and the window server will actually create a new window which has a modified name. For example, if a user executes the **wsh** command twice in a row then there will be two Create requests for a window called **shell** and the window server will create two new windows **shell** and **shell.1**. Note that the System library **Create()** is somewhat different from the Posix library **creat()**, which is just a modified version of **open()**.

Open() is used to establish a stream connection, and it will return a Stream structure that can be used for calls to **Read()**, **Write()** and so on. Like the Posix **open()** routine there are various different open modes including **O_ReadOnly**, **O_Truncate**, and **O_Create**.

ObjectInfo() is the System library's equivalent to **stat()**, but the information produced is somewhat different. Similarly there are routines **Link()**, **Delete()**, and **Rename()** which perform the obvious actions.

ServerInfo() can be used to get additional information about a server. For example, applied to a file server it gives disc usage statistics, and applied to a processor it gives performance statistics.

SetDate() can be used to change the time stamps associated with a file. It is used by the **touch()** command.

Protect() Refine() and **Revoke()** are used to implement the Helios protection mechanisms.

The routines operating on streams tend to be similar to the Posix ones. The following table indicates the equivalents.

System library	Posix equivalent
`Read()`	`read()`
`Write()`	`write()`
`Seek()`	`lseek()`
`Close()`	`close()`
`GetFileSize()`	`fstat()`
`GetAttributes()`	`tcgetattr()`
`SetAttributes()`	`tcsetattr()`
`SelectStream()`	`select()`

The System library's equivalent to the **isatty()** routine is to examine the **Flags** field of the Stream structure. In addition the System library provides a considerable number of other routines that are useful.

EnableEvents(), Acknowledge() and NegAcknowledge() are used to interact with servers
such as mice that generate real time data. An example is given in section 3.3.7 on the mouse server.

Socket(), Bind(), Listen(), Accept(), and Connect() are the System library equivalents of the Posix routines for manipulating sockets.

Load() and Execute() can be used to start programs on the local processor or on a specific processor. These are normally used only by system programs. Application programs should use the Posix library's **vfork()** and **execve()** routines for executing a single program, and the Resource Management library for executing parallel applications.

SendSignal(), InitProgramInfo(), and GetProgramInfo() can be used to interact with a running program. **SendEnv()** and **GetEnv()** manipulate program environments. Again these should not normally be used except in system programs.

Malloc() and **Free()** are the System library's equivalent of the Posix memory management routines. They offer no advantages over the Posix routines but are useful for writing certain applications that are not linked with the Posix or C libraries, as described in the previous section.

Exit() can be used to force a program termination. It works at a lower level than the C library's **exit()** routine, bypassing the C library tidying up code. In particular, if **Exit()** is used instead of **exit()** then the application's buffers may not get flushed and hence the output files may be incomplete.

MachineName() can be used to determine the full name of the current processor.

AddAttribute(), IsAnAttribute() and similar routines manipulate window and serial line attributes in a more general way than the Posix mechanisms. These are described in more detail in section 3.3.4 on the **/window** server, below.

The vast majority of applications should not attempt to use the System library. Its main purpose is to support higher-level facilities such as the Posix and C libraries.

3.3.3 File systems

Helios has a wide range of servers offering file I/O of some sort. The main ones are listed below.

1. The Helios file system can be used on processors in the network with suitable hardware, for example a SCSI hard disc. This is the main file system supported by Helios, and in particular it supports symbolic links and the full Helios protection mechanism. The file system can also be used with raw disc servers attached to a host machine, typically a spare partition on a PC.

2. NFS permits Helios to access remote file systems over the ethernet. NFS is designed around the Unix file access model so this system does not support Helios capabilities.

3. Unix I/O servers attached to the network allow access to any disc drives attached to the Unix workstation, as well as NFS drives mounted on that workstation. The same limitations apply as for NFS.

4. PC I/O servers allow access to the MS-DOS file system, usually including floppy disc drives. The MS-DOS file system is limited in many respects, in particular it can only support filenames of eleven characters, three of them in the suffix. File names are not case sensitive and the server will automatically translate names to lower case. Symbolic links and protection are not supported at all. Text files are held in a slightly different format, using two characters at the end of every line instead of one, and Language libraries such as the C library must translate data between these formats when it is read or written. Given an open Stream structure to a file it is possible to check whether or not it is on an MS-DOS compatible filing system, as shown below.

```
#include <nonansi.h>
#include <stdio.h>
#include <syslib.h>

bool is_file_msdos(FILE *str)
{ Stream *stream = Heliosno(str);
/* convert from C library to system library descriptor */
  if (stream->Flags & Flags_MSdos)
   return(TRUE);
  else
   return(FALSE);
}
```

5. The MS-DOS compatible disc server allows Helios to use floppy or hard disc hardware attached directly to a processor in the network. This server supports only MS-DOS compatible discs so it is subject to the same limitations as a disc in the I/O server.

6. Every processor in the network can run a RAM disc and this is loaded automatically on demand. RAM discs provide fairly fast I/O, but obviously the data is

not preserved if the machine is switched off or rebooted. RAM discs provide the same functionality as the Helios file system, including symbolic links and protection.

7. ROM discs are used occasionally in stand-alone systems without a host processor which boot a Nucleus from EPROM or from disc. The ROM disc may contain various useful configuration files, allowing the system to start up fully. ROM discs are read-only, and to change the files it is necessary to rebuild the ROM disc and then incorporate it into a new Nucleus. This Nucleus then has to be blown into EPROM or put on the appropriate location of the hard disc.

8. The include disc provides a read-only file system containing the Helios header files, and is designed to speed up compilation.

File systems essentially all look alike, providing a directory hierarchy with files as the leaf nodes. They permit files and directories to be opened, examined, read, closed, and so on. Usually writing to a file is also permitted. Some file systems support more advanced facilities such as symbolic links and full multi-user protection.

3.3.4 The /window server

After file I/O in its various forms, the most common type of server used under Helios is the window server. A typical window supports 80 columns and 25 rows of text output with various special escape sequences to perform operations such as clearing the screen or moving the cursor to a particular location. Shells, editors, and most of the commands supplied with Helios expect to run inside such text windows.

Helios provides multiple windows wherever possible. If it is possible to display real windows on the screen, for example on a bit-mapped display running the X window system, then every Helios window corresponds to a separate graphics window on the screen. If the underlying display provides only text output then Helios can still provide multiple windows. At any one time only one window will be visible, but one or more hot keys can be used to switch between windows and bring another one to the front. Typical keys used for hot-key switching include Alt-F1 and Alt-F2 in the PC I/O Server, and PageUp/PageDown in the **tty** server.

A **/window** server is a directory containing entries for every window. Usually it is provided in the I/O server, either on a PC or in a Unix workstation. There is also a tty server **/tty.0**, **/tty.1** and so on which is used, typically, when logging in to Helios over the ethernet. Occasionally there will be a **/console** server which supports only a single window rather than a collection of windows. The exact server used does not matter since they all behave in the same way, defined below. Programs inherit the information they need to interact with the window server through their environment, just like details of the current directory and the user's session are inherited. There is a command **tty** which displays the name of the current window. At the Posix level, the **ctermid()** routine provides the same facility.

Creating new windows

Because Helios provides multiple windows wherever possible, it must be able to create and delete windows. Normally this is done with the **wsh** and **run** commands rather

than from inside user's applications, but the facilities are available if desired. The following code fragment illustrates how it can be done.

```
    /* display a message on an empty window for a few seconds */
void write_to_new_window(char *message)
{ Environ *env     = getenviron();
  Object  *CServer = env->Objv[OV_CServer];
  Object  *window;
  Stream  *str;

  window = Create(CServer, "Message", Type_Stream, 0, Null(BYTE));
  if (window == Null(Object))
   { fputs("Error: failed to create new window.\n", stderr);
     return;
   }

  str = Open(window, Null(char), O_ReadWrite);
  if (str == Null(Stream))
   { fputs("Error: failed to open new window.\n", stderr);
     return;
   }

  (void) Write(str, message, strlen(message), -1);
  Delay(10 * OneSec);

  Close(str);
  Delete(window, Null(char));
  Close(window);
}
```

When the application has finished with the window and all streams to it have been closed then the window can be deleted. This can be done from inside the application or using the **rm** command. It is not possible to delete a window if there are still open streams to it.

Executing a program inside the new window is slightly more difficult, as it is necessary to patch the environment for the child program. If this is done using Posix **vfork()** and **execve()** then it should be done inside the child process, directly after the **vfork()**, by overwriting parts of the environment returned by **getenviron()**. If it is done with Helios calls then the environment has to be built manually anyway. The relevant fields that must be manipulated are the **OV_Console** entry of the object vector, and the first three entries in the streams vector which are the program's standard I/O streams.

Console modes

Windows can be complicated servers to program because they need to operate in a variety of different ways. For example, when a shell or an editor needs to read data from the keyboard these expect to get the data one character at a time, and the characters should not be echoed. On the other hand if the **cat** program is used to take some data from a keyboard and redirect it to a file then the program expects to receive its data a whole line at a time and, even it wanted to do so, it would not be able to echo the characters until a whole line had been typed in. To support all the different requirements

windows can be put into various different modes.

Echo mode means that the window server should echo characters typed in to a particular window. This is enabled by default and some applications such as editors will need to disable this mode.

Pause is used to enable CTRL-S and CTRL-Q handling and is enabled by default. Hence if a big file is displayed on the screen then the user can type CTRL-S to suspend output for a file and CTRL-Q to resume it later. Again applications such as editors need to disable this mode because they use these keys as input.

RawInput determines whether characters are read one character at a time or a whole line at a time. The default is cooked input, in other words line at a time. In cooked mode the window server also takes care of operations such as backspace to delete a character. The application does not receive any data until a whole line has been typed in, terminated by the return key. The input mode controls the character produced by the return key. In raw mode the return key produces a carriage return character `'\r'`, hex 0x0D. In cooked mode the return key marks the end of a record, and hence it returns a linefeed character `'\n'`, hex 0x0A.

RawOutput is less useful. By default a window is in cooked output mode which means that any linefeed characters `'\n'` , hex 0x0A, are translated into carriage-return/linefeed pairs. In raw output mode a linefeed character simply moves the cursor down one line, scrolling if necessary, but leaves the horizontal position unchanged.

IgnoreBreak controls some of the behaviour of the CTRL-C key. By default this option is disabled. If an application enables it then the CTRL-C key will be ignored completely, which may be useful occasionally.

BreakInterrupt is enabled by default. If enabled then hitting the CTRL-C key will generate an asynchronous event, probably resulting in the current application being sent a **SIGINT** signal and terminating. Applications such as editors need to disable this mode or they will be unable to process this key.

The Posix and System libraries have similar mechanisms for controlling window modes, based on an attributes structure. The following code fragment illustrates the Posix way of disabling echoing on the standard input stream.

```
#include <termios.h>

void disable_echo(void)
{ struct termios attr;

  if (!isatty(0))
   { fputs("Fatal: standard input has been redirected.\n", stderr);
     exit(EXIT_FAILURE);
   }

  tcgetattr(0, &attr);
  attr.c_lflag &= ~ECHO;
```

```
  tcsetattr(0, &attr);
}
```

The **tcgetattr()** routine extracts the current window attributes into a **termios** structure. These attributes can now be modified locally, and then they can be installed in the window server by a call to **tcsetattr()**. The **termios** structure contains four fields defining the screen mode: **c_oflag**, **c_iflag**, **c_cflag**, and **c_lflag**. At the Posix level, applications need to check these four sets of flags explicitly and set or clear individual bits. The following table shows which Helios screen modes correspond to which flags in the **termios** structure.

Mode	Posix name	Flag
Echo	ECHO	c_lflag
Pause	IXON	c_iflag
RawInput	not ICANNON	c_lflag
RawOutput	not OPOST	c_oflag
IgnoreBreak	IGNBRK	c_iflag
BreakInterrupt	BRKINT	c_iflag

Note that the flags used by Posix to control raw input and output have the inverse meaning to the Helios ones, so for example to set raw input mode it is necessary to clear the **ICANNON** bit. Equivalent code to clear the echo console mode using only Helios code is shown below.

```
#include <syslib.h>
#include <attrib.h>
#include <nonansi.h>

void disable_echo(void)
{ Attributes attr;
  Stream     *str = Heliosno(stdin);

  if ((str->Flags & Flags_Interactive) == 0)
   { fputs("Fatal: standard input has been redirected.\n", stderr);
     exit(EXIT_FAILURE);
   }

  GetAttributes(str, &attr);
  RemoveAttribute(&attr, ConsoleEcho);
  SetAttributes(str, &attr);
}
```

The equivalent routines to **tcgetattr()** and **tcsetattr()** are **GetAttributes()** and the routine **SetAttributes()**. The resulting attribute information should not be examined or changed directly. Instead the System library provides various routines to do this:

IsAnAttribute() returns TRUE if the specified attribute is enabled, FALSE otherwise.

AddAttribute() enables one specific attribute.

RemoveAttribute() disables one specific attribute.

The recognised Helios attributes for windows are: **ConsoleEcho**, **ConsolePause** and also **ConsoleIgnoreBreak**, **ConsoleBreakInterrupt**, **ConsoleRawInput** and **ConsoleRawOutput**. Using the System library has the advantage that the programmer need not worry about which field of the attributes structure holds the relevant bit, but it does mean that attributes can be manipulated only one at a time.

Screen size

Text windows come in various sizes. Usually they are all 80 columns wide, but heights vary from 20 rows to 25 and higher. Hence there must be some way of determining the current window size, and this is handled by an extension to the attributes system.

At the Posix level, the **termios** structure contains two fields **c_min** and **c_time**. The first contains the current number of rows, in other words the screen height. The second contains the number of columns, the screen width. At the System library level, the **Attribute** structure contains fields **Min** and **Time** with the same meanings. The following code fragment obtains the current screen size.

```
void find_screen_size(int *rows, int *columns)
{ Stream       *str = Heliosno(stdin);
  Attribute    attr;

  GetAttributes(str, &attr);
  *rows = attr.Min;
  *cols = attr.Time;
}
```

Usually the size of a window is fixed and it is not necessary for applications to cope with resizable windows. Helios does not have a mechanism for informing applications that the window size has changed. If an application needs to cope with such changes then it must check the current window size at regular intervals, typically in a separate thread that is **Fork()**ed off.

Output sequences

When an ordinary ASCII character is written to the screen it appears at the current cursor position, which is advanced one column. If a character is written into the final column then the cursor stays in that column. However, writing a second character without outputting a carriage return or linefeed will cause an implicit wrap (the cursor is moved to the first column of the next row, scrolling if necessary, before the second character gets written). This behaviour ensures that it is possible to write a character into the bottom right-hand corner of the screen. There are a number of special output characters such as linefeed.

`0x07 ,'\a'`, the bell character. Outputting this will produce an alert of some sort. It depends on the window server exactly how this alert is implemented. If suitable hardware is available then the bell character will actually cause a bell to be rung. Alternatively the alert might cause the screen to flash.

`0x08 ,'\b'`, the backspace character. If the cursor is already in the left-most column then outputting this character has no effect. Otherwise the cursor is moved left

one column, without overwriting the character that used to be there. To erase a character the sequence `"\b \b"` can be used.

`0x09` , `'\t'`, the tab character. This moves the cursor horizontally to the next tabbing position. Helios defines the tabbing positions to be eight characters apart at all times. If a tab character moves the cursor past the last column on the current row then the cursor will automatically move to the first column of the next line, scrolling if necessary.

`0x0A` , `'\n'`, the linefeed character. This has different effects depending on whether the window is currently in raw output mode or in cooked output mode. In raw mode a linefeed character moves the cursor down to the next row, without changing the column position. If necessary this will cause the window to scroll. In cooked mode a linefeed character will cause the cursor to move to the first column of the next row, in other words both column and row positions are affected.

`0x0B` , `'\v'`, the vertical tab character. If the cursor is already in the top row then this character has no effect. Otherwise it moves the cursor up one row, leaving the column position unchanged.

`0x0C` , `'\f'`, the form feed character. This character clears the screen, leaving the cursor position in the top left corner.

`0x0D` , `'\r'`, the carriage return character. This moves the cursor to the first column of the current row. If the cursor is already in the first column then this character has no effect.

In addition Helios windows accept a number of special escape sequences, based on the ANSI standard x3.64-1979 *Additional controls for use with American national standard code for information interchange*. These sequences all start with a **control sequence introducer** or CSI. There are two types of CSI. The first consists of single character 0x9B. The second consists of two characters, escape 0x1B, followed by **'['** 0x5B. Both CSIs have the same effect. This CSI may be followed by some data, consisting of numbers separated by semicolons. The sequence is terminated by a single character which specifies the exact operation. For example, to move the cursor to a particular position on the screen the following code fragment could be used.

```
void move_cursor(int row, int column)
{ printf("%c%d;%dH", 0x9B, row, column);
  fflush(stdout);
}
```

Note that the CSI is output as a single character whereas the row and column numbers are output as plain text. The following escape sequences are supported.

cursor up `0x9B [n] A` moves the cursor **n** rows up, leaving the column position unchanged. If this would take the cursor past the top of the screen then it sticks at the top, and the screen does not scroll down.

cursor down `0x9B [n] B` moves the cursor **n** rows down, leaving the column position unchanged. This will cause the screen to scroll if necessary.

cursor right `0x9B [n] C` moves the cursor **n** columns to the right, but not past the last
column.

cursor left `0x9B [n] D` moves the cursor **n** columns to the left, but not past the first column.

cursor on `0x9B [n] E` moves the cursor **n** lines down and to the first column. The screen will scroll up if necessary.

cursor back `0x9B [n] F` moves the cursor **n** lines up and to the first column. The screen will not scroll down.

move cursor `0x9B [m] ; [n] H` moves the cursor to row **m** column **n**. The top left corner of the screen has coordinates (1,1).

erase screen `0x9B J` erases all characters on the current row starting at and including the current cursor position. In addition it erases all characters on subsequent rows. If the cursor is currently in the top left corner then this sequence erases the whole screen, like the form feed character.

erase line `0x9B K` erases all characters on the current row starting at and including the current cursor position. Other rows are not affected.

insert line `0x9B L` inserts a blank row at the current cursor position. Rows below the current cursor position scroll down and the bottom row is lost completely. The current cursor position remains unchanged.

delete line `0x9B M` deletes the current row. All rows below the current cursor position scroll up, and the bottom row becomes blank. The current cursor position remains unchanged.

insert characters `0x9B [n] @` inserts **n** characters at the current cursor position. The characters currently below and to the right of the cursor are shifted right, and the ones on the right-hand side are lost. The current cursor position and the other rows are not affected.

delete characters `0x9B [n] P` deletes **n** characters starting at the current cursor position. The remaining characters on the current row are shifted left, and blank characters are placed in the right-most columns. The current cursor position and other rows remain unchanged.

scroll up `0x9B [n] S` scrolls the whole screen up **n** rows. Blank lines are inserted at the bottom of the screen.

scroll down `0x9B [n] T` scrolls the whole screen down **n** rows. Blank lines are inserted at the top of the screen.

set rendition `0x9B [n] m` can be used to affect the way characters are displayed on the screen. Currently the only features supported are 7 to enable inverse video and 0 to disable inverse video.

For all escape sequences that take arguments, if no digits are specified then the window server will default to the value 1.

Input sequences

Window servers usually provide input from a keyboard as well as output to a screen. The majority of keys will generate the expected ASCII characters. For example pressing key a will generate the byte 0x61, which can then be read by an application in the usual way.

Some keys such as the function keys do not have associated ASCII values. These keys generate byte sequences similar to the output escape sequences, consisting of a control sequence introducer which is always the character 0x9B, followed by one or more extra characters. The following sequences are commonly available.

For example, pressing function key 5 would generate a sequence of three bytes: the control sequence introducer 0x9B; the ASCII character 4, 0x34; and the tilde character ~ 0x7E. Helios runs on a wide range of hardware with a corresponding variety of keyboards. Applications should avoid relying on particular keys such as PageUp and PageDown because these may not always be available. For example MicroEmacs supports CTRL-B, CTRL-F, CTRL-N, and CTRL-P as alternatives for the four arrow keys in case the arrow keys do not work. See the following table for a list of keys and the sequences they generate.

Key	Sequence
Up arrow	0x9B A
Down arrow	0x9B B
Right arrow	0x9B C
Left arrow	0x9B D
Help	0x9B ? ~
Undo	0x9B 1 z
Home	0x9B H
End	0x9B 2 z
PageUp	0x9B 3z
PageDown	0x9B 4z
Insert	0x9B @
F1	0x9B 0 ~
F2	0x9B 1 ~
F3	0x9B 2 ~
F4	0x9B 3 ~
F5	0x9B 4 ~
F6	0x9B 5 ~
F7	0x9B 6 ~
F8	0x9B 7 ~
F9	0x9B 8 ~
F10	0x9B 9 ~

Special events

Occasionally an event occurs that should bypass the normal flow of events. For example, if the user presses the CTRL-C key then that user expects the current application to terminate immediately, and not wait for all previously typed keys to be processed. To cope with such requirements Helios provides an asynchronous event mechanism. In the case of the window server the only supported event is a CTRL-C break event.

Usually the shell intercepts all such events and ensures that the appropriate action is taken, which usually means sending a SIGINT signal to the current foreground application. Hence application programmers do not need to worry about special window events.

Subsection 3.3.7 on the mouse and keyboard server gives an example code fragment illustrating the use of the events mechanism. Should an application need to intercept console events then it can use similar code, substituting **Event_Break** for **Event_Mouse**.

3.3.5 The /rs232 server

Even with modern bit-mapped displays there are still uses for serial RS232 lines. Conventional dumb terminals can be attached to such serial ports and, using the tty server to provide multiple windows, it is possible to have additional users logged in to the Helios machine through these terminals. Another use involves a terminal emulator to login to a different machine from Helios, as an alternative to the ethernet mechanisms. A third use is to control a dial-up modem.

Typically an RS232 server needs to cope with more than one port. For example, a common add-on card for a Transputer system would have eight RS232 lines to connect to dumb terminals. Hence the **/rs232** server is implemented as a directory containing a number of ports, which can be viewed with the **ls** command just like any other directory. There will always be one entry in the directory, called **default**. Hence an application program can always open **/rs232/default**, without needing to know what the ports are actually called. If the server supports more than one port (for example, **com1** and **com2**), then there will be three entries in the directory called **default**, **com1** and **com2**. There will be some way of configuring the RS232 server so that default maps onto either **com1** or **com2**. Also, it is possible to rename either **com1** or **com2** to **default** to change the default port dynamically. The following three shell commands would make **com1** the default port.

```
% pushd /rs232
% mv com1 default
% popd
```

For example, a user runs the public domain communications utility kermit to connect to a remote machine. Kermit opens the server **/rs232/default** unless instructed otherwise, and the RS232 server has been configured to map default onto **com2**. Hence the user attempts to connect through **com2**. However, if it is necessary to connect through **com1** occasionally then the user can write a shell script which renames **com1** to **default**, runs kermit, and then renames **com2** to default again to restore the system. Alternatively the **-l** option to kermit could be used to explicitly specify the port.

Once an application has opened a stream to an RS232 port, the port can be reconfigured to suit particular needs. In this they are similar to windows. Just as a window may need to be set to raw input mode to suit the needs of a particular application, so do RS232 ports. The same **termios** and **Attributes** mechanisms are used, but with a different set of modes. Also, serial ports can operate at a variety of different baud rates or speeds, and they can generate asynchronous events such as modem rings.

Baud rates

RS232 lines can operate at a number of different speeds or baud rates. In theory an RS232 line should cope with different baud rates for sending and receiving data, but not all hardware can support this. For example, the 8250 UART chip used in the IBM PC and compatibles can cope with only a single baud rate for both input and output. In such a case only the input speed is used, and the output speed is ignored.

At the Posix level, the routines **cfgetispeed()** and **cfgetospeed()** can be used to determine the current baud rates. The Helios equivalents are **GetInputSpeed()** and **GetOutputSpeed()**. Similarly there are Posix routines **cfsetispeed()** and **cfsetospeed()**, and Helios routines called **SetInputSpeed()** and **SetOutputSpeed()**, to change the baud rates. The recognised baud rates are:

Baud rate	Posix name	Helios name
50	`B50`	`RS232_B50`
75	`B75`	`RS232_B75`
110	`B110`	`RS232_B110`
134	`B134`	`RS232_B134`
150	`B150`	`RS232_B150`
200	`B200`	`RS232_B200`
300	`B300`	`RS232_B300`
1200	`B1200`	`RS232_B1200`
1800	`B1800`	`RS232_B1800`
2400	`B2400`	`RS232_B2400`
4800	`B4800`	`RS232_B4800`
9600	`B9600`	`RS232_B9600`
19200	`B19200`	`RS232_B19200`
38400	`B38400`	`RS232_B38400`

Not all hardware can cope with all the baud rates, particularly 19200 baud and 38400 baud, and if an attempt is made to set the baud rate to an illegal value then the serial port will use some default instead. To detect this, the application should check the attributes again after installing a new set.

Incoming and outgoing data

Data is sent and received in units of characters which may vary in size from 5 to 8 bits. This size excludes the optional parity bit, described below. The modes for controlling the size are: **Csize**`_5`, **Csize**`_6`, **Csize**`_7` and **Csize**`_8`. Only one data size can be active at any one time, so to change the data size it is necessary to remove the current size from the **termios** or **Attributes** structure and then insert the new size. Determining the current size may involve checking up to four bits at the Posix level, or up to four calls to **IsAnAttribute()**. The usual character sizes are 8 bits without parity or 7 bits with parity.

Data is transmitted as a single start bit, followed by the character, an optional parity bit, and either one or two stop bits. To choose between one and two stop bits, you should use the **Cstopb** mode: if set, the RS232 port will use two stop bits; otherwise it will use only one.

Flow control

Flow control has to be used between the sending and receiving ends of an RS232 line to control the rate at which data is sent. The recommended approach to flow control is XON/XOFF. When the sender is transmitting data too quickly and the receiver is unable to process it quickly enough, usually because its client is not reading the data, the receiver must suspend the sender for a while to avoid overflowing its buffer. This is done by sending a single XOFF character. When the receiver is ready for more data, it should send an XON character. After sending the XOFF character, there may be some delay before the sender gets a chance to process it so the receiver must be able to buffer at least another 128 characters.

To control XON/XOFF flow control there are two modes. First, IXON controls XON/XOFF on output. If this mode is set and the application is writing down the serial line, then the other side can suspend the write by sending an XOFF character. If the mode is not set then the XON/XOFF characters may be read by the application. Note that there may be some considerable delay between the port receiving the XON/XOFF character and the application reading it, so applications should not normally perform their own XON/XOFF handling. The second mode is IXOFF and controls XON/XOFF on input. If this mode is set and the port is receiving data faster than the application is reading it, then the port can send XOFF characters to the other side. If the attribute is not set then the port may overflow its buffers, and data will be lost irretrievably. Data may also be lost irretrievably if the other side is ill-behaved and continues to send data after an XOFF.

The alternative approach to handshaking is to use the modem status lines. If the **Clocal** mode is set then these lines are ignored. If the mode is not set then the server will use the handshake lines to the best of its ability. However, the exact operation of these handshake lines is not well-defined and different pieces of hardware are likely to disagree about their interpretation. Under Helios the Data Terminal Ready (DTR) line should be kept high whilst there is an outstanding read to be satisfied, allowing the other side to continue to send data; also, while there is an outstanding write the Request To Send (RTS) line should be kept high, and the server will send data when the other side asserts the Clear To Send (CTS) line. However, RS232 servers are free to interpret these handshaking lines in different ways or ignore them completely, if this is appropriate for the hardware.

Parity

No less than six modes control the parity behaviour of an RS232 line. The first mode is **ParEnb**: if this mode is set then the server will use either odd or even parity on both input and output. Note that it is not possible to use parity on input but not on output or vice versa, because little if any hardware supports it. If the mode is not set then no parity is used, which means that the application does not need to worry about the various parity errors.

The next mode is **Istrip**. If this mode is set then all data received from the RS232 line is stripped to the bottom seven bits. This is useful if the application is unsure whether the other side is using seven bits with parity or eight bits without parity, because it ensures that at least seven bits are correct assuming no transmission errors. If

the mode is not set then the top bit is not stripped.

The **ParOdd** mode is useful only if parity has been enabled: if set, odd parity will be used; otherwise even parity is used.

The remaining parity modes control the detection of parity errors on input. Note that the server cannot take any action if there are parity errors on output because this can be detected only at the other side, and higher levels of protocol must take recovery action. The attribute **InPck** controls whether it is the client or the server that should notice parity errors: if the attribute is not set then any data received with parity errors is sent to the application program as usual (in effect, the server ignores the parity error) otherwise the server takes some recovery action depending on modes **IgnPar** and **ParMrk**. If the **IgnPar** mode is set and the server detects a parity error then the data received in error is discarded; otherwise some special characters are placed in the read stream, depending on the remaining attribute, **ParMrk**. If that attribute is not set then the data received in error will be replaced by a single byte 0x00, and it is up to the application to work out whether this 0x00 is the result of a parity error or some real data. If the **ParMrk** attribute is set then a parity error will generate two bytes, a byte 0xFF followed by a byte 0x00; this can lead to ambiguity if a character 0xFF is received correctly, possible only if the **Istrip** attribute is not set, so in that case a character 0xFF received correctly is placed in the read stream as two bytes 0xFF 0xFF.

Consider the following example: an RS232 port is configured with 7-bit characters, **ParEnb**, **ParOdd**, **InPck**, **ParMrk**, not **IgnPar**, and not **Istrip**; the data received is the byte 00111001. Parity is enabled so the first bit (that is, a 0) is the parity bit. However, odd parity is in use and there are an even number of 1s in the byte, so a parity error of some sort has occurred. **InPck** is enabled, so the parity error should be handled by the RS232 server rather than by the application. Since **IgnPar** is not set, the data received should not be thrown away, but instead placed in the read stream as a special sequence. Because **ParMrk** is set the data placed in the read stream will be two bytes, 0xFF followed by 0x00. When the application discovers a byte 0xFF in the read stream, and the next byte is an 0x00 rather than an 0xFF, this means that a parity error has occurred and it can take whatever recovery action is appropriate.

There is a related error code: if the RS232 server is kept sufficiently busy that it cannot handle the incoming data as it arrives, and some data is lost before the server had a chance to buffer it, an overrun error has occurred; this is indicated to the application as the sequence 0xFF 0x01.

Break signals

A break signal occurs when the voltage on the RS232 line drops to 0 for a time, and is usually generated when one of the sides wishes to drop the connection. First, the application must be able to generate a break signal. There are two ways to do this: it can set the baud rate to B0 or **RS232_B0** (the output baud rate if separate baud rates are supported on input and output, otherwise the input baud rate), in which case the server will reset the baud rate to its default value afterwards; alternatively, if the HupCl attribute is set and the stream to the RS232 port is closed then the server will generate a break signal automatically. Typically this mechanism is used to shut down a modem at the end of a session.

The second problem is the detection of a break signal. There are two attributes to

control this. The first is **IgnoreBreak**: if this is set, any break signals are ignored completely; otherwise the behaviour is determined by the next attribute. If **BreakInterrupt** is not set then a break signal causes a byte 0x00 to be inserted in the read stream, and the application must distinguish this byte 0x00 from a transmitted byte 0x00 or a byte 0x00 generated by parity errors given the appropriate attributes. If **BreakInterrupt** is set, the server will send a break event to an event handler if the application has installed one; if the application has not installed an event handler, the break is ignored. To install such an event handler the application should use an **EnableEvents()** call with **Event_RS232Break** as one of the arguments. Subsection 3.3.7 on the mouse and keyboard servers gives an example of how to use the **EnableEvents()** mechanism.

Modem interrupts

One of the lines specified in the RS232 protocol is **Ring Indicator** (RI). This is used mainly by dial-up modems, to inform an application that somebody is trying to dial in. To detect such events, the application should install an event handler by calling **EnableEvents()** with **Event_ModemRing** as one of the arguments.

The default configuration

The default configuration for a Helios RS232 port is as follows: 9600 baud for both input and output; 8 bits per character, and one stop bit; parity and Istrip disabled; XON/XOFF flow control enabled on both input and output with hardware handshaking disabled; break interrupts enabled, but the client has to install an event handler; HupCl disabled. This configuration should work correctly on all implementations.

Mode names

The following table gives the names and fields associated with the various RS232 modes, for both Posix and Helios calls.

Mode	Posix name	Posix field	Helios attribute
Csize_5	CS5	c_cflag	RS232_Csize_5
Csize_6	CS6	c_cflag	RS232_Csize_6
Csize_7	CS7	c_cflag	RS232_Csize_7
Csize_8	CS8	c_cflag	RS232_Csize_8
Cstopb	CSTOPB	c_cflag	RS232_Cstopb
IXON	IXON	c_iflag	RS232_IXON
IXOFF	IXOFF	c_iflag	RS232_IXOFF
Clocal	CLOCAL	c_cflag	RS232_CLocal
ParEnb	PARENB	c_cflag	RS232_ParEnb
Istrip	ISTRIP	c_iflag	RS232_Istrip
ParOdd	PARODD	c_cflag	RS232_ParOdd
InPck	INPCK	c_iflag	RS232_InPck
IgnPar	IGNPAR	c_iflag	RS232_IgnPar
ParMrk	PARMRK	c_iflag	RS232_ParMrk
HupCl	HUPCL	c_cflag	RS232_HupCl
IgnoreBreak	IGNBRK	c_iflag	RS232_IgnoreBreak
BreakInterrupt	BRKINT	c_iflag	RS232_BreakInterrupt

Examples

Suppose an application needs to open an RS232 port, either the one specified or the default one. This port should operate with 8-bit characters, XON/XOFF flow control, and ignoring break events. The application does not know what parity to use so the 8-bit is stripped off and parity errors are ignored. The port should operate at 9600 baud in both directions. The following two code fragments show how this can be done at the Posix and at the Helios level.

```
int open_port(char *name)
{ int fd;
  struct termios trm;

  fd = open((name == NULL) ? "/rs232/default" : name, O_RDWR);
  if (fd < 0)
   { fputs("open_port: failed to open rs232 port.\n", stderr);
     return(-1);
   }

  tcgetattr(fd, &trm);
  trm.c_cflag &= ~(CS5 | CS6 | CS7 | CSTOPB | CLOCAL | PARENB |
                   HUPCL)
  trm.c_cflag |=  (CS8);
  trm.c_iflag &= ~(INPCK | PARMRK | BRKINT);
  trm.c_iflag |=  (IXON | IXOFF | ISTRIP | IGNBRK);
  tcsetinputspeed(&trm, B9600);
  tcsetoutputspeed(&trm, B9600);
  tcsetattr(fd, &trm);

  return(fd);
}
```

```
Stream *open_port(char *name)
{ Object       *port;
  Stream       *result;
  Attributes   attr;

  if (name == Null(char)) name = "/rs232/default";

  port = Locate(Null(Object), name);
  if (port == Null(Object))
   { fprintf(stderr, "open_port: cannot find %s\n", name);
     return(Null(Stream));
   }

  result = Open(port, Null(char), O_ReadWrite);
  if (result == Null(Stream))
   { char buf[80];
     Fault(Result2(port), buf, 80);
     fprintf(stderr, "open_port: failed to open %s, fault %s\n",
             port->Name, buf);
```

```
    Close(port);
    return(Null(Stream));
   }

  Close(port);

  GetAttributes(result, &attr);
  RemoveAttribute(&attr, RS232_Csize_5);
  RemoveAttribute(&attr, RS232_Csize_6);
  RemoveAttribute(&attr, RS232_Csize_7);
  AddAttribute(&attr,    RS232_Csize_8);
  RemoveAttribute(&attr, RS232_Cstopb);
  RemoveAttribute(&attr, RS232_CLocal);
  RemoveAttribute(&attr, RS232_ParEnb);
  RemoveAttribute(&attr, RS232_HupCl);
  RemoveAttribute(&attr, RS232_InPck);
  RemoveAttribute(&attr, RS232_ParMrk);
  RemoveAttribute(&attr, RS232_BreakInterrupt);
  AddAttribute(&attr,    RS232_IXON);
  AddAttribute(&attr,    RS232_IXOFF);
  AddAttribute(&attr,    RS232_Istrip);
  AddAttribute(&attr,    RS232_IgnoreBreak);
  SetInputSpeed(&attr,   RS232_B9600);
  SetOutputSpeed(&attr,  RS232_B9600);
  SetAttributes(result, &attr);

  return(result);
}
```

3.3.6 The centronics server

In addition to serial RS232 ports, some hardware, notably PC I/O processors, can be equipped with parallel centronics ports. These ports are much easier to control because centronics ports work at a single speed and there are no configuration options. The **/centronics** server consists of a simple directory. This will always contain an entry **default**, like the RS232 server. If there are more than one parallel ports then the directory will contain additional entries for every port. Again the server usually provides some mechanism for specifying the initial default, and this default can be changed dynamically.

```
% pushd /centronics
% ls
default      lpt1     lpt2
% mv lpt2 default
% popd
```

A centronics server provides only a simple transport service. Typically the server would be used by a higher-level printer spooler which is responsible for queueing users' print jobs.

3.3.7 Mouse and keyboard servers

Some servers, notably a mouse, can generate data rapidly. This data may need to be transmitted from the processor with the mouse hardware to the processor using it, typically the processor running the X windows server. If each piece of data were to be read separately then this would require two messages through the network, a read request and a reply from the mouse server. This could consume a large proportion of the available communications bandwidth, significantly slowing down file I/O and users' parallel applications. To avoid this Helios has an alternative mechanism, the event system. Essentially an application such as the X server registers its interest in one or more types of event with the server. Until this registration is cancelled the server will now send all such events directly to the application, without any further requests. For example, the mouse server will automatically send any mouse events directly to the X server without having to be asked.

Another use for the event mechanism is to handle asynchronous events. For example, suppose the user presses the CTRL-C key to abort the current foreground application. In a strictly synchronous mode this would have no effect until all previous keys had been read in and processed, and the CTRL-C key itself had been read in. This is rather unsatisfactory. Hence under Helios the CTRL-C key can cause an asynchronous event to be sent by the window server to whichever application has registered itself, usually the shell.

The main routine used with the events mechanism is **EnableEvents()**. This routine acts on an open stream and sends a message to the server. It takes a second argument, the type of event to enable. The routine returns a message port which should be used for low-level message passing.

The following code fragment illustrates the use of the event mechanism to receive mouse movements:

```
#include <ioevents.h>

extern void handle_mouse_event(IOEvent *);

void start_mouse(char *name)
{ Object     *server;
  Stream     *str;
  BYTE       buffer[IOCDataMax];
  MCB        message;
  Port       incoming;
  int        rc, i;

  if (name == Null(char)) name = "/mouse";

  server = Locate(Null(Object), name);
  if (server == Null(Object))
   { fprintf(stderr, "start_mouse: cannot find %s\n", name);
     exit(EXIT_FAILURE);
   }

  str = Open(server, Null(char), O_ReadWrite);
  if (str == Null(Stream))
   { Fault(Result2(server), buffer, IOCDataMax);
```

```
    fprintf(stderr, "start_mouse: failed to open %s, fault %s\n",
             server->Name, buffer);
    Close(server);
    exit(EXIT_FAILURE);
   }

  Close(server);

  incoming = EnableEvents(str, Event_Mouse);
  if (incoming == NullPort)
   { Fault(buf, Result2(str), IOCDataMax);
    fprintf(stderr,
            "start_mouse: failed to enable mouse %s, fault %s\n",
              str->Name, buffer);
    exit(EXIT_FAILURE);
   }

  forever
   { message.Data        = buffer;
    message.Timeout     = -1;
    message.MsgHdr.Dest = incoming;

    rc = GetMsg(&message);
    if (rc < 0)
     { if (((rc & EC_Mask) == EC_Error) ||
           ((rc & EC_Mask) == EC_Fatal))
       break;
      else
       continue;
     }

    for (i = 0; i < MsgHdr.DataSize; i += Mouse_EventSize)
     { IOEvent *event = (IOEvent *) &(buffer[i]);
      handle_mouse_event(event);
     }
   }

     /* This is reached only if there is a serious error,   */
     /* if another program has grabbed the mouse port. */
  Fault(rc, buffer, IOCDataMax);
  fprintf(stderr, "start_mouse: lost contact with %s, fault %s\n",
          str->Name, buffer);
  Close(str);
  exit(EXIT_FAILURE);
}
```

In theory several events can actually be packed together into one single message of up to **IOCDataMax** bytes. This is certainly possible with mouse and raw keyboard servers. It is unlikely to happen when using the window server's **Event_Break** or with the RS232 server's **Event_RS232Break** or **Event_ModemRing**. The event structure contains various bits of information. Full details of these can be found in the header files **ioevents.h**. For the mouse server the important fields are:

event->Mouse.X a 16-bit integer giving the new horizontal position of the mouse. This integer can take values in the range 0-32767, and if the mouse is moved too far in one direction this number will wrap. The number is not meant to refer to a real screen position. However, because the number is absolute rather than relative it allows the application to recover quickly from any lost event messages: a lost message simply results in a slightly abrupt jump rather than a smooth cursor movement.

event->Mouse.Y is another 16-bit integer giving the new vertical position of the mouse. It has the same behaviour as the X coordinate.

event->Mouse.Buttons reflects any changes to the mouse buttons state. It can take various different values such as **Buttons_left_Down**, all defined in the header file **ioevents.h**, describing the change to the button state.

The raw keyboard server is needed by systems such as X which need to do all their own keyboard handling. Normally when a key is pressed the window server will immediately generate a single ASCII character. If the key is held down then, after a short delay, the window server will start auto-repeating. When the key is released the auto-repeat stops but no further data is generated. With a window server it is not possible to work out which shift keys, control keys, Alt keys, and so on are currently held down.

The data supplied by a window server does not suffice for X. Under X it is possible to hold down a shift key and press a mouse button at the same time, and this may have a different effect from simply pressing the mouse button by itself. Hence the X server must be informed as soon as a shift key is pressed, and again as soon as the key is released. To achieve this Helios has a raw keyboard server **/keyboard** which generates events, just like the mouse events. The keyboard event structure contains two useful fields:

event->Keyboard.Key is a scancode for the key. In general this will bear no relation to the ASCII values normally associated with that key, and the application will need its own tables to convert scan codes to ASCII.

event->Keyboard.What is set to either **Keys_KeyUp** or to **Keys_keyDown**. It indicates whether the key has been pressed or released.

The mouse and keyboard servers are not normally accessed directly by application programmers. Instead they are used by higher-level graphics software such as the X server, and application programs should interact with this higher-level software, typically through the X library, to obtain mouse and keyboard data.

3.3.8 Networking servers

The Helios networking software includes various servers: the Network server or **/ns** controls the network as a whole; the Session Manager or **/sm** controls the various users logged in to the Helios machine; there is a Task Force Manager associated with every user's session, with a name derived from the user name and two subdirectories **tfm** and **domain**.

The networking servers support some simple operations. For example, it is possible to list the contents of the **/ns** directory with the **ls** command to find out the names of all the processors and subnetworks in the machine. Similarly the **/sm** directory can be listed to give details of the various users logged in. For anything more complex the networking software comes with its own Interface library, the Resource Management library. Applications should go through this library rather than interact with the servers directly.

3.3.9 /tasks and /loader

These servers are part of the Helios Nucleus, so they are present on every processor. Between them they permit program execution on the processor. Usually this happens automatically by executing a shell command or CDL script, using **vfork()** and **execve()** in the Posix library, or using the routines in the Resource Management library. Accessing **/tasks** and **/loader** directly is not usually possible (these servers are protected). On the rare occasion that direct access is possible and desirable, the following code fragment indicates how this may be achieved.

```
#include <syslib.h>

Stream *run_program(Object *processor, Object *program)
{ Object       *procman    = Locate(processor, "tasks");
  Object       *loader     = Locate(processor, "loader");
  Object       *code       = Load(loader, program);
  Object       *exec       = Execute(procman, code);
  Stream       *progstream = Open(exec, Null(char), O_ReadWrite);
  Environ      env;
  Environ      *myenv      = getenviron();
  char         argv[4];
  Object       *objv[OV_End + 1];

  env.Strv         = myenv->Strv;   /* inherit standard streams */
  env.Objv         = objv;
  env.Envv         = myenv->Envv;   /* inherit environment strings */
  env.Argv         = argv;

  argv[0]          = objname(program->Name);
  argv[1]          = "Hello";
  argv[2]          = "world";
  argv[3]          = Null(char);

  objv[OV_Cdir]    = myenv->Objv[OV_Cdir];
  objv[OV_Task]    = exec;
  objv[OV_Code]    = code;
  objv[OV_Source]  = program;
  objv[OV_Parent]  = myenv->Objv[OV_Task];
  objv[OV_Home]    = myenv->Objv[OV_Home];
  objv[OV_Console] = myenv->Objv[OV_Console];
  objv[OV_CServer] = myenv->Objv[OV_CServer];
  objv[OV_Session] = myenv->Objv[OV_Session];
  objv[OV_TFM]     = myenv->Objv[OV_TFM];
  objv[OV_TForce]  = myenv->Objv[OV_TForce];
```

```
  objv[OV_End]     = Null(Object);

  SendEnv(progstream->Server, &env);

  Close(exec);
  Close(code);
  Close(loader);
  Close(procman);
  return(progstream);
}
```

Obviously in a real program it would be necessary to test for errors throughout the above code fragment to cope with running out of memory or any of the other possible failures. The routine returns a stream to the executing program which could be used for **InitProgramInfo()** and **GetProgramInfo()**, or alternatively for **SendSignal()**, but not for both.

3.3.10 The null server

The **/null** server is a very simple server. It is loaded on demand into any processor, so there is no need to start it explicitly from the **initrc** file or from the network resource map. The **/null** server can be opened like any file. Data written to the server is discarded. Any attempt to read from the server will return end-of-file immediately. Typically the server is used only from the command line as a way of discarding output while retaining diagnostics. For example the following command compiles a program, discarding the assembler file produced as output but leaving the diagnostics messages on the screen.

```
% cc -D__HELIOS -D__TRAN -D__HELIOSTRAN hello.c > /null
Helios C 2.03 15/01/91
(c) Copyright 1988-91 Perihelion Software Ltd.
All rights reserved.
...
```

3.3.11 The error logger

The error logger **/logger** is designed to provide an emergency debugging facility. The server is fairly simple and as independent as possible, so that even if other parts of Helios are failing then it should still be possible to access the error logger. Care must be taken not to abuse it. Typically any data written to the logger appears on the system console so that the system administrator is informed and can take appropriate action. Hence it should be used mainly for data that the system administrator needs to know about.

Within a single-user environment the user is effectively the system administrator, so these guidelines can be relaxed somewhat. In particular in such an environment it is possible to use the error logger to help debug applications. Any such debugging code should be removed from the final product, or the application may be unsuitable for a multi-user environment.

The error logger is primarily a write-only server. An application can open a stream to **/logger** and write to it. Alternatively the BSD compatibility library provides a routine **syslog()** that can be used. In addition Helios provides two routines **IOdebug()** and **IOputs()** which interact directly with the error logger by low-level message passing, bypassing the streams mechanism of the System library. The error logger completely ignores file positions when writing data, and any data written is always appended to the end. This prevents accidental or deliberate overwriting of previously logged information.

Where possible the error logger will buffer some or all of the data written to it. If a convenient and reliable filing system is available then it can be made to use a file for this buffer, and the amount of data held is limited by the size of the filing system. Alternatively the error logger could buffer a fixed amount of data in memory, typically 10K, and overwrite old data when it runs out of buffer space. The data in the buffer can be retrieved simply by opening a stream to **/logger** and reading it, typically through **cat** or **more** from the command line. If an editor is used to examine the contents of the logger's buffer then care must be taken not to write the data back at the end of the editing session, or the whole contents of the buffer will be appended to the end of the logger.

Occasionally it may be desirable to clear the contents of the buffer, and this can be done simply by deleting it, for example by the command **rm /logger**. This delete does not terminate the server, it merely empties the buffer.

The exact behaviour of the error logger depends on its implementation. If it is part of the I/O Server then it can use the host's filing system as a buffer. The logger can be configured from the **host.con** file and at run-time using keyboard control sequences to send its data to the screen only, to a file in the host filing system, or to both. On the other hand, if the error logger runs as a separate server within the Helios network then it will use a buffer in memory to hold the data. In addition it can be configured to use a user-defined device driver and/or to send data to some other stream, typically a file or a window.

3.3.12 Real-time clock

Some networks may provide a clock service to determine the current time. Helios networks do not normally have a battery-backed clock attached to every processor. Hence every processor receives the current time when it is booted up, and maintains its own software clock as accurately as possible. After a while it is possible for the various processors to have slightly different ideas about the current time, with a possible drift of several seconds, but experience to date indicates that this is not a problem for the vast majority of applications. Should an application need a highly accurate time value it can examine the **/clock** server. From the command line this can be done as follows:

```
% ls -l /clock
v rwe---da   0        0 Tue Apr 16 15:11:16 1991 clock
```

Alternatively the Posix **stat()** or the System library **ObjectInfo()** routines can be used. The **/clock** device normally does have a battery-backed hardware clock associated with it so it can use hardware to maintain an accurate time value. However, accessing the clock server involves at least one request and reply message which may

have be routed through the network, with potentially an unpredictable delay, so even this technique will not give a completely accurate time.

Very occasionally it may be necessary to reset the hardware clock, for example because the batteries had to be changed. This may not always be possible from Helios. For example, if Helios is hosted from a Unix workstation such as a Sun then the clock can only be set by the super user on that Sun. If the clock can be set then the **date** command may be used to achieve this: please see the *Helios Encyclopaedia* or the online help system for details. Alternatively it could be done from inside a user's application using the System library's **SetDate()** routine.

3.3.13 The lock server

The Helios lock server can be run in a network to permit locking of resources between different applications. To avoid ambiguity there should only ever be one lock server in a network, and typically this would be run from either the **initrc** file or from the network resource map. Essentially the lock server provides only two facilities: create named lock and delete named lock. The following code fragments give routines which interact with the lock server.

```
bool Lock(char *name)
{ Object *lock_server = Locate(Null(Object), "/lock");
  Object *lock;

  if (lock_server == Null(Object))
   { fputs("Lock: there is no lock server in this network.\n",
           stderr);
     exit(EXIT_FAILURE);
   }

  lock = Create(lock_server, name, Type_Stream, 0, Null(BYTE));
  Close(lock_server);
  if (lock == Null(Object))
   return(FALSE);
  else
   { Close(lock);
     return(TRUE);
   }
}

void Unlock(char *name)
{ Object *lock_server = Locate(Null(Object), "/lock");
  Object *lock = Locate(lock_server, name);

  (void) Delete(lock, Null(char));
  Close(lock);
  Close(lock_server);
}
```

The first routine attempts to create the named lock, returning TRUE for success or FALSE for failure. The probable cause of failure is that there is already a lock with

that name, although other failures such lack of memory in the lock server are also possible. The second routine removes the named lock.

3.3.14 Raw disc servers

The **/rawdisk** server provides raw access to one or more hard discs. A raw disc server has no file system, directory structure, or anything else. It appears as a simple file of, perhaps, 40 megabytes. This file can be read from or written to only in blocks of 512 bytes, in other words the sector size. Typically a raw disc server can be used to implement a higher-level filing system. For example in a PC hosted system one or more spare partitions on the PC hard disc can be turned into raw disc drives, and the Helios file system can then be run using these partitions. Another possible use for a raw disc server is for applications which require very rapid I/O, such as databases, and which can organise the whole disc themselves to meet the application's requirements.

A **/rawdisk** server is a directory of one or more partitions. Each partition is treated completely separately, so for example it is possible to run the Helios filing system in one partition **/rawdisk/0** and use a second partition **/rawdisk/1** for an application such as a database. The partitions are named **0**, **1**, and so on.

ObjectInfo() on a raw disc partition gives the size of the partition in sectors rather than in bytes, each sector being 512 bytes. The same information is produced by Posix **stat()** and **fstat()**. For example the **ls** command might produce the following output.

```
% ls -l /rawdisk
2 Entries
f rw----da  0   41668 Tue Apr 16 09:53:16 1991 0
f rw----da  0   20834 Tue Apr 16 09:53:16 1991 1
```

In this case the **/rawdisk** server contains two partitions `0` and `1`, and these have size of 20 Mbytes and 10 Mbytes respectively. The sizes are given in sectors rather than in bytes because this information is likely to be clearer.

The stream operations supported by the **rawdisk** server are **read()**, **write()**, **lseek()**, and **close()**. I/O must always involve multiples of 512 bytes, for example it is illegal to attempt to read 1000 bytes.

3.3.15 The X window system

To interact with an X server application programs should use the X library, and possibly higher-level libraries such as the toolkit, the Widget library, Motif, and so on. Applications should **never** interact directly with the X server, for example by opening sockets and writing to it. For more information on X see the Helios X window system manual.

3.3.16 Pipe and socket I/O

There are two correct ways to create a pipe. The first involves letting the system do it for you. For example, when starting a parallel application using CDL or the Resource Management library, the system will automatically create the necessary pipes and the individual programs will inherit these files in their environment. Similarly executing

the command **ps all | more** causes the shell to create the pipe, and the two programs inherit this pipe in their environments. The second correct way is to use the Posix library's **pipe()** routine shortly before executing a child program. The child program inherits the pipe in its environment. This is illustrated by the following code fragment.

```
      /* Run a child program, and return a C FILE pointer    */
      /* that corresponds to a pipe to that child's standard */
      /* input. The child program is identified by the first */
      /* entry in the argument vector.                       */
FILE *run_child(char **argv)
{ int   pipe_descriptors[2];
  FILE  *result;
  int   pid;

  pipe(pipe_descriptors);    /* create the two ends of the pipe */

  pid = vfork();
  if (pid == 0)
   {       /* Executing in the child process             */
           /* Make a copy of the pipe's read-only end.   */
     dup2(pipe_descriptors[0], 0); /* overwriting stdin  */
           /* and close the unnecessary file descriptors */
     close(pipe_descriptors[0]);
     close(pipe_descriptors[1]);
           /* finally start the child program.           */
     execvp(argv[0], argv);
   }
  else
   {       /* Executing in the parent process.           */
           /* Turn the write-end of the pipe into a C    */
           /* FILE * stream.                             */
     result = fdopen(pipe_descriptors[1], "w");
           /* and close the unnecessary file descriptor. */
     close(pipe_descriptors[0]);
     return(result);
   }
}
```

To use a pipe the application should call the Posix library's **read()**, **write()**, **close()** and **select()** routines. Pipes are not quite synchronous, so if one end writes to a pipe before the other end reads then a small amount of data can be buffered by the system. The exact size of the buffer is not defined. Using language-level I/O on pipes, for example the C library's **fprint()** routine, is fine provided communication is one way only, in other words if the application involves a simple pipeline. For a more complicated arrangement language-level I/O can give problems in some cases because the C library itself will perform some buffering. This subject is discussed in more detail in the CDL chapter.

The correct way to use sockets, either within a Helios network or over the ethernet, is to use the Posix calls **socket()**, **accept()**, **connect()**, and so on. System library equivalents are available if necessary, but these provide little or no extra functionality. The socket calls return file descriptors which should be used in the same way as pipe

file descriptors. See chapter 6, *Communication and Performance*, for a more complete description of pipe and socket usage.

3.4 Protection: a tutorial

This tutorial show how you can use the Helios protection mechanism to protect your files from other users, and how you can then use it to give other users limited access to your files. It takes the form of an example session which may be followed by the reader at his own machine. Only the Helios File Server and the **/ram** server can support the full protection mechanism. For simplicity this tutorial uses only the **/ram** server so you can follow it on your own machine even if you do not have the file server.

We will start by moving into the ram server and creating two user directories:

```
% cd /ram
% mkdir dale bob
```

We now change the prompt in this shell and start another shell in another window:

```
% set prompt="dale: "
dale: wsh
```

In the new window we also change the prompt, and also change our current directory:

```
% set prompt="bob: "
bob: cd bob
```

Now go back to the first window. If you are using the PC window system you can do this with ALT-F1, if you are using Windows 3, your host is a SUN workstation, or if you are using the Helios X Window server, you should move the mouse and click on the window. In the latter cases you should now rearrange your windows so you can see both at once. From now on the window you should type the commands to will be indicated by the prompt.

We start by moving into the appropriate user directory, and print out the **/ram** server's **access matrix**:

```
dale: cd dale
dale: matrix /ram
d rwv----a:rw-x----:rw--y---:r----z-- ram
```

The **matrix** command simply prints out the access matrix associated with the file or directory given, in this case **/ram** (from now on the word **object** will be used where it does not matter whether we are talking about a file or a directory, or even some other object like a processor, pipe, task or task force). The matrix can be thought of as an array of bits, eight wide by four deep, making 32 bits in all. The matrix for **/ram** is:

column:	r	w	v	x	y	z	d	a
row v:	1	1	1	-	-	-	-	1
row x:	1	1	-	1	-	-	-	-
row y:	1	1	-	-	1	-	-	-
row z:	1	-	-	-	-	1	-	-

Each of the four rows is named by one of the letters `v`, `x`, `y` and `z`. Each of the eight columns corresponds to an access right, and is also named by letters. Four of the columns have the same meaning in all objects: `r` for read access, `w` for write access, `d` for the right to delete the object, and `a` for the right to alter the object's access matrix. The remaining four columns depend on the type of the object to which the matrix is attached. For files, only one of these columns is used, and is used to denote execute permission, with the letter `e`. For directories the remaining four columns correspond to the four matrix rows, `v`, `x`, `y` and `z`. These are used to select which rows of the matrices attached to objects inside the directory will be used to control access to those objects; exactly how this works will be shown later.

In the output of the **matrix** command shown above the letter `d` indicates that this is a directory. Following this it prints each row of the matrix out in the order v, x, y and z separated by colons. The bits set in each row are shown by the letters which correspond to the columns they occupy; unset bits are shown by a hyphen. Hence, in this matrix, the **/ram** server has bits set in the `v` row to give read, write and alter access to the directory itself, and `v` access to objects inside the directory. When typing in a matrix, or any set of access rights, it is not necessary to follow this format exactly. The hyphens may be left out, and the column letters may be given in any order. So, the matrix above may be written as `awrv:wxr:ryw:zr` without confusion.

We can change the access matrix of an object with the **chmod** command:

```
dale: chmod v=rz x=rza y=rz /ram
dale: matrix /ram
d r----z--:r----z-a:r----z--:r----z-- ram
```

The arguments to **chmod** are similar to the arguments of the UNIX command of the same name. The argument `v=rz` alters the `v` row of the matrix to just have bits set in the `r` and `z` columns. The same effect could have been achieved with the sequence `v-wva v+z` which causes the `w` `v` and `a` bits to be cleared and the `z` bit to be set.[4]

The access matrix controls the access rights that users have over objects just like the mode bits in UNIX. The four rows may be though of as corresponding to similar access classes, where the `v` row controls the owner's access rights, the `z` row controls the access rights of the general public, and the `x` and `y` rows can perform the same job as the group rights. However, unlike UNIX modes, the access rights a user gets are not fixed only by the matrix in the object, but can also be influenced by it parent directory, and any other directory the user must pass through to access the object.

As an example, let us create a file and look at its access matrix:

```
dale: echo "The owls are not what they seem" >audrey
dale: matrix audrey
f rw----da:rw----d-:rw------:r------- audrey
```

This is the default matrix for any file, it gives the owner full access rights while the other classes get successively fewer rights until the general public can only read it.

Directories also have a default matrix:

```
dale: matrix .
d rwv---da:rw-x--d-:rw--y---:r----z-- ram
```

[4] See the *Helios Encyclopaedia* entry for **chmod** for more details.

As far as the r, w, d and a bits are concerned, this follows the same pattern as for files. Additionally, each row in the matrix has its own corresponding bit set (so the v row has a bit set in the v column). This simply propagates the same access rights down into the objects in this directory. So, the directory owner, whose rights are controlled by the v row, will also have his rights controlled by the v row in the matrices of the objects inside the directory.

A user's current access rights to any object can be examined using the **access** command:

```
dale: access .
d rwv---da /00/ram/dale
dale: access audrey
f rw----da /00/ram/dale/audrey
```

This produces output similar to `matrix` except that only one set of access rights are printed out and not four matrix rows. Here, the user has full access to both his current directory and to the file `audrey`. However, if the same commands are executed, on the same objects, but using a different way of naming the objects, a different result will be seen:

```
dale: access /ram/dale
d r----z-- /00/ram/dale
dale: access /ram/dale/audrey
f r------- /00/ram/dale/audrey
```

By using absolute path names, rather than naming them relative to the current directory, we get only the public access rights. The reason for this is that we changed the matrix of **/ram** to contain only r and z bits in all its rows. This means that whatever we started with, our access rights to the **/ram** directory will always be just rz:

```
dale: access /ram
d r----z-- /00/ram
```

The r bit means that we are only allowed to read the directory, we are not allowed to create new entries, delete it, or change its matrix. The z bit means that for any entries in the directory **/ram**, we can only have the access rights contained in their z matrix rows. In the directory `dale` this row is `r----z--`, so we get exactly the same set of rights as for **/ram**, and in particular, we only get z access rights to the file `audrey`, whose z matrix row is `r-------`, meaning that we can only read it.

The reason why we get different access results is that the Helios shell possesses a **capability** for the directory `dale`. A capability is a set of access rights for a particular object. The shell's capability for the directory `dale` forms part of its current directory (which can be changed with the **cd** command). Whenever the shell runs a command it passes this capability on to it. The access rights to any object named relative to the current directory are calculated relative to the set of rights stored in the capability. Whenever a capability for an object is obtained, it effectively takes a snapshot of the access rights which are then in force. If the matrix is subsequently changed, this does not affect the rights stored in the capability, which are still enforced.

In our example the shell's capability for `dale` contain the rights `rwv---da`, exactly the contents of the matrix's v row (which is where it came from). Because the v bit is set, access to objects within `dale` will use their v matrix rows, so access to `audrey` is `rw----da` which is exactly what we saw earlier. However, if a program

has a capability for an object, its access rights are derived only from the capability and not from the matrix in that object at all. So, if we change the matrix attached to `dale`, we will not affect the shell's access rights at all, but we will affect the rights of other users:

```
dale: chmod v-rwd x-d z-rz .
dale: matrix .
d --v----a:rw-x----:rw--y---:-------- dale
dale: access .
d rwv---da /00/ram/dale
dale: access /ram/dale
could not locate /ram/dale: c6040000
```

Here we remove some rights from the matrix, yet our access to `dale` has remained the same. In particular, note that while all `d` bits have been removed from the matrix, we still have `d` access to `dale`. We have also taken the opportunity to remove all bits in the `z` row of `dale`'s matrix. Because **/ram** restricts all accesses to just the `z` category, this eliminates all external access rights to `dale`.

Meanwhile, in the other window, that shell has a capability for directory `bob` but has no access to directory `dale` or any of its contents:

```
bob: access .
d rwv---da /00/ram/bob
bob: mkdir laura
bob: access laura
d rwv---da /00/ram/bob/laura
bob: access /ram/dale
could not locate /ram/dale: c6040000
bob: access /ram/dale/audrey
could not locate /ram/dale: c6040000
```

The directory `bob` can also be protected against the outside world:

```
bob: chmod z-rz .
bob: matrix .
d rwv---da:rw-x--d-:rw--y---:-------- bob
```

Now neither `bob` nor `dale` is accessible from the outside. This can be shown as follows:

```
bob: shell
% cd /helios
% ls /ram
/bob    /dale
% access /ram/bob /ram/dale
ls: could not locate /ram/bob - c6040000
ls: could not locate /ram/dale - c6040000
% exit
bob: access .
d rwv---da /00/ram/bob
```

It is necessary to create a new shell because the original shell's capability is now the only access route to `bob`, and the **cd** command would have destroyed it. The new shell does destroy its copy, and thus cannot ever access `bob` again. Only when the new shell is exited and the original shell used can access be regained to `bob`.

So far we have seen how access matrices can be used to control access rights to objects, and how users may protect their directories against outside access. The other side of the coin is to allow users to share information in a controlled way. The mechanism for doing this is for one user to pass the other a capability for the object to be shared.

A capability is normally stored in an internal form by Helios, it may be converted into a printable string by the **refine** command:

```
bob: refine laura
@hmnnmfocamhedhmg/00/ram/bob/laura
```

The result of the **refine** command is a string which starts with an `@` character, encodes the capability in the next 16 characters, and ends with the full name of the object. This may be used almost everywhere a file name may be used and effectively bypasses any access matrices to give direct access to the object. (Note that the string you will get will look different from the one shown above).

The **refine** command gets its name from the fact that it can alter the access rights carried in the capability. To do this it takes an option argument similar to that given to **chmod** except that since it is changing only one set of access rights it needs no row letter (see the *Helios Encyclopaedia* for more details). We can see how this works as follows:

```
bob: access `refine laura`
d rwv---da /00/ram/bob/laura
bob: access `refine =rx laura`
d r--x---- /00/ram/bob/laura
```

In the first command the standard access rights are encoded in the string produced by `refine`. In the second example the access rights are set to `rx`, which is confirmed by the **access** command.

Now that a suitable capability has been manufactured, it is necessary to transfer it to `dale`. Since the directory `dale` is inaccessible, it is not possible to transfer it directly. A simple option would be for user bob to print out or write down the string and pass it to user dale who could then type it in. An equivalent is to put it somewhere they can both access. For the sake of this example we will use a `fifo`.

```
bob: refine =rx laura >/fifo/to.dale
bob: echo "leland did it" >laura/dream
```

Here we generate the encoded capability string and write it out to a fifo. Before leaving bob, we put the information he wants to share into the directory `laura`.

Back with dale, we can see that the information is ready for us by examining the fifo server:

```
dale: ls /fifo
to.dale
dale: cat /fifo/to.dale > from.bob
dale: cat from.bob
@jaapjncioahedhmg/00/ram/bob/laura
```

We start by copying the fifo into a local file. The contents of this file is simply the string produced by refine. It can be used as it stands from the shell:

```
dale: ls `cat from.bob`
dream
```

A more convenient way of using it, however, is to use it to create a **symbolic link**. A symbolic link is nothing more than a capability stored in the file system. It can then be used just like a normal file or directory and access through it is totally transparent to the user.

A symbolic link is created with the **ln** command:

```
dale: ln `cat from.bob` laura
dale: ls
audrey     from.bob     laura@
```

In listings symbolic links are terminated with an `@` character to distinguish them, this character should never actually be typed.

Now, user dale can access directory `laura` in the normal way:

```
dale: access laura
d r--x---- /00/ram/bob/laura
dale: ls laura
dream
dale: access laura/dream
f rw----d- /00/ram/bob/laura/dream
dale: cat laura/dream
leland did it
```

Because bob gave dale a capability with just the `r` and `x` bits set, dale is only allowed to read the directory `laura`, and has `x` access to objects within it. Hence, the access rights to `laura/dream` show only `rw----d-` which is the `x` row of that file's matrix. Like the shell's current directory capability, this symbolic link capability is independent of any subsequent alterations to the access matrix of `laura` by user bob, but bob can still affect dale's rights to objects within the directory `laura` by changing their matrices. For example, bob can stop dale reading `laura/dream` as follows:

```
bob: chmod x= laura/dream
```

This sets the `x` row of the matrix of `laura/dream` to all zeros, so if dale repeats his last command:

```
dale: cat laura/dream
cat: Can't find `laura/dream`
```

So far we have restricted ourselves to using just one of the access classes `v`, `x`, `y` or `z`. However, the access rights allow any combination of these four bits to be set in a capability or a matrix row. It is therefore possible for a matrix row to specify a totally different access class for entries inside the directory to that for the directory itself.

For example, suppose user bob has a change of character and wants all users to be able to both read and write all his files and directories. Rather than go through his entire file space altering the `z` row of all his matrices, he can do this simply by changing the `z` row of his home directory:

```
bob: chmod z=rwy .
bob: matrix .
d rwv---da:rw-x--d-:rw--y---:rw--y--- bob
bob: access /ram/bob/laura
d rw--y--- /00/ram/bob/laura
```

Now the `z` row of `bob` allows read and write access, but it also selects `y` access, not `z`, to objects within the directory. So the access rights to `laura` are now `rw--y---`. Effectively, the `y` matrix row for all objects in bob's file space now controls public access and not the `z` row.

It is also possible to select more than one matrix row, for example:

```
bob: chmod z+x .
bob: matrix .
d rwv---da:rw-x--d-:rw--y---:rw-xy--- bob
bob: access /ram/bob/laura
d rw-xy-d- /00/ram/bob/laura
```

Now, public access to `bob` selects both the `x` and `y` rows from the matrices of subentries. When more than one row is selected, the access rights from each rows are simply combined. In this case public access to `laura` contains `rw-x--d-` from the `x` row and `rw--y---` from the `y` row.

This ends this example session. You will need to reboot your system if you want to use the **/ram** server again since it is irrevocably protected. A more detailed technical description of the protection mechanism may be found in chapter 14, *Protection*.

Chapter 4

CDL

The Component Distribution Language, or CDL, the language which enables you to carry out parallel programming under Helios, is described in this chapter. The purpose of CDL is to provide a high-level approach to parallel programming, where the programmer defines the program components and their relative interconnections and allows Helios to take care of the actual distribution of these components over the available physical resources.

This chapter contains six sections. Section 4.1 describes the underlying model behind the design of CDL. Section 4.2 describes the language syntax and explains how to execute your parallel programs. Section 4.3 provides some detailed examples and programming guidelines. Section 4.4 tells you about CDL farms and how to balance the workload between components. Section 4.5 is devoted to miscellaneous problems and design issues. Appendix B provides additional reference material (the allocation of streams).

4.1 The CSP model for parallel programming

There are a number of different models of parallel programming. Different models may be appropriate for different applications, or may be better suited to different hardware. Communicating Sequential Processes or CSP is possibly the most popular at present: in addition to the Helios CDL language, Transputers in general and the occam language in particular are based on it. The basic idea is very simple. An application is decomposed into a number of smaller parts, or processes. Each process receives data from a source (which is usually another process), does some work on it, and outputs the results to one or more other processes.

Figure 4.1 illustrates such an application, consisting of eight black boxes or 'sequential processes'. Each box obtains data from one or more sources, manipulates this data, and outputs results to one or more other boxes. Parallelism is possible because each of these black boxes can run on a different processor. The boxes interact with each other only through the communication channels.

The occam language implements this scheme at a low level: every black box corresponds to a single Transputer process, and every communication channel corresponds to an occam channel. Some of these channels are actually Transputer links, so that the process is communicating with a process on an adjacent Transputer. Note that the

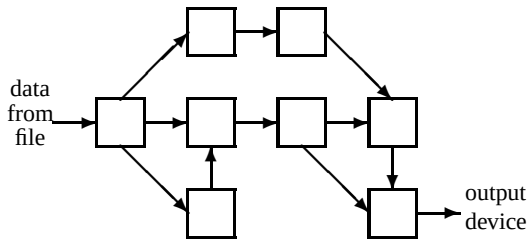


Figure **4.1** CSP

user has to allocate processes to Transputers and specify the interconnecting channels correctly, using the placement facility, which causes problems if the network size or topology changes. If the user is forced to work out the placement, performance is improved. This is because the user will generally have quite a good knowledge of what data will be transferred across the channels. The occam language can be thought of as a low-level approach to parallelism , allowing the user to get optimal performance at the cost of greater programming effort.

The Helios approach is to work at a much higher level. Under Helios every black box is known as a task, and the application as a whole is known as a task force. Each task is a separate program, compiled and linked separately, and quite possibly debugged separately. The individual tasks may be written in any appropriate or preferred language. In a given task force some tasks might be written in C, others in FORTRAN, others still in Pascal or any other language. If the tasks agree on what data to communicate, there is no problem. In theory it is also possible to run different parts of a task force on different types of processors for example, to run all the tasks requiring integer arithmetic only on T414s and all the ones requiring floating point on T800s. Eventually this could be generalised so that one part of the task force runs on, for example, a network of Intel [1] i860s and another part of the task force runs on a network of Transputers. This assumes that there is a communication facility between the two networks which has sufficient bandwidth. Communication between tasks takes place over Unix style pipes, which are set up automatically by the Task Force Manager when the task force is executed. This means that the standard I/O calls can be used for the communication between tasks, and there is no need to add new language constructs to support parallelism.

The purpose of the CDL language is to allow the user to specify a task force. This includes all the component tasks in the task force, the various communication paths between them, for example the pipes to be created, and the particular requirements of individual tasks. The CDL language allows the user to specify task forces of an arbitrary topology. Normally the task force is completely independent of the size and topology of the Transputer network. Helios takes care of mapping the task force onto the available resources. The user can choose to do part or all of the mapping by hand, modifying the CDL script appropriately if the network changes.

[1] Registered trademark of the Intel Corporation

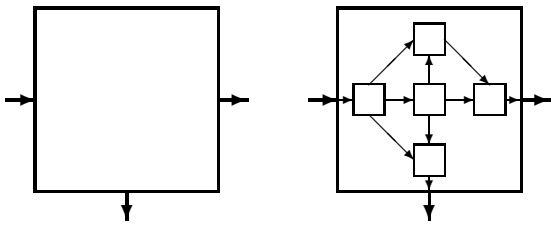


Figure **4.2** Tasks and processes

There is one other point which is worth noting. Consider the two black boxes in Figure 4.2. The one on the left represents an ordinary sequential task with one input and two outputs. The task on the right consists of five separate Transputer processes (Helios threads), all within the same task and hence on the same processor. Both boxes take the same input and produce the same two outputs. As far as the task force as a whole is concerned, the two are indistinguishable. Under Helios, threads can be **Fork()**ed off dynamically, if required.

4.2 The CDL language

This section describes the various constructs available in the CDL language, and how to execute task forces defined using CDL. Like most languages CDL is best taught by example. A typical CDL script might look like this:

```
component master { memory 500000; }

master ( <> slave, <> slave, <> slave, <> slave)
```

This CDL script defines a task force of five tasks. This includes a program called **master** and four invocations of the program called **slave**. The task force is depicted in Figure 4.3. Each task is a component of the task force. In the following discussion, the terms task, program and component are synonymous. The CDL script consists of two parts: the component declaration(s) and the task force definition. The former describes requirements of particular components in the task force. The latter describes the task force as a whole, that is, how the components interact.

A simple introduction to CDL can be found in the *Helios Parallel Programming Tutorial*.[2]

4.2.1 How to execute task forces

The Helios Task Force Manager is the program responsible for mapping and executing task forces. This program is a Helios Server, which obeys the General Server Protocol and hence some of the standard commands can be used on it. For example, `ls /tfm` would list all the task forces currently running, and `rm /tfm/job.6` can be used

[2] Published by Distributed Software Limited.

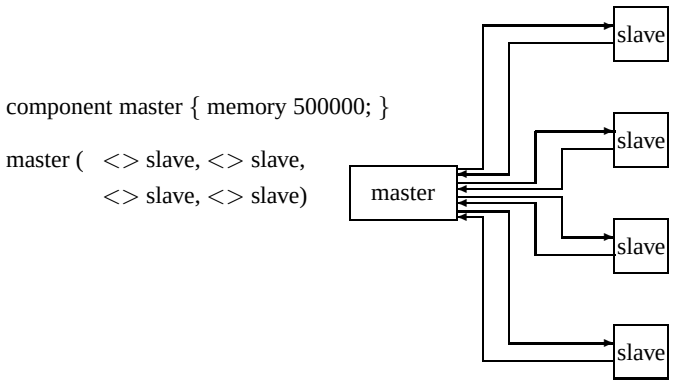


Figure **4.3** A simple task force

to attempt to kill off a task force: whether or not it will succeed depends mostly on the programs making up the task force. There are two ways in which the user can interact with the Task Force Manager to execute new task forces: the shell and the CDL compiler.

There are two types of binary object to consider: programs and compiled task forces. A program is produced by, for example, compiling a C program. A compiled task force can be produced by running the CDL compiler on a CDL script. The Helios shell can operate in two modes: a Unix mode and a CDL mode. In the Unix mode the shell behaves like the Unix C shell, and any commands are executed on the same processor as the shell. These commands must be simple programs, not compiled task forces. To switch from Unix to CDL mode and vice versa the user should use **set cdl** and **unset cdl**. It is possible to have multiple shells running at the same time, each in a separate window, with some shells running in Unix mode and others in CDL mode.

In CDL mode the shell will send all commands, whether programs or compiled task forces, to the Task Force Manager for execution, and in this way the workload is spread over the available network. In addition, in CDL mode the shell understands a subset of the CDL language, which means that programs can be combined using the pipe, subordinate and parallel constructors (but not the interleave constructor). Neither replicators nor component declarations are available, so this facility is limited. However, in the case of a command like

```
cc test.c | asm -p -o test.o
```

the C compiler and the assembler would run on separate processors in parallel, if enough processors were available. Typically such commands would be found in a makefile. To define a non-trivial task force the user should produce a text file, the CDL script, and invoke the CDL compiler on this file. This CDL compiler is an ordinary command like the C compiler or the assembler and takes the following arguments:

cdl <options> <source file> <'compile time' arguments>

with the following options :

[-i] [-l listfile] [-n] [-c] [-o outfile]

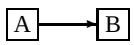
The compiler takes a CDL script as input, defaulting to **stdin** if no source file is specified. Given the **-n** option the compiler will only parse the file, and not compile it. If neither the **-c** nor the **-o** option is given, the compiler will execute the resulting binary immediately. If the **-c** option is given, the compiler will not execute the binary object, but write it to **stdout** instead. If the **-o** option is given, the binary object will be written to the output file specified. The **-i** option is used to make the compiler produce a fully expanded listing of the compiled CDL script, giving details of all the components and the streams on which they communicate. This listing will be sent to the **stderr** stream. The **-l** option is similar, but makes the listing go to the file specified. Please note that all the CDL compiler options must come before the source file, in order to distinguish them from the compile time arguments. If the binary object produced by the CDL compiler is written to a file this file may be executed directly from the shell, if the CDL flag is set in that shell. This can be used to avoid recompiling the CDL script every time you want to run a task force. Many CDL scripts take the following format:

```
#! /helios/bin/cdl

master [10] ||| slave
```

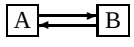
This file can be used as input to the CDL compiler since in CDL lines beginning with a # sign are treated as comments . It can also be used as a shell script. The shell recognises the `#!` sequence at the start of the file and executes the command following it, **/helios/bin/cdl**, using the rest of the file as the standard input to the CDL compiler. No **-o** option is given so the resulting binary will be executed immediately. If the shell's CDL flag has not been set the CDL compiler will run on the same processor as the shell, but the resulting task force will be sent to the Task Force Manager for distributing over the network. If the CDL flag has been set the CDL compiler will itself be executed as a simple task force, running on any suitable processor within the network.

4.2.2 The task force definition

The CDL language defines four parallel constructors: `|`, `<>`, `^^`, `|||`. The pipe constructor | defines a uni-directional pipeline between two programs.

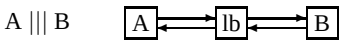
A | B 

The subordinate constructor <> defines a bi-directional pipeline, that is to say, there is a pipe from task A to task B, and another pipe from B to A.

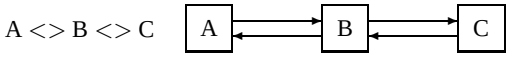
A <> B 

The parallel constructor ^^ defines no communication between the two programs. Of course the two programs may set up a communication channel themselves. For example, one program might write to a file and the other could read the file. Communication pipes could also be set up through the component declarations.

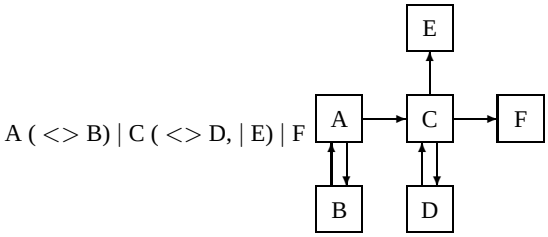
The interleave constructor ||| is used mainly in conjunction with the replicator facility discussed below, to construct farms. It involves the automatic insertion of an additional component: the load balancer.



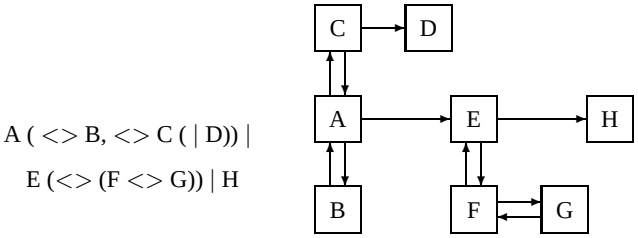
It is possible to combine the CDL constructors to produce more complicated task forces, for example:



It is possible to have branches off the main left to right chain, for example:

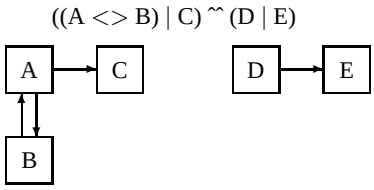


Here `(<> B)` and `(<> D, | E)` are known as auxiliary lists. A more complicated example would be:

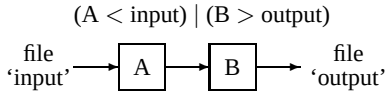


Each constructor has a unique precedence. The order of precedence is `^^ ||| | <>`. `<>` has the highest precedence and `^^` has the lowest. For example: the task force `A <> B | C` is equivalent to `(A <> B) | C` and not `A <> (B | C)`.

Similarly, `A <> B | C ^^ D | E` is equivalent to



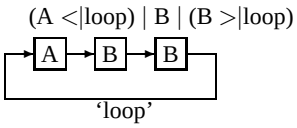
In addition to the task force constructors, it is possible to use Unix style redirection in the task force.



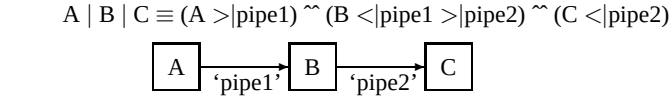
Here component A takes its standard input from a file, and component B writes its standard output to a file. These redirections must not contradict the communication specified by the main definition. For example:

```
(A > output) | (B < input)
```

Here the standard output of A is defined to go to the output file as well as to a pipe, and the standard input of B is defined to come from an input file as well as from a pipe. The CDL compiler will object, because it will think that this is invalid. In addition to the standard redirections $<$ for input, $>$ for output, and $>>$ for output in append mode, the CDL language supports $<|$ for input from named pipe and $>|$ for output from named pipe. For example, the following constitutes a task force in the form of a ring.



Normally when the CDL script specifies a pipe it is unnamed, and the CDL compiler will automatically generate a unique name. However, named pipes can always be used as an alternative. For example, the following two task forces are equivalent.



From the above it should be clear that CDL provides a very powerful way of specifying the parallelism in a task force, and it is fairly easy for users to become confused as to the best way to specify a particular topology. There are a number of ways to proceed. First, the CDL compiler's **-i** option makes it display information about exactly what it has compiled (which may not be what you thought it had compiled). Second, the component declaration part of the CDL script may be used to specify the connecting streams instead of the task force definition.

4.2.3 Allocation of streams

So far this chapter has described how the CDL constructors may be used to combine component programs to give a task force, with pipes connecting the components. This subsection describes how the components can access these pipes.

First consider an ordinary C program. Every C program has three standard streams at the C library level: **stdin**, **stdout**, **stderr**. At the Posix level, these correspond to file descriptors 0, 1, and 2; there are also underlying Helios streams accessible through the **Heliosno()** and **fdstream()** calls, but these are rarely needed by the application program. Other languages may need more or fewer standard file descriptors. For example, FORTRAN has standard streams corresponding to units 5 and 6. CDL allows for up to four standard streams, Posix file descriptors 0–3 with file descriptor 3 not currently used by any language, and will use additional streams from 4 onwards. Now whenever a user runs an application, whether a simple program or a task force, that application inherits an environment from its parent which is usually the shell. This environment includes the application's current directory, some global arguments, some environment strings, and standard streams. These streams usually refer to the current window, which means that if the application reads from **stdin** it expects the user to type something at the keyboard, and if the application writes to **stdout** or **stderr** the data should appear in the window. It is important that task forces can access the window just like ordinary programs, but the question arises as to which component(s) of the task force can do so. Consider the following task forces.

`A | B`

For compatibility with Unix, the standard output of program A goes to the pipe and the standard input for program B comes from the pipe. **Stdin** for A is usually a console stream, as is **stdout** for B and **stderr** for both components.

`A <> B`

Here component B is a subordinate of component A, that is, A is considered to be the senior of the two. Hence **stdin**, **stdout**, and **stderr** for A all correspond to console streams. Additional file descriptors are used for the pipes: A can read from file descriptor 4 to get data from the pipe, and write to file descriptor 5.

File descriptor	C stream	Used for
0	stdin	console I/O
1	stdout	console I/O
2	stderr	console I/O
3	not used in C	
4	undefined	input from pipe
5	undefined	output to pipe

Component B is the junior one, and hence its **stdin** and **stdout** streams do not need to refer to the console. In fact it would be wrong for the **stdin** to refer to the console. This would imply two programs reading from the same window, with possible confusion as to which key presses go where. It is still very useful for component B to have an error stream. Hence the stream allocation for B looks like this:

File descriptor	Used for
0	input from pipe
1	output to pipe
2	error output to console

`A ^^ B`

This defines no communication between the two components. Hence both programs inherit all their standard streams from the environment, which means that **stdin**, **stdout** and **stderr** for both programs will correspond to the console. It is assumed that the application is sensible, and that the two programs do not both try to read the keyboard.

`A <> B <> C`

Again, component A is considered to be the senior one requiring access to the console. Component B now has four pipe streams instead of two. The allocation of streams for all components is:

FD	component A	component B	component C
0	console	input from A	input from B
1	console	output to A	output to B
2	console	console	console
3	unused	unused	
4	input from B	input from C	
5	output to B	output to C	

`A ||| C`

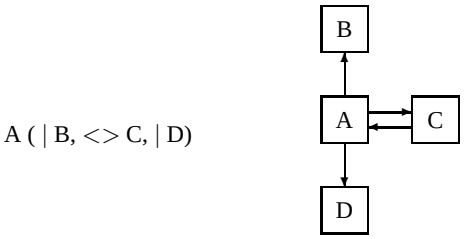
This is equivalent to `A <> B <> C`, with B being the load balancer. The pattern should now be fairly clear. All components have access to **stderr**, file descriptor 2, for debugging output. Additional file descriptors are allocated as

required starting with file descriptor 4. Inputs always correspond to even file descriptors, and outputs correspond to odd file descriptors.

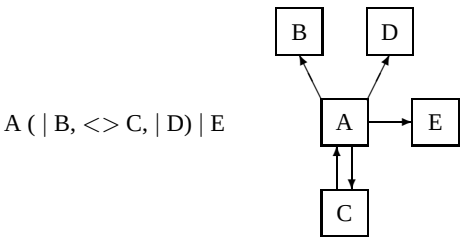
The precedence of constructors affects the allocation of streams. For example, consider the task force `A | B <> C`. The subordinate constructor has a higher precedence so the task force is equivalent to `A | (B <> C)`. When allocating streams the subordinate constructor is handled first, and then the pipe. The complete allocation is:

File descriptor	component A	component B	component C
0	console	input from A	input from B
1	output to B	console	output to B
2	console	console	console
3	unused	unused	unused
4		output to C	
5		input from C	

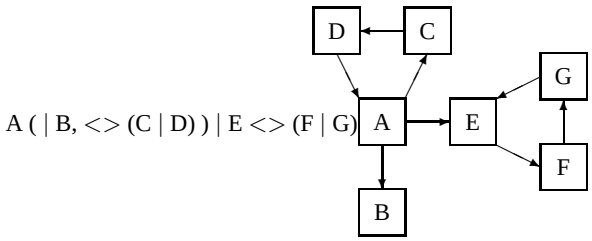
Next we should consider auxiliary lists. Consider the following:



There are two uni-directional pipes from component A to other components, and clearly it is not possible for both to use the standard output of A. To avoid confusion, stream allocation for such auxiliary lists always starts at file descriptor 4, never using the standard output. This allows something like:



The allocation of streams can become complicated. Consider the following:



It may seem unlikely that any real task application would require such a topology, but CDL allows it. However, working out the exact stream allocation for such a task force is difficult and hence the CDL compiler's **-i** option will make it display the standard streams. Appendix B gives the formal rules for stream allocation. As an alternative, it is possible to specify a component's streams in the component declaration part of the CDL script.

4.2.4 Component declarations

The task force definition part of a CDL script specifies the task force as a whole. It is also necessary to specify additional details for particular components, for example to tell the Task Force Manager that a particular component must run on a T800 and not just on any processor. This is the purpose of the component declarations. The following CDL script is a typical example:

```
#
# These are the component declarations
#
component master  { processor T800;  memory 500000; }
component slave   { memory 200000; }
component display { attrib frame_store; }

# this is the task force definition
master ( <> slave, <> slave, <> slave) | display
```

The master program must run on a T800 with at least 500,000 bytes of memory. The slave programs require 200,000 bytes each, and the display program must run on a processor which has been given the user defined **frame_store** attribute in the resource map. Note that all three invocations of the slave program share the slave component declaration. There are six useful fields in a component declaration: **code**, **processor**, **puid**, **attrib**, **memory**, and **streams**. The **code** field specifies the actual program to execute. For example,

```
component master { code /c/usr/bin/mandel; }
```

specifies that the program in file **/c/usr/bin/mandel** should be used for the component **master**. By default the component name must correspond to a program in the current search path. For example: if the shell's current search path (inherited by the CDL compiler through the environment string PATH), is **(/helios/bin .)**, then the CDL compiler

would search through the directory **/helios/bin** and then through the current directory for the program(s) specified.

The processor field may be used to specify the type of processor on which the component can run. For the Transputer version of Helios, the recognised processor types are T800, T414, and ANY. The default is ANY. The **puid** field may be used to specify a particular processor in the network using the full network address, for example `/Cluster/02`. This allows the users themselves to map all or part of the task force. Of course using this facility makes the task force dependent on the network. If the processor changes to `/net/subnetA/02` for example, the CDL script would have to be changed and recompiled. The **attrib** field is entirely under the user's control. It is possible to give processors attributes in the resource map, and to force components onto particular processors by specifying the attributes. For example, if the resource map contains the following two entries:

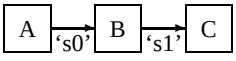
```
terminal 05 { ~03, ~04, ~06, ~07; attrib A1; attrib A2; }
terminal 06 { ~04, ~05, ~07, ~08; attrib A1; }
```

and a CDL script contains the following component declarations:

```
component p1 { attrib A1; }
component p2 { attrib A2; }
```

Component p1 could run on either 05 or 06, but p2 can only run on 05. Depending on the network loading and the rest of the network and task force it is possible that both components would be mapped onto 05, since this solution satisfies the user's specification. The **memory** field may be used to specify the minimum amount of memory that must be available on a processor if it is to run there. The **streams** field may be used to explicitly specify some of the streams on which a component is to communicate, extending the communication set up by the task force definition. For example:

```
component A { streams , >| s0, ; }
component B { streams <| s0, >| s1, ; }
component C { streams <| s1, , ; }

A ^^ B ^^ C
```



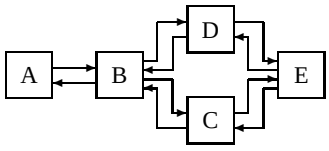
is equivalent to `A | B | C`. Component A has the usual streams for Posix file descriptors 0 and 2, but file descriptor 1 now corresponds to a named pipe. Component B has file descriptors 0 and 1 redefined to be named pipes, but file descriptor 2 remains unchanged. Note that commas are used as place holders. The entries in the streams field must not conflict with the streams specified by the task force definition, for example:

```
component A { streams , >|mystream, ; }

A | B
```

is illegal because file descriptor 1 of component A is used for two separate pipes.

The possible forms of stream redirection are the same as in the task force definition: $>$ for output to a named object, $<$ for input from a named object, $>>$ for appended output, $>|$ for output to a named pipe and $<|$ for input from a named pipe. Any number of streams may be specified. It is not essential to follow the convention of using even file descriptors as inputs and odd file descriptors as outputs, but sticking to this convention may avoid confusion.

Certain task force topologies cannot be defined easily (or at all) using just a task force definition. However, any topology can be specified using a combination of the streams fields in the component declarations and a simplified task force definition. For example,



A possible CDL script for this is:

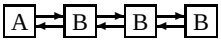
```
component C { streams , , , , <| s0, >| s1; }
component D { streams , , , , <| s2, >| s3; }
component E { streams <|s1, >|s0, <| s3, >| s2; }

(A <> B (<> C, <> D)) ^^ E
```

4.2.5 Replicators

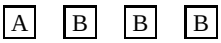
So far this section has described task forces consisting of a small number of components, all different. In practice most task forces consist of a single controller or master task, a number of worker or slave tasks, and possibly some specialised tasks for operations such as graphics I/O. The syntax described so far allows the user to specify a pipeline of perhaps a hundred components, but typing it in is rather tedious. To help the user to specify task forces where a component is repeated, the CDL language provides a facility known as replication. In fact there are two forms: pre-replicators which appear before a constructor and post-replicators which appear after a constructor. The following illustrate pre-replicators.

A [3] | B ≡ A | B | B | B

A [3] <> B ≡ A <> B <> B <> B



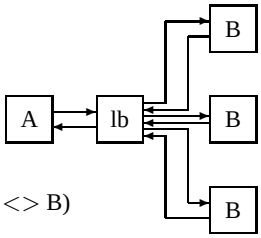
A [3] ^^ B ≡ A ^^ B ^^ B ^^ B



A [3] ||| B

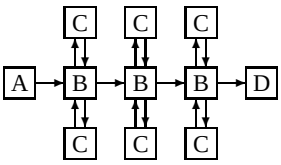
≡

A <> lb 3 (<> B, <> B, <> B)



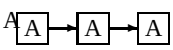
Using replicators with the interleave constructor is particularly interesting. The master component only interacts with the load balancer, and this interaction is independent of the number of slaves. Similarly a given slave component only interacts with the load balancer. This means that the task force can be run with any number of slaves simply by changing the CDL script, without changing the code for the master and slave components. Replicators can be used in more complicated task forces, for example:

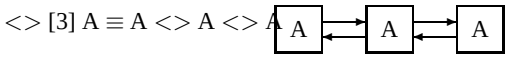
A [3] | (B (,[2] <> C)) | D

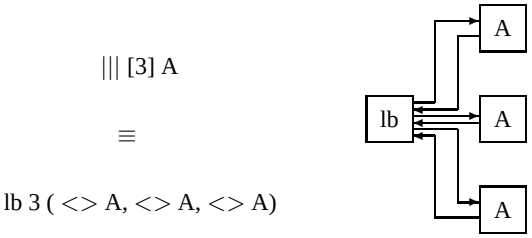


The use of pre-replicators in auxiliary lists is limited in the current release of CDL. If a pre-replicator is required there can be no other components in the auxiliary list and the replicator must be preceded by a comma. Post-replicators can be used without these restrictions. The pre-replicators shown so far, when expanded, define a sub task force as well as the communication for this sub task force. Thus in the definition `A [2] <> B` the replicator expands to a sub task force `B <> B`, and it specifies how this sub task force interacts with the rest of the task force, which means that component A post-replicators define only a sub task force, and not the communication with this sub task force.

| [3] A ≡ A | A | A



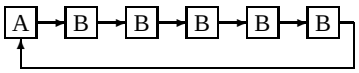




Post-replicators may be used to define exactly the same task forces as pre-replicators.

```
A [3] | B   == A | ( | [3] B)

A [3] <> B  == A <> ( <> [3] B)

A [3] ^^ B  == A ^^ ( ^^ [3] B)

A [3] ||| B == A <> ( ||| [3] B)
```

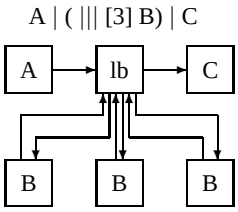
However, many common task force topologies can be described much more easily using post-replicators. For example, consider a ring:



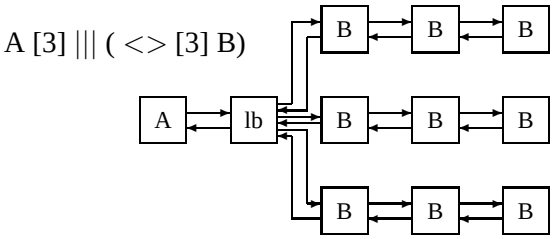
Using a post-replicator this task force can be defined by:

```
A <> ( | [5] B)
```

The sub task force `(| [5] B)` is a simple pipeline, and this pipeline as a whole is a subordinate of component A. Hence the standard input of the pipeline comes from A, and the standard output of the pipeline goes to A. Post-replicators can also be used to define alternative farm topologies:



or even:

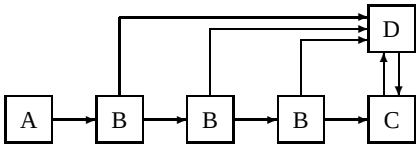


4.2.6 Replicated component declarations

Given a task force definition involving replicators, for example: `A [5] | B`, all five instances of component B would share a single component declaration. This is acceptable in most cases, for example if the purpose of the component declaration is simply to specify the memory requirements or the processor types. However, if the component declaration defines additional streams for the component in question then every component must be defined uniquely. CDL provides a way of specifying subscripts for the replicated components. For example,

```
A [i < 5] | B{i}  == A | B{0} | B{1} | B{2} | B{3} | B{4}
```

Using these subscripts it is possible to define topologies like this:



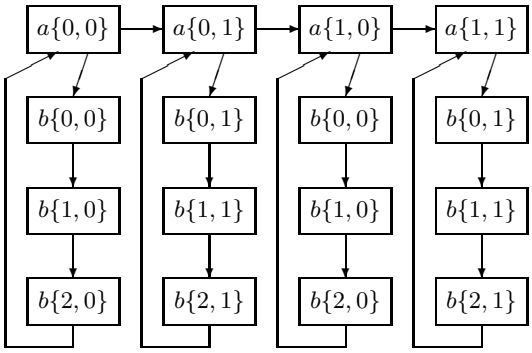

```
component B[i] { streams , , , , ,>| x{i}; }

component D { streams , , , , <| x{0}, , <| x{1}, , <| x{2}; }

A [i < 3] | B{i} | C ( <> D)
```

The sub task force which is being replicated may contain other replicators , and this means that the iteration variable may be defined several times. Replicators are always expanded starting at the innermost level. For example,

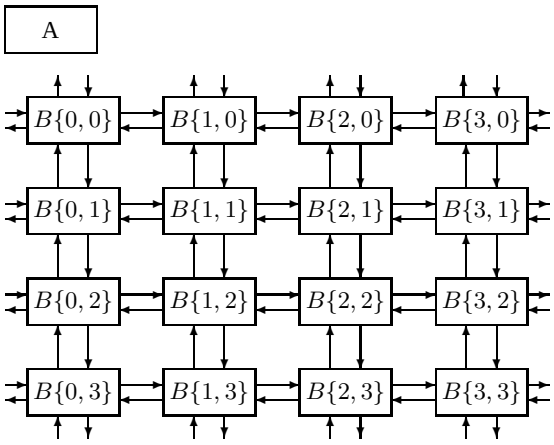
```
| [i<2, j<2] (a{i,j} <> (|[i<3] b{i,j} ) )
```

expands to

```
(a{0,0} <> ( b{0,0} | b{1, 0} | b{2, 0} ) ) |
(a{0,1} <> ( b{0,1} | b{1, 1} | b{2, 1} ) ) |
(a{1,0} <> ( b{0,0} | b{1, 0} | b{2, 0} ) ) |
(a{1,1} <> ( b{0,1} | b{1, 1} | b{2, 1} ) )
```



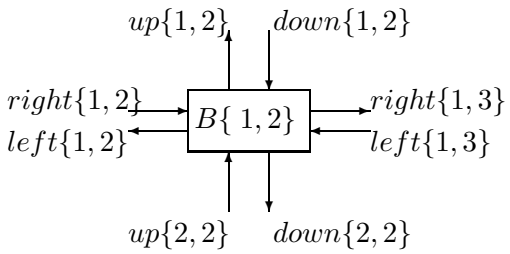
Of course in the above definition there are two components with the name b{0, 0}, two with the name b{1, 0}, and so on. This makes it impossible to use a component declaration for b{i, j} which involves streams. Multi-dimensional indices are permitted, allowing arrays of components. For example, for an image processing or graphics application each component might be responsible for part of the image, and each component would have to interact with its four neighbours to resolve border conditions. Typically there would be a single controller component responsible for actually displaying the final output and interacting with the user. Consider the following topology, which uses wrap around to resolve the edge conditions:



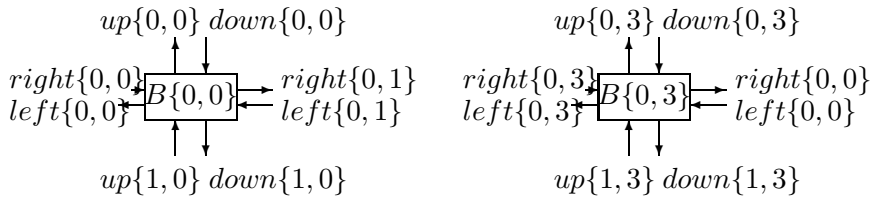
A CDL script which implements this topology is:

```
component B[i, j] {
  streams , , , ,
          <| right{i, j},          >| right{i, (j + 1) % 4},
          <| left{i, (j + 1) %4}, >| left{i, j},
          <| down{i, j},           >| down{(i + 1) % 4, j},
          <| up{(i+1) % 4, j},     >| up{i, j}
 ; }

A (,[i < 4, j < 4] <> B{i,j})
```

For example, component B$\{1, 2\}$ has the following streams:



Use of the remainder operator % is used to provide the desired wrap around.



The expressions used for stream subscripts are written in standard arithmetic format and may contain integers, subscript names, the standard binary operands $+$, $-$, $*$ and %, the unary operands $+$ and $-$, and parentheses.

4.2.7 The environment

Whenever a program or a task force is executed it receives an environment from its parent, usually the shell. This environment contains four types of information: the current directory, the environment strings, the standard streams, and the argument vector. When the Task Force Manager executes the various components of the task force it must send an environment to every component, based partly on the CDL script and partly on the environment sent to the task force as a whole. The current directory and the environment strings are straightforward. Every component of the task force simply inherits these from the parent. Therefore a task force consisting of a single component (the **ls** command) would list the current directory just like the **ls** command itself.

The allocation of streams has already been discussed. For a task force `A <> B`, component A inherits its streams for file descriptors 0, 1, and 2 from the environment

and has additional streams for file descriptors 4 and 5. Component B uses file descriptors 0 and 1 for the pipes, and inherits 2 from the environment. Suppose that this task force is compiled to a binary file called **job**, and the user types the command `job < infile > outfile`. The exact stream allocation would be:

	A	B
0	input from infile	input from pipe
1	output to outfile	output to pipe
2	output to window	output to window
3	not used	
4	input from pipe	
5	output to pipe	

Conversely, if the task force definition was `(A < file1) <> B` and the same command was used as above, the input redirection for infile would have no effect because none of the components inherit standard input from the environment. There is a subtlety if the task force definition contains redirections. For example, if the task force definition is `(A < file1) <> B` then component A takes its standard input from the file `file1` in the directory where the task force was compiled, which is usually (but not always) the directory in which the task force is executed. The arguments passed to the various components in the task force can be divided into three groups: constant arguments, 'compile time' arguments and 'run time' arguments; These are best illustrated by an example. Consider the following CDL script.

```
component A { code xx; }

(A 5 $1) ^^ (B $1 $2 \$1) ^^ (C \$3)
```

Here component A has a constant argument 5 plus the first 'compile time' argument, which means that the CDL compiler will substitute the first 'compile time' argument for every occurrence of the string `$1`. Component B has two 'compile time' arguments and the first 'run time' argument, so the Task Force Manager will substitute the first 'run time' argument for every occurrence of the string `\$1`. Component C is given the third 'run time' argument. Suppose the CDL script is compiled with the command

```
cdl -o test test.cdl xx 123
```

and the resulting binary is executed with the command

```
test 456 yy 789
```

then the arguments passed to all the components will be:

	A	B	C
argv[0]	A	B	C
argv[1]	5	xx	789
argv[2]	xx	123	
argv[3]		456	

Note that the second 'run time' argument is discarded because none of the components use it. Also note that argument zero (conventionally the program name) which is passed to all the components, is in fact the component name. 'Run time' arguments only make sense if the CDL script is compiled to a file and the resulting binary is executed. If the CDL compiler executes the resulting binary immediately there is no way of supplying 'run time' arguments.

4.2.8 Arguments and replicators

When using replicators in a CDL script it is possible to use 'compile time' arguments to specify the size of the task force. For example, given the following task force definition,

```
A [$1] ||| B
```

and the command line,

```
cdl test.cdl 5
```

the CDL compiler would produce a farm with five worker components, and this farm would be executed immediately. Note that with the current release of the software the size of the task force must be specified at 'compile time'. For example the task force definition `A [\$1] ||| B` is illegal because the CDL compiler does not know the size of the task force: `\$1` is a 'run time' argument.

If the CDL script reads:

```
#! /helios/bin/cdl

A [$1] ||| B
```

then it is possible to execute the command `test.cdl 10` to execute a farm with 10 workers. This approach appears to give 'run time' control over the size of the task force, although of course there is an implicit invocation of the CDL compiler When using named replicators it is possible to pass the current value of the replicator to every component. For example,

```
| [i < 3] (B %i) == (B 0) | (B 1) | (B 2)
```

This can be used to tell every component its place within the pipeline. 'Compile time' arguments can also be used in component declarations. For example, consider the two dimensional array of tasks defined earlier. It is possible to change the size of the array both horizontally and vertically at 'compile time', using the following CDL script:

```
component B[i, j] {
  streams , , , ,
      <| right{i, j},           >| right{i, (j + 1) % $1},
      <| left{i, (j + 1) % $1}, >| left{i, j},
      <| down{i, j},            >| down{(i + 1) % $2, j},
      <| up{(i+1) % $2, j},     >| up{i, j}
 ; }

A (,[i < $2, j < $1] <> (B{i,j} %i %j) )
```

If this script is compiled using the command

```
cdl -o test test.cdl 16 8
```

this would produce an array of 128 components, 16 horizontally and 8 vertically, with every component being given its vertical and horizontal offsets.

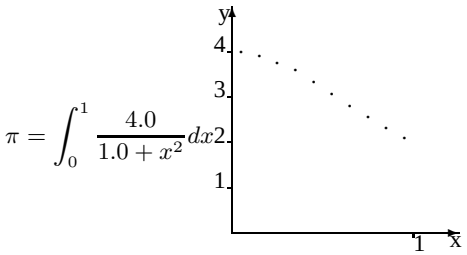


Figure **4.4** The value of π

4.2.9 Signals and termination

To conclude this section, it is necessary to explain what happens in terms of signals and termination of tasks and the task force as a whole. Signals are fairly straightforward. If a signal is sent to the task force as a whole, for example a SIGINT signal if the user presses CTRL-C to abort the task force, this signal is sent to every component in the task force. Unless the application has installed its own signal handling, the system signal handling routines will terminate every component and the task force as a whole will be terminated.

Another important signal to consider is SIGPIPE. This signal is likely to occur if one of the components terminates abruptly and closes its pipes, while other components are attempting to write to these pipes. Reading such a pipe would produce an 'end of file' result. The default handling for a SIGPIPE signal is to terminate the component. Hence if one component terminates abruptly this is likely to cause a chain reaction and terminate other components (possibly the entire task force). This should not be considered a reliable way of terminating a task force.

A task force has terminated when all the components in that task force have terminated, normally or abnormally. The recommended way to cause termination is for the controller task to send a terminate message to the rest of the task force. If all the components exit without an error code, that is, with a return code of 0, then the task force as a whole will give a return code of 0. If any of the components exit with an error code (a value other than 0), or as the result of a signal, then the task force as a whole will exit with an error code.

4.3 An example as easy as PI

4.3.1 A simple problem

The previous section described the CDL language in detail, explaining how it can be used to combine component programs into a task force. This section gives a complete example of parallel programming with CDL, concentrating on how to write the component programs. The example involves estimating the value of π by approximating an integral. (Figure 4.4.)

One way of approximating the area under the curve is to split this area into rectangles and add the areas of the rectangles, as shown in Figure 4.5. In theory the larger the number of rectangles the better the approximation. In practice digital computers have

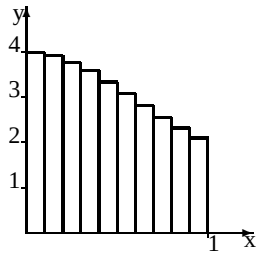


Figure **4.5** Approximating π

a limited precision, with even double precision floating point arithmetic providing only 52 bits of accuracy. Beyond a certain point rounding errors will become more significant than the increased precision obtained from a larger number of intervals. Since this is intended to be an example of parallel programming rather than numerical analysis, such errors will not be considered any further.

4.3.2 How to parallelise the problem

Solving this problem in 'parallel' rather than 'sequential' is quite straightforward. Every interval, that is, every small rectangle which contributes to the total area can be evaluated independently from every other interval. In a fine grained parallel solution every interval could be calculated by a separate component, but that would be pointless. Every component would do a tiny bit of arithmetic, communicate its result, and exit. The cost of loading the components from disc and starting them, the cost of the communication, and of tidying up after the components have exited, would be far greater than the cost of doing the calculation. Instead every component should calculate a large number of intervals. If there are ten processors it would make sense to have ten components, each calculating perhaps 100,000 intervals, giving a total of a million intervals.

The task force can be split into a single controller and a number of workers. The controller is responsible for setting up the workers and for interacting with the user. The workers are responsible for doing the actual arithmetic. It is necessary to consider the possible topologies, and there are two main candidates: a ring and a non-load-balanced farm. These are shown in Figure 4.6. There is little point in having a load-balanced farm since the controller can ensure that all the workers do exactly the same amount of work.

4.3.3 The ring

The CDL script for a ring would be:

```
control <> ( | [$1] worker)
```

Note that the number of workers is determined by a compile time argument. The control component writes to the start of the worker pipeline using file descriptor 5, and reads from the end of the pipeline using file descriptor 4. Each worker reads from file descriptor 0 and writes to file descriptor 1. The first steps are to determine the number

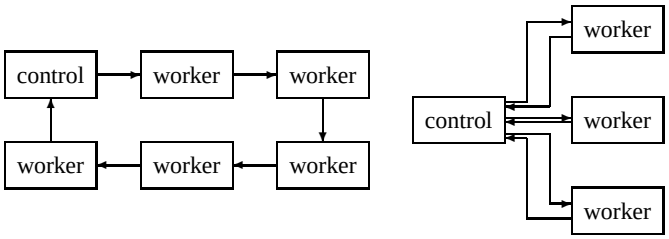


Figure **4.6** Possible topologies

of worker components and to initialise every worker so that it knows its position in the pipeline and the length of the pipeline. One way of doing this is to add arguments to the CDL script and to have every component examine its arguments:

```
CDL script :  (control $1) <> ( | [i<$1] (worker %i $1) )

controller :  int number_workers = atoi(argv[1]);

worker     :  int position       = atoi(argv[1]);
              int number_workers = atoi(argv[2]);
```

An alternative approach involves the components working these things out by communication. This is particularly appropriate if the language has few or no facilities for examining the arguments passed to a program. The controller starts by writing a number 0 to the start of the pipeline. The first worker in the pipeline now knows that its position is 0, adds 1 to this value, and writes it to the next worker. Eventually the last worker writes a value back to the controller, and this value is the length of the pipeline. The controller circulates this value through the pipeline, and every worker is now partially initialised.

```
controller :
    int number_workers = 0;

    write(5, (BYTE *) &number_workers, sizeof(int));
    read( 4, (BYTE *) &number_workers, sizeof(int));

    printf("Pi controller: number of workers is %d.\n",
           number_workers);

    write(5, (BYTE *) &number_workers, sizeof(int));
    read( 4, (BYTE *) &number_workers, sizeof(int));

worker :
    int number_workers, position, temp;

    read( 0, (BYTE *) &position, sizeof(int));
    temp = position + 1;
    write(1, (BYTE *) &temp, sizeof(int));
```

```
    read( 0, (BYTE *) &number_workers, sizeof(int));
    write(1, (BYTE *) &number_workers, sizeof(int));
```

Note that the code above uses Posix I/O calls **read()** and **write()**. The reasons for this, the various alternatives, and doing this I/O in another language such as FORTRAN will be discussed later. The remaining piece of information needed by the controller and all the workers is the number of intervals each worker should calculate. This could be passed as a run-time argument.

```
CDL script : control \$1 <> ( | [$1] worker)

controller : int intervals = atoi(argv[1]);
```

Alternatively the controller could ask the user. The controller's standard streams **stdin**, **stdout**, and **stderr**, are inherited from the environment and hence they should still refer to the console.

```
controller :
    printf("Number of intervals per worker ? ");
    fflush(stdout);
    scanf("%d", &intervals);
```

This information must be sent to every worker.

```
controller :
    write(5, (BYTE *) &intervals, sizeof(int));
    read( 4, (BYTE *) &intervals, sizeof(int));

worker     :
    read( 0, (BYTE *) &intervals, sizeof(int));
    write(1, (BYTE *) &intervals, sizeof(int));
```

Now, consider the fourth worker in a pipeline of 5. The first value read in will have been 3, giving its position in the pipeline, and the worker will have passed 4 to the next one. The second value will have been 5, giving the length of the pipeline. If the user specified 100000 intervals per worker this would have been the third value read. Using these values the worker can determine that it should calculate the areas of 100000 rectangles in the range 0.6 to 0.8. The following code does this:

```
worker     :
    double width, sum, tmp;
    int    first, current, last;

    width = 1.0 / (intervals * number_workers);
    first = position * intervals;
    last  = first + intervals;
    sum   = 0.0;

    for (current = first; current < last; current++)
     { tmp = ((double) current + 0.5) * width;
       sum = sum + width * (4.0 / (1.0 + tmp * tmp));
     }
```

The above worker evaluates intervals 300,000 to 399,999, each of width 0.000002. The first rectangle has a centre point at 0.600001, a height of $(4.0/(1 + 0.600001^2))$ = 2.941173875 ..., and hence an area of 0.000005882.... This is added to the current total. The next rectangle is centred at 0.600003, and so on.

The final problem is how to collect all the partial results produced by the workers and add them together. This can be done by sending a partial sum through the pipeline: the controller sends an initial value 0.0 into the pipeline; every worker reads the current partial sum, adds its result, and sends it to the next worker; finally the controller can read in the result.

```
controller :
    double total = 0.0;

    write(5, (BYTE *) &total, sizeof(double));
    read( 4, (BYTE *) &total, sizeof(double));

worker :
    double total;

    read( 0, (BYTE *) &total, sizeof(double));
    total = total + sum;
    write(1, (BYTE *) &total, sizeof(double));
```

It remains for the controller to print out the value of π, and some statistics. Every interval involves eight floating point operations: conversion from integer to double; adding 0.5; multiplying by width; squaring; adding 1.0; dividing into 4.0; multiplying by width; and a final addition. There are some other floating point operations in every worker, but these are not in the central loop so they can be ignored. Using the number of intervals it is possible to estimate the floating point performance of the task force. Another useful statistic is the proportion of time spent communicating rather than calculating. Putting all this together gives the following two programs.

The controller

```
#include <helios.h>
#include <stdio.h>
#include <posix.h>
#include <nonansi.h>

int main(void)
{ int    number_workers, intervals;
  double total;
  int    comm_start, comm_end, comp_start, comp_end;

  number_workers = 0;
  write(5, (BYTE *) &number_workers, sizeof(int));
  read( 4, (BYTE *) &number_workers, sizeof(int));

  printf("Pi controller: the number of workers is %d.\n",
         number_workers);
```

```
  write(5, (BYTE *) &number_workers, sizeof(int));
  read( 4, (BYTE *) &number_workers, sizeof(int));

  printf("Number of intervals per worker ? ");
  fflush(stdout);
  scanf("%d", &intervals);
  printf("Evaluating a total of %d intervals.\n",
         number_workers * intervals);

  comm_start = _cputime();
  write(5, (BYTE *) &intervals, sizeof(int));
  read( 4, (BYTE *) &intervals, sizeof(int));
  comm_end   = _cputime();

  total = 0.0;
  comp_start = _cputime();
  write(5, (BYTE *) &total, sizeof(double));
  read( 4, (BYTE *) &total, sizeof(double));
  comp_end = _cputime();

  printf("\nCalculated value of pi is %.14f.\n", total);
  printf("Computation time is %.3f seconds.\n",
         ((double)(comp_end - comp_start)) / 100.0);
  printf("Communication time around ring is %.3f seconds.\n",
         ((double) comm_end - comm_start) / 100.0);
  printf("Rating is approximately %d flops.\n", (int)
     ( 100.0 * 8.0 * (double)(number_workers * intervals) /
       (double)(comp_end - comp_start) ) );

  return(0);
}
```

The worker

```
#include <helios.h>
#include <stdio.h>
#include <posix.h>

double eval(int position, int number_workers, int intervals);

int main(void)
{ int    position, number_workers, temp, intervals;
  double sum, total;

        /* get the worker's position in the pipeline */
  read( 0, (BYTE *) &position, sizeof(int));
  temp = position + 1;
  write(1, (BYTE *) &temp, sizeof(int));


        /* get the length of the pipeline */
  read( 0, (BYTE *) &number_workers, sizeof(int));
  write(1, (BYTE *) &number_workers, sizeof(int));
```

```
        /* get the number of intervals per worker */
  read( 0, (BYTE *) &intervals, sizeof(int));
  write(1, (BYTE *) &intervals, sizeof(int));

  sum = eval(position, number_workers, intervals);

  read( 0, (BYTE *) &total, sizeof(double));
  total = total + sum;
  write(1, (BYTE *) &total, sizeof(double));
  return(0);
}

double eval(int position, int number_workers, int intervals)
{ int    first, current, last;
  double width, sum, tmp;

  sum    = 0.0;
  width  = 1.0 / (double) (number_workers * intervals);
  first  = position * intervals;
  last   = first + intervals;

  for (current = first; current < last; current++)
   { tmp = (0.5 + (double) current) * width;
     sum = sum + width * (4.0 / (1.0 + tmp * tmp));
   }
  return(sum);
}
```

4.3.4 A farm topology

For the pi problem an alternative task force topology is a farm, as shown in Figure 4.7. The controller task can ensure that every component receives the same amount of work, so there is no need for a load-balancing component. At the time of writing CDL has no syntax for such a farm, but all the workers can be treated as auxiliaries of the controller giving the following CDL script:

```
CDL script     control (<> worker, <> worker, <> worker)
```

or, using replicators,

```
CDL script     control (, [$1] <> worker)
```

There is no simple way for the controller to determine the number of workers at run-time, so this value should be passed as an argument. In addition, the number of intervals per worker can be passed as a run-time argument.

```
CDL script    control $1 \$1 (,[$1] <> worker)

controller    int number_workers = atoi(argv[1]);
              int intervals      = atoi(argv[2]);
```

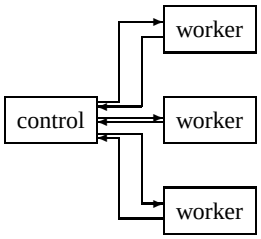


Figure **4.7** A farm topology

As before the controller should now write the initialisation data to all the workers. This can be done as three separate writes: **position**, **number_workers**, and **intervals**. It can be done more efficiently as a single write of a data structure.

```
typedef struct pi_data {
        int position;
        int number_workers;
        int intervals;
} pi_data;
```

Consider the stream allocation in this task force. Every worker is a subordinate of the controller, so it reads from file descriptor 0 and writes to file descriptor 1. All the workers are auxiliaries of the controller, so stream allocation in the controller starts at file descriptor 4. The following macros can be used.

```
controller
    #define to_worker(i)    (5 + i + i)
    #define from_worker(i)  (4 + i + i)
```

The workers can now be initialised by the following code:

```
controller
    pi_data data;
    int     i;

    data.number_workers = number_workers;
    data.intervals      = intervals;

    for (i = 0; i < number_workers; i++)
     { data.position = i;
       write(to_worker(i), (BYTE *) &data, sizeof(pi_data));
     }

worker
    pi_data data;
    double  result;

    read(0, (BYTE *) &data, sizeof(pi_data));
    result = eval(data.position, data.number_workers,
                  data.intervals);
```

Collecting the results is straightforward.

```
controller
    double total, tmp;

    total = 0.0;
    for (i = 0; i < number_workers; i++)
     { read(from_worker(i), (BYTE *) &tmp, sizeof(double));
       total = total + tmp;
     }

worker
    write(1, (BYTE *) &result, sizeof(double));
```

Putting all the above together, we end up with the following two programs.

The controller

```
#include <helios.h>
#include <stdio.h>
#include <posix.h>
#include <stdlib.h>

typedef struct pi_data {
        int    position;
        int    number_workers;
        int    intervals;
} pi_data;

#define to_worker(i)   (5 + i + i)
#define from_worker(i) (4 + i + i)

int main(int argc, char **argv)
{ pi_data  data;
  int      i;
  double   result, temp;
  int      number_workers = atoi(argv[1]);
  int      intervals      = atoi(argv[2]);

  data.number_workers = number_workers;
  data.intervals      = intervals;

  for (i = 0; i < data.number_workers; i++)
   { data.position = i;
     write(to_worker(i), (BYTE *) &data, sizeof(pi_data));
   }

  printf("Pi : evaluating %d intervals on %d workers.\n",
         number_workers * intervals, number_workers);

  result = 0.0;
  for (i = 0; i < number_workers; i++)
   { read(from_worker(i), (BYTE *) &temp, sizeof(double));
```

```
      result = result + temp;
    }

  printf("\nCalculated value of pi is %.14f.\n", result);

  return(0);
}
```

The worker

```
#include <helios.h>
#include <stdio.h>
#include <posix.h>
#include <stdlib.h>

typedef struct pi_data {
        int    position;
        int    number_workers;
        int    intervals;
} pi_data;

double eval(int position, int number_workers, int intervals);

int main(void)
{ pi_data  data;
  double   result;

  read(0, (BYTE *) &data, sizeof(pi_data));

  result = eval(data.position, data.number_workers,
                data.intervals);

  write(1, (BYTE *) &result, sizeof(double));

  return(0);
}

double eval(int position, int number_workers, int intervals)
{ int    first, current, last;
  double width, sum, tmp;

  sum     = 0.0;
  width   = 1.0 / (double) (number_workers * intervals);
  first   = position * intervals;
  last    = first + intervals
  for (current = first; current < last; current++)
   { tmp = (0.5 + (double) current) * width;
     sum = sum + width * (4.0 / (1.0 + tmp * tmp));
   }
  return(sum);
}
```

4.3.5 Different levels of communication

The two versions of the pi task force described above both use Posix-level I/O calls for the communication between components. This subsection describes the different levels of I/O available under Helios. The lowest level of I/O is provided by System library calls:

```
result = Read( Stream *stream, byte *buffer, word amount,
               word timeout);
result = Write(Stream *stream, byte *buffer, word amount,
               word timeout);
```

These functions and the Stream structure are defined in the header file **syslib.h**. All Helios I/O occurs through these routines, directly or indirectly. The routines take a buffer and a buffer size: the contents of the buffer are entirely up to the application. The next level of I/O is the Posix level, using the library calls:

```
result = read( int file_descriptor, byte *buff, word amount);
result = write(int file_descriptor, byte *buff, word amount);
```

Essentially Posix I/O is equivalent to Helios I/O using an infinite timeout. Every Posix file descriptor has an underlying Helios stream. For many applications timeouts are irrelevant and there is no need to use the underlying Helios calls. The highest level of I/O is the language level. In FORTRAN this corresponds to **READ** and **WRITE** statements. The C language has a large number of I/O routines: **printf()**, **scanf()**, **vfprintf()**, **fgetc()**, **fgets()**, **fputc()**, **fputs()**, **gets()**, **puts()**, **fread()**, **fwrite()**, **feof()**, **fflush()**, **setvbuf()**, to name just a few.

There is a very important difference between system and Posix I/O on the one hand, and language-level I/O on the other: the first two are unbuffered; the latter is buffered. If an application uses a Posix **write()** call for 10 bytes, this write takes place immediately. If it is a write to a file the data is sent to the file server immediately: depending on the implementation of the file server, this may store the data in a cache or it may perform some physical disc activity. On the other hand, if the application uses C library calls to transfer the 10 bytes this data would be put into a buffer by the C library: the data would not be sent to the file server until either the buffer had filled up, or the stream was closed possibly because of a program exit, or the application used the **fflush()** call to explicitly flush the buffer. For many applications buffered I/O is extremely desirable because it greatly reduces the number of actual I/O operations that take place: one operation per buffer instead of one operation per piece of data.

Now consider how this affects a simple task force: `A <> B`. Component A writes 50 bytes of data to B, which simply echoes it back. Using C library routines this might be coded as:

```
component A : fwrite(buffer, 1, 50, out_stream);
              fread( buffer, 1, 50, in_stream);

component B : fread( buffer, 1, 50, stdin);
              fwrite(buffer, 1, 50, stdout);
```

This would not work. The **fwrite()** in component A would simply copy the data into a buffer, and the data would not be sent onto component B. Hence component B never receives the data and cannot echo it back. The task force is now in a state of deadlock, because of the buffered I/O. On the other hand, using Posix-level I/O routines:

```
component A : write(5, buffer, 50);
              read( 4, buffer, 50);

component B : read( 0, buffer, 50);
              write(1, buffer, 50);
```

Posix I/O is not buffered, so a **write()** of 50 bytes really does cause the 50 bytes to be transferred immediately, and component B can **read()** the data without problems. For a simple pipeline buffering is not important. Consider:

```
cc test.c | asm -p -o test.o
```

Both the compiler and the assembler use buffered I/O. Hence no data is transferred between the two programs until the C compiler has filled up the buffer, typically with 1K of data. However, the assembler will be waiting patiently suspended on a **read()** until the data is available. There is no feedback from the assembler to the compiler, so there is no possibility of a deadlock. This explains why ordinary utilities can be used in pipelines but not in a more complicated topology.

Some languages provide adequate I/O facilities to allow components to interact correctly, but others do not. FORTRAN I/O is still based around punched cards and line printers, and there are no facilities in the language to manipulate the buffering. Hence FORTRAN components of a task force must always use Posix calls for their I/O. These are illustrated later in this section, using a FORTRAN version of the pi ring example. The C language is much better equipped, and in particular the following calls may be found useful:

```
fflush(FILE *)
```

This routine flushes an output buffer, which means that it causes all the data in the buffer to be sent immediately.

```
setvbuf(FILE *, buffer, mode, size)
setbuf(FILE *, buffer)
```

These routines allow the application to specify the buffering to be used. Note that they may only be used once on a stream, and must be used before any I/O takes place. The mode in **setvbuf()** can be unbuffered, line-buffered, or fully buffered with the specified buffer size. In unbuffered mode every byte is transferred immediately, so that no buffering takes place: this mode is very inefficient for most applications. Line-buffered mode is intended for interaction with terminals, not for interaction with pipes. Fully-buffered mode allows the user to specify the buffer size: if the component always reads or writes data using just one size, this size can be used as the buffer size to give the desired effect.

It should be noted that C I/O always goes through the buffers, which causes a significant overhead compared with Posix or system I/O. In practice it is unusual for a task force component to use anything other than the Posix calls for its interaction with other components.

4.3.6 More about pipe I/O

So far the rules for accessing pipes under Helios have not been discussed. When a component program starts up it receives an environment from its parent, usually the Task Force Manager. Helios Stream structures are set up for all the streams passed in the environment, and Posix file descriptors are allocated to these streams. The library start-up will not actually open the streams: this is delayed until the program actually uses the stream. For example, in a typical task force all components will inherit a stream to the console as the standard error stream, file descriptor 2. However, under normal circumstances this error stream will not be used, so opening the stream as soon as the environment is received is a waste of resources. As far as the application is concerned, this delayed opening is transparent.

All the environment streams will be available at the Posix and Helios levels. In addition, some of them will be opened at the language level. For example, in C the library start-up will initialise C **FILE** structures for **stdin**, **stdout** and **stderr** corresponding to the first three file descriptors. Additional file descriptors passed in the environment will not be opened at the C level: it is likely that they will not be used at the C level, and there is little point in using up the space for the **FILE** structures. Furthermore, the C library has a limit of 20 open C streams at any one time. If an application wishes to use file descriptor 4 as a C stream, for example, the library routine **fdopen()** may be used to convert a Posix file descriptor into a C **FILE** * pointer.

At any one time only two components may access a pipe, one for writing and one for reading. This I/O is synchronous. When an application writes to a pipe it is suspended until the write has completed. When an application reads from a pipe it is suspended until data is available. Note that this is asymmetric. If the writer sends 10 bytes and the reader reads 5, the writer remains suspended until the rest of the data is read but the read returns immediately with the first 5 bytes. If the reader reads 15 bytes, the write completes immediately and the read completes with just 10 bytes: the reader does not remain suspended until all the 15 bytes are available. If it is desired to have asynchronous I/O additional processes can be **Fork()**ed off to perform the I/O.

If the writer closes the pipe, possibly as a side effect of program termination, any further reads on that pipe will return **end-of-file**. However, if the reader has closed the pipe when the writer tries to send data this is an error which will generate a SIGPIPE signal.

4.3.7 Running the task force

There are a few additional points worth discussing regarding this task force. The first involves the problem of distributing the problem over a real network. As it stands, the controller task performs mainly communication rather than calculation. Whilst the pipeline of workers is calculating, the controller is suspended. This means that the controller could run on the same processor as one of the workers, without affecting the performance. Given n free processors in the network, the task force should consist of n workers and one controller.

A problem arises. As far as the Task Force Manager is concerned the controller and the workers are equal: it is not given any information to tell it otherwise. It has the job of putting $(n + 1)$ components on n processors. This means that the Task Force

Manager is as likely to put two workers on the same processor with the controller on a processor by itself, as to put a worker on the same processor as the controller. Having two workers on the same processor will halve the speed of both components, and without a load-balancing component this means that the speed of the task force as a whole is halved.

One solution is to overload the network. If every processor is given at least three workers, the difference in performance between a good mapping and a bad mapping is much less. This approach is particularly suitable in a load-balanced farm. A disadvantage of this approach is that it increases the amount of communication.

An alternative solution is to give Task Force Manager more information about the task force. One way is to explicitly place the controller and one worker on the same processor.

```
component control   { puid /Cluster/00; }
component worker0   { code worker; puid /Cluster/00; }

control <> (worker0 [$1] | worker )
```

Another approach is to use the memory attribute. Suppose that every processor in the network has one megabyte, and this information is supplied to the Task Force Manager using the resource map. If the Task Force Manager is told that every worker needs 600K of memory, it believes that it is impossible to put two workers on the same processor. Hence it would have to put a worker on the same processor as the controller.

```
component worker { memory 600000; }

control <> (| [$1] worker)
```

A third solution is to make the controller do the same amount of work as the workers. This is relatively simple. The controller can install itself as the first worker during the initialisation stage, simply by sending an integer 1 to the start of the pipeline instead of 0. On an n-processor network the task force would now consist of $(n-1)$ workers and a controller, all doing the same amount of work.

During the initial runs the performance of the task force may be disappointing, particularly compared with the official performance rating of a Transputer. The first point to consider is the target processor: the task force performs floating point arithmetic, so if it is compiled for a T414 but runs on T800s the floating point unit will not be used efficiently. If the network is a mixture of T800s and T414 it becomes more difficult. It is possible to have two types of worker binary, one compiled for a T414 and one compiled for a T800, and use both workers in the task force. Clearly it is necessary to specify the processor type in the CDL script. On a network with four T414s and four T800s, a suitable CDL script might be:

```
component worker.t4 { processor T414; }
component worker.t8 { processor T800; }

control <> ( (| [4] worker.t4) | (| [16] worker.t8) )
```

This should run four workers on every T800, using the floating point unit efficiently, and one worker on every T414, hopefully balancing the workload equally between

the two types of processor. Some experimentation may be required to get the balance exactly right.

It should be noted that the official performance statistics for Transputers are based on having both program and data in the fast internal memory. Normally under Helios external memory will be used, and this makes a significant difference to the performance. The amount of internal memory on a Transputer is limited: it may be inadequate even for a program's current data area, and it will certainly be inadequate for the code of a non-trivial program. If it is desired to make use of internal memory, Helios provides the **Accelerate()** and **AccelerateCode()** functions to move the current program stack into internal memory and to place program code into internal memory. The reader is referred to the *Helios Encyclopaedia* and online help system for further details.

4.3.8 FORTRAN task forces

Producing parallel task forces with CDL is independent of the language used to implement the individual components. The same CDL script can be used irrespective of the language(s) used to implement the components, provided that the language provides access to the Posix file descriptors set up by the Task Force Manager. It is possible for the controller to be written in C, and the worker in FORTRAN or Pascal. It is even possible to have the controller in C, some workers in C, some in FORTRAN, and some in Pascal:

```
ccontrol <> ( (| [$1] cworker) |
              (| [$2] fworker) |
              (| [$3] pworker) )
```

The reader is referred to the appropriate language manual for details of the I/O facilities provided. The FORTRAN programs below are written using the Meiko [3] FORTRAN compiler. This provides library routines **POS_WRITE** and **POS_READ** to access the Posix library routines. To perform FORTRAN I/O with the neighbours, units 30 onwards map onto the Posix file descriptors. Thus a **WRITE(31,*)** statement would perform buffered I/O to Posix file descriptor 1. The I/O facilities provided by FORTRAN are inadequate for any task force more complicated than a pipeline, because of the buffering problems already discussed. Meiko FORTRAN provides access to the component's arguments using a **GETPARAMETERS()** function.

The controller

```
      PROGRAM CONTROL

      INTEGER WORKERS, INTERVALS
      DOUBLE  PRECISION TOTAL

      WORKERS = 0
      CALL POS_WRITE( 5, WORKERS, 4)
      CALL POS_READ(  4, WORKERS, 4)
```

[3] Meiko is a trademark of Meiko Limited

```
        WRITE(*, 10) WORKERS
10      FORMAT (' Pi : the number of workers is ', I4 )

        CALL POS_WRITE( 5, WORKERS, 4)
        CALL POS_READ(  4, WORKERS, 4)

        WRITE(*, 20)
20      FORMAT (' Number of intervals per worker ? ')

        READ(*, *) INTERVALS

        CALL POS_WRITE( 5, INTERVALS, 4)
        CALL POS_READ(  4, INTERVALS, 4)

        TOTAL = 0.0
        CALL POS_WRITE( 5, TOTAL, 8)
        CALL POS_READ(  4, TOTAL, 8)

        WRITE(*,30) TOTAL
30      FORMAT (' Calculated value of pi is ', F16.14)

        END
```

The worker

```
        PROGRAM WORKER

        INTEGER WORKERS, INTERVALS, POSITION, TEMP
        DOUBLE  PRECISION TOTAL, SUM
        INTEGER FIRST, CURRENT, LAST
        DOUBLE  PRECISION WIDTH, TMP

        CALL POS_READ(  0, POSITION, 4)
        TEMP = POSITION + 1
        CALL POS_WRITE( 1, TEMP, 4)

        CALL POS_READ(  0, WORKERS, 4)
        CALL POS_WRITE( 1, WORKERS, 4)

        CALL POS_READ(  0, INTERVALS, 4)
        CALL POS_WRITE( 1, INTERVALS, 4)

        SUM = 0.0
        WIDTH = 1.0D0 / (WORKERS * INTERVALS)
        FIRST = POSITION * INTERVALS
        LAST  = FIRST + INTERVALS

        DO 100 CURRENT = FIRST,LAST-1,1
        TMP   = (CURRENT + 0.5D0) * WIDTH
        SUM   = SUM + WIDTH * (4.0D0 / (1.0D0 + TMP * TMP))
100     CONTINUE
```

```
        CALL POS_READ(  0, TOTAL, 8)
        TOTAL = TOTAL + SUM
        CALL POS_WRITE( 1, TOTAL, 8)

        END
```

4.3.9 Pascal task forces

To end this section, here are the same task force components written in Prospero Pascal. For more details of the facilities used, please refer to the Pascal manual.

The controller

```
PROGRAM control(input, out);

FUNCTION _read(hand:integer; place:integer; amount:integer):
        integer;EXTERNAL;
FUNCTION _write(hand:integer;place:integer;amount:integer):
        integer;EXTERNAL;

VAR     number_workers, intervals, junk:integer;
        total:longreal;

BEGIN
        number_workers := 0;
        junk := _write(5, addr(number_workers), 4);
        junk := _read( 4, addr(number_workers), 4);

        writeln('Pi : the number of workers is ',
                 number_workers);

        junk := _write(5, addr(number_workers), 4);
        junk := _read( 4, addr(number_workers), 4);

        write('Number of intervals per worker ? ');
        readln(intervals);
        write('Evaluating a total of ');
        writeln(number_workers * intervals,' intervals');

        junk := _write(5, addr(intervals), 4);
        junk :=_read( 4, addr(intervals), 4);

        total := 0.0D0;
        junk  := _write(5, addr(total), 8);
        junk  := _read( 4, addr(total), 8);

        writeln('Calculated value of pi is ', total:16:14);
END.
```

The worker

```
PROGRAM worker(input, out);
```

```
FUNCTION _read(hand:integer; place:integer; amount:integer):
              integer;EXTERNAL;
FUNCTION _write(hand:integer;place:integer;amount:integer):
              integer;EXTERNAL;
         { the evaluation routine }
FUNCTION eval(position, workers, intervals : integer):
         longreal;
VAR
        first, current, last : integer;
        width, sum, tmp : longreal;

BEGIN
        sum     := 0.0D0;
        width   := 1.0D0 / (workers * intervals);
        first   := position * intervals;
        last    := first + intervals;

        for current := first to (last - 1) do
        BEGIN
             tmp := (0.5D0 + current) * width;
             sum := sum +
                    width * (4.0D0 / (1.0D0 + tmp * tmp));
        END;

        eval := sum;
END;
        { the main routine }
VAR     position, number_workers, intervals,
        junk, temp:integer;
        total, sum:longreal;

BEGIN
        junk := _read( 0, addr(position), 4);
        temp := position + 1;
        junk := _write(1, addr(temp), 4);

        junk := _read( 0, addr(number_workers), 4);
        junk := _write(1, addr(number_workers), 4);

        junk := _read( 0, addr(intervals), 4);
        junk := _write(1, addr(intervals), 4);

        sum := eval(position, number_workers, intervals);

        junk  := _read( 0, addr(total), 8);
        total := total + sum;
        junk  := _write(1, addr(total), 8);
END.
```

4.4 CDL farms and load balancing

In the pi example described in the previous section the task force had no need of a load balancing component The problem could be divided into a number of smaller jobs, all requiring the same amount of CPU time. Every worker could be given one of the jobs, thus ensuring that all the workers and hence all the processors in the network were kept busy.

Many problems can be parallelised without load balancing, like the pi example. Other problems are not suitable for simple load balancing, particularly if the individual jobs are not independent and the workers need to communicate with each other. Nevertheless, for many applications and under many circumstances a load balancing component is essential or very desirable. Consider a ray tracing program. One pixel might just display empty space, while an adjacent pixel might involve rays bouncing off a hundred objects. There is no way of predicting in advance how much work might be required for a given pixel. Another problem occurs if the network is heterogeneous or shared. If a task force involves floating point arithmetic and the network has a mixture of T414s and T800s, the latter should do most of the work. In a multi-user environment with a shared pool of perhaps 64 processors, one user might start a 64 component task force whilst a neighbour starts up 16 components. The resulting performance will be unpredictable, probably unrepeatable, and almost certainly disappointing unless load balancing is being used.

One other advantage of load balanced task forces should be mentioned at this point. They tend to be very easy to write, because the chore of distributing the workload amongst an arbitrary number of workers is handled by a standard component, the load balancer.

This section introduces the reader to programming farms, the main task force topology which uses load balancing. It starts with a description of the communication between the master and worker components and the load balancer, with an example task force. Next the inner workings of a simple load balancing component are described, in addition to ways in which this program might be modified for particular applications. The sources of this load balancer are shipped with Helios.

4.4.1 A simple farm

The CDL syntax for a simple farm is

```
master [3] ||| worker
```

as shown in Figure 4.8.

The master components send jobs, in the form of packets, to the load balancer. A packet contains all the information needed to perform the required calculations. The load balancer reads in packets from the master, checks whether any of the slaves are free, and if so the packet is sent to that slave. All of the slave components read in one job packet, process it, send back a reply packet, and read in the next job. The load balancer continuously reads reply packets from the workers and sends them back to the master.

This scheme provides load balancing without needing any information about the application. If the task force has been mapped so that two workers are mapped onto

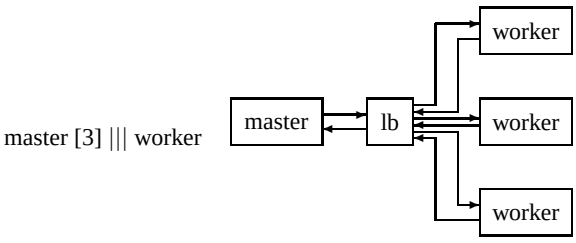


Figure **4.8** A simple farm

one processor, and a third worker has a processor to itself, the first two will work at half speed. The third worker will be able to process packets twice as fast, so it will receive twice as many packets as each of the other two. Therefore it receives the same number of packets as the other two together, and the two processors both handle exactly the same number of packets and are kept equally busy.

What exactly is a packet? In general both job packets and reply packets can contain an arbitrary amount of data, to be read in by the other side. However, the amount of data must be known by the other side before it can be read in. The solution is to split the packet into two parts: a fixed size header, and a variable size data field whose size is contained in the header.

```
typedef struct LB_HEADER {
        word   size;
        word   control;
} LB_HEADER;
```

First, consider the slave. This component must execute a loop, as described below.

```
forever
       read packet header
       allocate space for data, if necessary
       read rest of data
       perform calculation
       construct reply packet header
       send packet header and packet data
```

The master is slightly more complicated. It must send out enough packets to keep all the workers busy, but not so many that the load balancer runs out of memory in trying to buffer them. In addition it must read back reply packets from the load balancer as quickly as possible. The simplest way to achieve this is to have two separate processes, one generating job packets as quickly as the load balancer will accept them, the other reading reply packets from the load balancer as quickly as they are produced (see Figure 4.9).

```
PAR
 writer : loop
              generate job
              initialise packet header
              send packet to load balancer
```

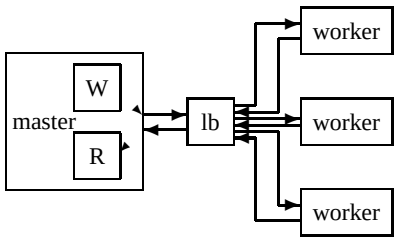


Figure **4.9** Two processes

```
reader : loop
             read packet header
             allocate space for data, if necessary
             read packet data
             process the result, (write it to a file)
```

Now consider how to apply this to a particular problem. Given a range of numbers a–b where $1,000,000,000 < a < b < 2,000,000,000$, find the integer in this range with the largest number of factors. The range of numbers will be at least 100,000. For a number x, the number of factors can be evaluated by the following code:

```
worker :     int root = square_root(x);
             int number_factors = 2;

             for (i = 2; i <= root; i++)
              if (x % i == 0)
               number_factors += 2;

              /* do not count an exact root twice */
             if (root * root == x)
               number_factors -= 1;
```

Consider the size of a job very carefully. The above loop will be executed at most 44721 times, since $\sqrt{2,000,000,000} = 44721$. Therefore the amount of work per number is small. To get the right balance between the computation time per packet and the communication overheads, every job should be a set of numbers, perhaps 100, giving at least 1000 packets in total.

What exactly should the packets contain? A job packet sent by the master to the workers must contain the set of 100 numbers to be handled as part of that job. This can be done by sending the first number in that set. The following code can be used.

```
master, writer process :

        typedef struct job_data {
                LB_HEADER header;
                int       start;
         } job_data;
```

```
        job_data data;
        int      i;

        data.header.control = 0;
        data.header.size    = sizeof(int);

        for (i = base; i < end; i+= 100)
         { data.start = i;
           write(5, (BYTE *) &data, sizeof(job_data));
         }
```

The load balancer is a subordinate of the master, so file descriptors 4 and 5 are used for communication. The job packet is sent as a single 12 byte structure. In fact the load balancer will read in the header so that it knows the amount of data to come (in this case another 4 bytes), and then it will read in this data. Because of the way the pipe protocols work this does not cause any confusion. The **write()** will be suspended until the load balancer has read in all 12 bytes.

Next, consider the worker. Every worker is a subordinate of the load balancer, so file descriptors 0 and 1 are used for communication.

```
worker :
          job_data  data;

          read(0, (BYTE *) &data, sizeof(job_data));
```

Strictly speaking the worker should read in the packet header and then the data. In practice the load balancer always writes packets all at once. The final result that the task force computes is the number with the greatest number of factors. Therefore the reply packet returned by the worker must contain the number within its current job with the greatest number of factors, together with the count. The master can compare this with the best result to date.

```
master, reader :

        typedef struct reply_data {
                LB_HEADER header;
                int       best;
                int       count;
        } reply_data;

        reply_data data;
        int i, best, count = -1;

        for (i = base; i < end; i += 100)
         { read( 4, (BYTE *) &data, sizeof(reply_data));
           if (data.count > count)
            { best = data.best; count = data.count; }
         }

        printf("The winner, with a score of %d, is %d.\n",
               count, best);
```

Note that the reader process reads exactly the same number of packets as the writer process sends out. For less trivial applications the two processes might synchronise, so that the writer does not start the next run of jobs until the current run has been completed.

Neither the load balancer nor the workers have any way of knowing when the last data packet has been sent. Therefore these programs will not exit, and the task force as a whole will not exit. To avoid this the master component should send a special 'terminate' packet, which the load balancer will broadcast to all workers before exiting. Every worker should check every packet to see if it is this special terminate packet, and if so the worker should exit.

Combining all the above, we get the following:

The CDL script

```
master \$1 \$2 [$1] ||| worker
```

The master

```
#include <helios.h>
#include <stdio.h>
#include <stdlib.h>
#include <posix.h>
#include <lb.h>
#include <sem.h>
#include <nonansi.h>

typedef struct job_data {
        LB_HEADER header;
        int       start;
} job_data;

typedef struct reply_data {
        LB_HEADER header;
        int       best;
        int       count;
} reply_data;

int base, end;
Semaphore finished;

static void reader_process(void);
static void writer_process(void);

int main(int argc, char **argv)
{ LB_HEADER terminate;

  base = atoi(argv[1]);
  end  = atoi(argv[2]);

  InitSemaphore(&finished, 0);
```

```
  unless(Fork(2000, &reader_process, 0))
   { fprintf(stderr, "Unable to fork off reader process.\n");
     exit(1);
   }

  writer_process();  /* send all the job packets */

  Wait(&finished);   /* signalled by the reader process */

  terminate.control = LB_MASTER + Fn_Terminate;
  terminate.size    = 0;
  write(5, (BYTE *) &terminate, sizeof(LB_HEADER));
  return(0);
}

static void writer_process(void)
{ job_data data;
  int      i;

  data.header.control = 0;
  data.header.size    = sizeof(int);
  for (i = base; i < end; i+= 100)
   { data.start = i;
     write(5, (BYTE *) &data, sizeof(job_data));
   }
}

static void reader_process(void)
{ reply_data data;
  int i, best, count = -1;

  for (i = base; i < end; i+= 100)
   { read(4, (BYTE *) &data, sizeof(reply_data));
     if (data.count > count)
      { best = data.best; count = data.count; }
   }

  printf("The winner, with a score of %d, is %d.\n",
         count, best);

  Signal(&finished);
}
```

The worker

```
#include <helios.h>
#include <stdio.h>
#include <stdlib.h>
#include <posix.h>
#include <lb.h>
#include <sem.h>
#include <nonansi.h>
typedef struct job_data {
```

```
        LB_HEADER header;
        int       start;
} job_data;

typedef struct reply_data {
        LB_HEADER header;
        int       best;
        int       count;
} reply_data;

static void process_job(job_data *, reply_data *);
static int  square_root(int);

int main(void)
{ job_data   job;
  reply_data reply;

  forever
   { read(0, (BYTE *) &job, sizeof(job_data));
     if ((job.header.control & LB_FN) == Fn_Terminate)
      exit(0);

     process_job(&job, &reply);
     reply.header.control = 0;
     reply.header.size    = sizeof(reply_data) -
                            sizeof(LB_HEADER);
     write(1, (BYTE *) &reply, sizeof(reply_data));
   }

  return(0);
}
static void process_job(job_data *job, reply_data *reply)
{ int x, i, root, number_factors;

  reply->count = -1;

  for (x = job->start; x < job->start + 100; x++)
   { number_factors = 2;
     root = square_root(x);

     for (i = 2; i <= root; i++)
      if (x % i == 0)
       number_factors += 2;

     if (root * root == x)
      number_factors--;

     if (number_factors > reply->count)
      { reply->count = number_factors;
        reply->best  = x;
      }
   }
}
```

```
    /* evaluate a square root without using floating point */
    /* five iterations of the Newton-Raphson method with a */
    /* starting point of sqrt(1,500,000,000) will suffice  */
static int square_root(int x)
{ int estimate = 38730, i;

  for (i = 0; i < 5; i++)
   estimate = (estimate + (x / estimate)) / 2;

  return(estimate);
}
```

4.4.2 A simple load balancer

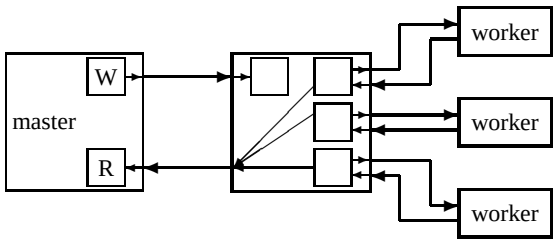


Figure **4.10** Processes in the load balancer

The previous subsection gave an example of a simple farm. However, to make efficient use of farms it is necessary to understand the inner workings of the load balancing component, and possibly to modify the program to suit the application. The sources of the load balancer are shipped with Helios.

At any one time the load balancer must be ready to read in new job packets from the master, and reply packets from some or all of the workers. Simultaneous inputs from different sources can be handled conveniently by separate processes (see Figure 4.10).

A worker component goes through the following loop.

```
forever
       wait for a job packet to be sent
       read the packet
       process the job
       send the reply packet
```

It is convenient for the load balancer processes interacting with the workers to use a similar loop.

```
forever
       wait for a job packet to be available
       send the packet to the worker
       wait for and read the reply packet
       pass the reply on to the master
```

It is a good idea to have an input buffer of new job packets, waiting to be sent to the workers. If a worker finishes its current job a new one will be available immediately to be sent to that worker. Assuming that the master can generate job packets much faster than a worker can process a packet, the largest buffer size that makes sense is one packet per worker. Even if all the workers were to finish their current job at the same time they could all be sent a new job immediately. The master should then have enough time to refill the buffer.

If the master component cannot generate jobs fast enough to keep the workers busy, the farm is unbalanced and some of the workers will be idle, wasting valuable processors. This is particularly important in a multi-user environment. Suppose that a problem involves n packets, to be evaluated on x workers. Every worker can handle y packets per second, and the master can generate z packets per second. The workers can process a total of $(x * y)$ packets per second, so for a balanced farm we must have: $z >= (x * y)$, or $x <= (z/y)$. The values of y and z can be estimated by using some simple tests, giving an approximate idea of the number of workers that can be used sensibly. If too many workers are used then the master component is a bottleneck, and some of the workers are idle.

It is possible that the rate at which jobs can be generated varies as the run proceeds. For parts of the run the master can generate packets much faster than they can be processed, whilst for other parts of the run the master takes too much time. Under these circumstances it makes sense to increase the size of the input buffer in the load balancer to a larger number than the number of workers in the task force.

Another possible bottleneck arises if the master component cannot read back the reply packets as quickly as they are generated. Again, if x is the number of workers, y the rate at which workers can generate reply packets, and z the rate at which the master component can read reply packets, the largest farm that makes sense is: $x = z/y$. If the rate z varies, it may make sense to have some buffering for the reply packets. This buffer can be incorporated into the master component, as shown in Figure 4.11.

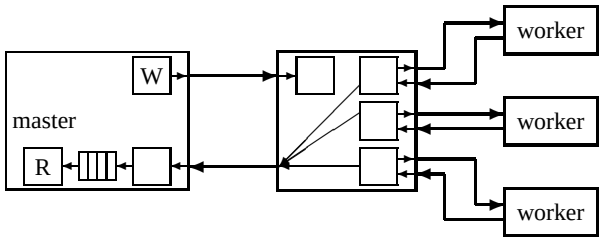


Figure **4.11** Buffering in the master

Consider the load balancer in this scheme. With x worker components, the load balancer contains $(n + 1)$ processes. All of these access the input table, and all but one access the single stream back to the master. These are shared resources which must be protected by semaphores **table_lock** and **master_lock**. In addition, it is useful to have two counting semaphores controlling the table usage. The process reading from the master is suspended automatically if there is no space left in the input table, and this in turn suspends the writer process in the master component when it tries to send its

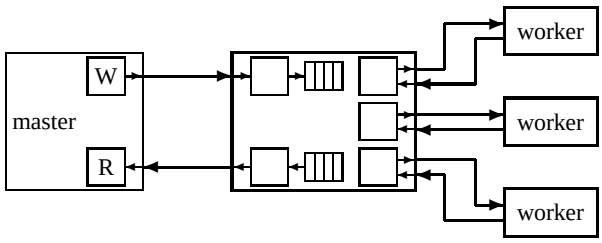


Figure **4.12** Buffering in the load balancer

next job packet. The processes interacting with the workers are suspended if there are no new jobs in the input table, leaving the worker components idle.

In the load balancer all the **interact_with_worker()** processes must send the reply packets to the master themselves. This means that these processes are performing pipe I/O, or even that these processes are suspended on a semaphore waiting to do pipe I/O, whilst the corresponding worker components are idle waiting for their next job packets. Depending on how quickly the master reads the reply packets, this may or may not affect performance significantly. One solution is to add an output buffer to the load balancer, matching the input buffer, as shown in Figure 4.12.

Given this output buffer, it is a relatively small change to add some new control packets which the master component can send to modify the load balancer's behaviour. One control packet could change the size of the input table, reducing it if the load balancer runs out of memory in buffering too many packets, or increasing it to allow for variations in the rate at which job packets can be produced. Another control packet could change the size of the output table, increasing it to allow for variations in the rate at which reply packets are handled. This eliminates any need for buffering in the master component. A very intelligent load balancer could monitor the rates at which the master produced job packets and collected reply packets, adjusting the buffer sizes as required.

This brings us to the concept of control packets generally. The simple load balancer only supports two special control packets: 'terminate' and 'broadcast'. The 'terminate' packet requires no additional data. It is implemented simply as a broadcast of the terminate packet to all the workers, followed by the load balancer exiting. The workers are not expected to send a reply to the terminate packet. A broadcast packet can be used to send an arbitrary amount of data to all the workers. For example, in a ray tracing application all the workers must be initialised with details of all the objects in the picture being ray traced. This is done as a simple broadcast, with details of all the objects held in the data vector. The worker component should recognise this special broadcast packet and handle it as appropriate. While a task force is running it may be necessary to send many broadcasts. For example, to raytrace a completely different picture, or to raytrace the current picture from a different angle.

Broadcasting a packet involves synchronisation. The broadcast cannot be sent until all the previous job packets have been handled, because these must be processed with the old broadcast data. This can be implemented by waiting for all the workers to be idle and the table to be empty. A special broadcast packet is inserted into the table.

Whenever a process such as **interact_with_worker()** detects this packet it suspends itself until it is reactivated by the process interacting with the master. This is implemented using the **broadcast_master** and **broadcast_slave** semaphores. Broadcasting could be implemented more efficiently but with considerable effort, by sending the broadcast packet and the new jobs to a worker as soon as possible.

Many other control packets are possible, and it may well be worthwhile to add these to the load balancer if it makes the other components easier to write. Some of the possibilities that spring to mind are as follows.

sync packet

This is returned to the master when all the current jobs have been finished, leaving the workers idle. It can be used to inform the reader process in the master component that the end of the run has been reached.

worker count

The load balancer sends a reply packet indicating the number of workers in the farm.

to_worker

This could be sent by the master to direct the packet to one particular worker, rather than to the next free worker.

set_input_buffer

This could be used by the master component to control the buffering in the load balancer.

set_output_buffer

This is similar to **set_input_buffer**.

job_packet_size

This tells the load balancer about the size of job packets, so that it does not need to do any dynamic allocation every time a packet arrives. A special number, -1 for example, could be used to reset the load balancer to packets of variable length.

reply_packet_size

This is a similar facility for the reply packets.

Users can implement any control packets they choose. The LB_HEADER structure contains a 32 bit control field, whose interpretation is entirely up to the components. There is no need to follow the current encoding scheme, if this seems inappropriate. Different applications have different requirements for the load balancing component, and the program is intended merely as a basis on which users can build.

4.4.3 More about packets

When designing a task force with a farm topology, the critical design decision will be the nature of the job packets. There is a very important relationship between the cost of small packets and the need for a large number of packets. Again, consider a ray tracing application, with a picture size of $512 * 512$ pixels. One approach would be to make every pixel a separate job, thus generating 262,144 jobs. Each job involves 4 lots of pipe I/O: from master to load balancer, from load balancer to worker, from worker back to load balancer and from load balancer back to master. Therefore if every pixel is a separate job, over a million lots of pipe I/O is involved, and this will be a lengthy process.

Another approach would be to have every scanline as a separate job, giving 512 jobs. If the number of workers is fairly small, perhaps 10, every worker would have to process about 50 jobs which should allow for sensible load balancing. Conversely, with 100 workers each worker would have only 5 packets. Under the worst circumstances, calculation on the 512^{th} scanline would start just as scanlines 412–511 have been handled, leaving 99 workers idle while the other one is processing scanline 512. These worst circumstances can arise irrespective of the size of the job and the number of workers, but they become more serious as the size of each job increases.

It is not possible to give any rules for the amount of work per packet compared with the amount of communication and the number of workers. In general, it should be easy to experiment and produce a good compromise given a particular application and processor network. In the ray tracing example, it should be fairly easy to measure the performance for jobs consisting of 1, 4, 16, 64, 128, 256 and 512 pixels with little change to the programs. It is likely that the number of workers that can be used sensibly is small, as low as 5 or 10 perhaps (even on a 1024 processor network), because the master component is a bottleneck. Of course, if the optimal solution is to have 14 workers, a load balancer, and the master, then it is possible to run 64 of these task forces simultaneously on the 1024 processor network and get optimal performance from the entire network.

One other point must be noted. In general, the packets returned to the master will not be in the same order as the packets sent out by the master, because some of the jobs may require more time than others or because some of the workers can operate faster than others. Therefore the reply packet must always contain some information which allows the master component to identify it. This might be a sequence number, or it might be the screen coordinates of the pixels calculated during the current job.

4.4.4 Advanced farms

So far this section has discussed simple farms, consisting of one master component, one load balancer, and an arbitrary number of workers. For certain applications other topologies might be appropriate. For example, it may be desirable to have more than one load balancer, as shown in Figure 4.13.

This might be useful if the packet size is large, and a single load balancer is unable to buffer the nine packets in a simple farm. Of course the above topology forces the master component to distribute the jobs equally between the three load balancers. An alternative solution is to have a hierarchy of load balancers (see Figure 4.14).

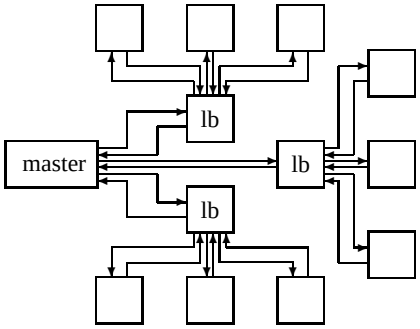


Figure **4.13** Multiple load balancers

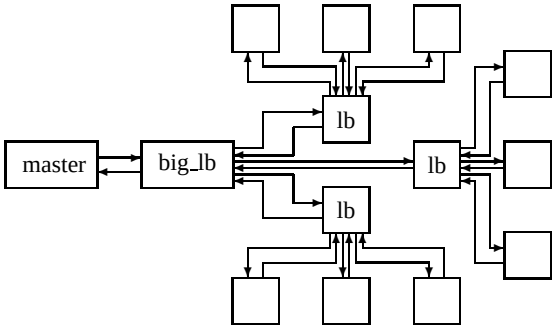


Figure **4.14** Hierarchy of load balancers

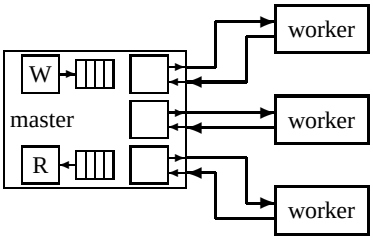


Figure **4.15** Built-in load balancer

A different version of the load balancer would be required in the middle. It would have to know that its three subordinate load balancers all had three workers, and hence that they could all receive and process three job packets immediately and buffer another three. It would have to take care not to fill up the buffer in a subordinate load balancer if the workers of another subordinate load balancer were idle. Note that the middle load balancer is given two arguments: the number of subordinate load balancers and the number of workers per subordinate. Therefore it is possible to produce a single load balancer which could operate in either position, simply by examining the number of arguments. Also note that exactly the same worker program can be used with all these configurations.

One more possibility should be considered. In a simple farm all packets involve four lots of pipe I/O as discussed earlier. This can be reduced to just two lots of pipe I/O by merging the master component and the load balancer. The writer process in the master, instead of sending a packet to the load balancer, now adds it to the input table. Similarly, the reader process in the master, instead of reading packets from the load balancer, takes them from the end of the output table. This creates the scheme shown in Figure 4.15.

The worth of the performance gain with this merger depends on the application. If the cost of communication is higher than the cost of computation, the merger may make the parallelisation worthwhile. If the cost of computation is much larger, the gains are not worthwhile.

4.5 Odds and ends

The reader can now produce task forces by using the information and the examples in the previous sections. However, there are a few points left that should be discussed. Firstly, there is a more detailed description of the relationship between computation and communication, together with the problems of bottlenecks. Secondly, the possible problems encountered with workers are mentioned. Thirdly, the possibility of producing a parallel server is discussed, using a number crunching server as an example. Fourthly, the possibility of using message passing rather than pipe **read()** and **write()** calls is discussed. Finally, there is a program illustrating message passing between components, rather than Posix **read()** and **write()** calls.

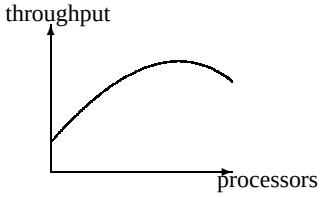


Figure **4.16** The throughput curve

4.5.1 Communication versus computation

When solving a problem in parallel, two types of cost must be considered. First of all there is the computation. If one processor can solve a problem in x seconds, then n processors should be able to do exactly the same amount of computation in (x/n) seconds. The n processors are doing useful work for (x/n) seconds. This assumes that the amount of computation can be split up in some way, so that the different processors can all do part of the computation.

The processors must exchange data if they are to carry out the computation. It is necessary to split up the data and transfer it from one processor to another. This combined cost is the communication cost. If the problem was solved on a single processor, the communication cost would be zero. Therefore all of the communication cost is an overhead.

There is a third cost: installation. This term applies to mapping the task force onto the available processors, loading the components from the disc into these processors, reading initial data from disc, writing final results to disc, etc. If the task force is going to be used only a few times, the extra cost of producing a parallel rather than a sequential solution must also be considered. For many task forces the installation cost will be negligible compared with the other two, but there are always exceptions. Installation cost will not be considered further here.

For a given amount of computation, as the number of processors used increases, the amount of computation done per processor must decrease. Also, the amount of communication will increase. Hence the proportion of time usefully spent will decrease. There are two parameters to consider. First, the computation throughput per second (the amount of real work done) (see Figure 4.16).

When the number of processors increases to a certain level, the throughput will actually decrease because the increase in communication is more than the computation done by the extra processors. It never makes sense to have so many processors that the throughput actually decreases. A closely related parameter is 'speed up'. This is a measure of the throughput on n processors compared with the throughput on 1 processor, the speed up achieved by having a parallel rather than sequential solution. The second parameter is the efficiency, which is the throughput per processor. This measures how much time is spent usefully on every processor, the time spent computing divided by the total time spent computing and communicating (see Figure 4.17).

On a single processor the efficiency is 100%, and this drops as the number of processors (and hence the communication) increases. The throughput for n processors is $n *$ efficiency(n). The turning point for throughput occurs when $(n *$ efficiency$(n)) >$

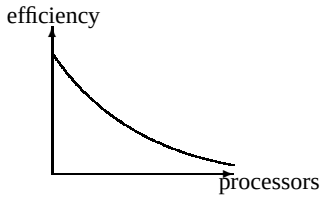


Figure **4.17** The efficiency curve

$((n+1)*$ efficiency$(n+1))$.

For some applications the efficiency and throughput curves might not be smooth. For example, an application might be quite efficient with 16 workers in a $4*4$ array, or with 20 workers in a $5*4$ array, but give poor performance with 17, 18 or 19 workers.

For a small number of processors, the throughput rises quickly. This has an important consequence. Given a 16 processor network, the throughput of a single task force will be less than the combined throughput of two task forces, both running on 8 processors. The most efficient use of the network can be achieved by having 16 task forces, all on one processor. Of course it will take longer to calculate these 16 results than it would take to calculate a single result on 16 processors, and it is necessary to strike a balance between the response time and efficient use of the network.

The exact slope of the throughput curve depends very much on the application. For some applications it is possible to maintain high efficiency even with many thousands of processors. For other applications the throughput is actually reduced by having even two processors. (The communication overheads are such that with two processors each operates at less than 50% efficiency.) For any given application it is difficult to estimate the throughput curve in advance with any degree of accuracy, but some simple experiments should give a good idea. There are various ways in which the communication costs can be reduced. The first way is to reduce the number of packets sent, by increasing the amount of work per packet. This solution is particularly appropriate for a farm. It is not always applicable if the amount of communication is determined by the problem. Thus in image processing it is possible to have a two-dimensional array of workers, with every worker needing to communicate with its neighbours to deal with the boundary conditions. This communication is implicit in the problem, and cannot be reduced in any way.

The second way to reduce the communication cost is to send a few large packets instead of many small ones. The cost for a single-pipe I/O can be divided into a fixed latency cost, to do any I/O at all, and a variable cost proportional to the amount of data. Sending one large packet instead of 10 small ones saves $9*$ latency cost. As an example consider the following transfer rates achieved for pipe I/O between two processors 4 links apart, for different packet sizes.

Packet Size (bytes)	Transfer Rate (Kb/s)
4	2
64	38
2048	507
65536	1372

It would take almost 3.5 seconds to transfer 128K of data across 4 links using 64-byte packets. Using two 64K packets the transfer could be achieved in a fraction of a second.

It should be noted that the above measurements were made in an otherwise idle network. Computation will not usually affect the time taken for communication, because under Helios the former will run at low priority whilst communication occurs at high priority. However, if many components are communicating over the same set of links then these will affect each other adversely.

Again it is not always possible to adjust the size of the packets to get better performance, because this size is determined by the problem. The third way to improve performance (possibly the most important) is to identify and attempt to remove any bottlenecks. These were discussed in some detail in the previous section. Not all bottlenecks can be eliminated, and if so this will severely limit the number of processors that can be used. To detect bottlenecks it is necessary to experiment, using performance monitoring code in the various components.

The final way of improving efficiency may seem obvious but is often overlooked. Make the components themselves more efficient. If a programming trick cuts 10% off an inner loop in a sequential program, exactly the same trick will make the worker components of a task force more efficient. The saving will not be the full 10% because of the communication costs, but there will be a saving none the less.

4.5.2 Problems with worker components

In a typical network there might be a 4 megabyte root processor with some additional 1 megabyte processors. Quite often the master component requires more than 1 megabyte, and so the CDL script will place it on the root processor, for example by specifying its memory requirements. The workers must run on the 1 megabyte processors. These processors really have just one megabyte, and it is remarkable how quickly that can be filled up. Consider the following innocuous FORTRAN statement.

```
DOUBLE PRECISION MATRIX[256,256]
```

Such a statement will give no problems at all on a traditional mainframe with a virtual memory system, where portions of the matrix can be swapped out to disc whenever necessary. On a Transputer this statement would use up $256 * 256 * 8$ bytes, or half of the 1 megabyte attached to the processor. If we also consider the rest of the memory required by the program data, the program code, the space required for the various libraries including approximately 100K for the FORTRAN library, and some space for the operating system, 1 megabyte may not be enough. Applications where every worker really needs so much memory may not be appropriate for the Transputer hardware in use, and there is nothing that Helios or any other operating system can do about

this. Of course it may well be possible to modify the application. If the matrix is a sparse one, with perhaps just a few thousand actual values instead of the 64K possible, it can be stored and processed in appropriate data structures without using so much memory.

The other common problem with task forces is that the worker components try to do their own I/O, instead of passing results back to the master component. For example, consider a task force of 100 components all trying to access a file on disc. If the Transputer network is hosted by a PC then MS-DOS [4] imposes a limit of 20 open files for all of the processors. On a Sun host the limit is 64, but still less than the application demands. Hence the application will fail because of a design error. Even worse, it may work perfectly on a small processor network with just 10 components and fail just when the time comes to demonstrate the application on a larger network, for reasons that are not immediately obvious, or it may fail if the application is moved to a slightly different environment.

Limits on the number of open files are not the only limits to consider. Suppose that every one of the workers writes a log to the standard error stream, which happens to be a window in the I/O Server running on a PC. With 10 workers this means 10 open streams to the I/O Server, all using up a significant amount of memory. With 100 workers, a PC I/O Server will run out of memory long before 100 streams can be opened. After all, the PC I/O Server is running in at most 640K of memory, which is less than the minimum of a megabyte attached to a typical Transputer. Again, the application will fail for reasons not immediately obvious if attempts are made to run it on a larger network or in a different environment.

If every worker tries to interact with the Helios X window system server the problem is slightly different. The X server may be running on a processor with sufficient memory to cope with a large number of connections. However, every worker would require the X library, about 128K, and possibly the X Toolkit library and the Widget library at 64K each. As a result 256K out of the megabyte available would be used up just for these libraries, quite possibly not leaving enough memory to do the real work.

Apart from the improper use of resources described above, there is another good reason why worker components should not perform any I/O. The Helios Task Force Manager is responsible for mapping the components onto the available network in an efficient way, trying to minimise the distance between communicating components. If all the workers spend their time doing I/O which is not related to the inter-component communication, the Task Force Manager will not have allowed for that and the effort spent trying to achieve an efficient mapping will have been wasted. The rule is that workers should communicate only with other components in the task force. Any real I/O should be left to the master component if possible.

4.5.3 Parallel servers

The Helios parallel operating system is based on a client-server model. To perform certain types of work, an application or client sends a message to a server, which may be on the same processor or it may be anywhere else in the network, and this server program does the real work. Under Helios it is possible to produce a server

[4] MS-DOS is a registered trademark of the Microsoft Corporation

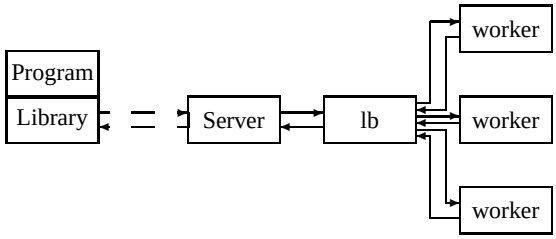


Figure **4.18** A parallel server

which is a task force. For example, consider a number crunching library on a single-processor machine which is used by many existing applications. Rather than rewrite all the applications to work in parallel, the library is turned into client code of a number crunching server. Suppose the application performs a library call for a complicated matrix operation. The library sends a request to the number crunching server, with all the data, and the server performs the work on multiple processors. When the work has been done the server returns the result to the library, which returns control to the application (see Figure 4.18).

The master component of the task force installs itself as a Helios server, in the standard way. When the library needs to access the server it opens a stream to the server, transfers all the required control information and data using this stream, and receives the results over this stream. The server accepts multiple open requests from different clients and receives jobs from these clients. Each job is evaluated using some or all of the workers, and the results are returned to the client. The advantage of the scheme is that the existing applications run unchanged.

Not all numerical problems can be solved using this approach. First of all this approach is less efficient than parallelising the actual applications, because there is additional communication overhead between the library and the server. Second, not all standard library routines can be handled in this way. In particular, if the application passes the address of a compiled function to a library routine, perhaps a routine for evaluating an integral, then there is no easy way of passing this function to a server on a remote processor, and certainly not to all the workers. A string representation of the function might be used, which could be interpreted or compiled in all the workers, but this is only possible for relatively simple functions and it means that the applications would have to be changed. However, there is no reason why the numerical library could not handle such routines on the local processor, whilst passing other calls to the remote server.

Chapter 5

Compatibility

5.1 Introduction

This chapter covers the compatibility of Helios with various actual and *de facto* standards, and the porting of programs to Helios. The baseline for Helios compatibility is Unix, in its various flavours. The intention is to make the porting of a program to Helios almost as easy as porting it from one type of Unix to another. The emphasis here is to aid the portability of **application** programs rather than system programs. Code which makes unreasonably detailed assumptions about the operating system it is using will not port directly. However, depending on the assumptions made, such system programs can be ported with few changes.

5.2 Unix compatibility

There are almost as many different forms of Unix as there are hardware vendors. However, some standards are emerging, and it is to these that any compatibility measures must adhere. The two explicit standards are POSIX (IEEE 1003.1-1988) and X/OPEN. There are also two *de facto* standards: System V.4 and BSD4.3. The former has the might of AT&T behind it and the latter is most widely used in the academic and scientific establishments, where Helios is most used. It must be emphasised that POSIX and X/OPEN merely codify current Unix practice. They are not generic operating system interface standards. These standards are currently converging, and whilst a common subset based on POSIX will probably emerge, features not covered by the standard will still be implemented in widely differing ways.

The model of computation under Unix is of a group of single-threaded processes all executing on a single processor. The Helios model consists of a group of multi-threaded tasks distributed across several different processors. Also, in Unix all operating system functions, such as the file system, terminal I/O or Resource Management, are performed by a single Kernel. In Helios these functions are distributed amongst several Kernels and servers. The result of this is that the Unix model is seriously inadequate for describing programming under Helios, and it makes a number of basic assumptions which are not true in a distributed environment. The following sections describe some of these inadequacies, and how Helios deals with them.

5.3 File handle sharing

Under Unix the file descriptors of a process are small integers which index a per-process Kernel table of pointers to globally shared file handles. These handles contain, amongst other things, the current file position. A **dup()** operation merely copies the pointer. A **fork()** will copy the pointer table, but not the handles. This means that every access to a particular open file, whether it is through duplicated descriptors in the same process or through the same descriptor in different processes, will use the same file handle. In particular, all such descriptors share the same file position, so a read or write through one descriptor will affect the position of the next read or write through another descriptor.

Under Helios, the task accessing a file and the server which contains it will almost always be on different processors. If the task creates a child, this may be on a third processor. Without shared memory between the processors, there is no possibility of maintaining a shared file handle, and each client has its own file position. This means that reads and writes made from different tasks are independent.

One of the design aims for Helios was to make it fault tolerant. To achieve this it must be possible for a client to continue running across a crash and to restart all of the servers it is using. In the same way, a server should not be affected by the failure of any of its clients. This has been achieved by making servers stateless, and maintaining with the client any state information required for access to an object. If the server crashes and restarts, the client can continue because no state has been lost. If the client crashes, all state relating to that session is lost with it, and the server does not need to take recovery action. For these reasons, the obvious approach of keeping the file position in the server, which would preserve Unix file handle sharing semantics, is unacceptable. Fortunately, few applications exploit this feature of Unix, so compatibility is seldom a problem. The Posix library implements the correct semantics in the case of **dup()**, but not in the case of **fork()/exec()**.

5.4 fork()

On a single processor with memory management hardware, **fork()** is a conceptually simple mechanism for creating new processes. However, depending on the processor architecture, available memory and swapping strategy, it can be complex and expensive to implement. In most cases this effort is wasted, since the new child almost immediately executes an **exec()** call which destroys the entire address space which was so expensively duplicated. For this reason, BSD introduced the **vfork()** call, which does not duplicate the address space of the process, but only its Kernel environment (current directory, file descriptors, process/group/user ids, etc). The new child uses its parent's address space until it executes an **exec()** call or an **exit()** call, and the parent is suspended during this period. This allows the child to manipulate the Kernel environment, move file descriptors, and change user or directory, before executing another program.

Clearly there are problems in the implementation of **fork()** in the environment in which Helios operates. The most fundamental problem is that the T400 and T800 series Transputers do not have any memory management hardware. If all the memory

belonging to a particular process was duplicated locally, it would not occupy the same range of addresses, and any pointers would still point back to the original memory. It might be possible to duplicate the process into another processor, at the correct position but the likelihood of finding a processor with exactly the right range of addresses free is slim.

For the above reason alone, **fork()** is impossible to implement. However, there are further, more fundamental problems, with the whole concept of **fork()** as a process creation mechanism. In an environment where processes are internally multi-threaded, the **fork()** operation is called by only one thread. The execution state of the other threads is indeterminate. What should be done with these threads? One alternative is to duplicate all threads into the new process. This then raises the problem of what happens to threads which are blocked in, or about to execute system calls which should not, or cannot, be executed in the new process. The only other alternative is to duplicate only the thread which called **fork()**. This raises a problem with process-local Resource Management. If a thread holds a resource in the parent, that resource will never be released in the child, because the thread which should release it has not been duplicated into the child. Both of the options described above would require considerable effort on the part of the programmer to deal with the consequences of a **fork()**.

Unix **fork()** also has problems in a distributed environment. In a distributed system, process creation is the ideal point at which to decide in which processor a process should execute. Unix **fork()** is potentially expensive even in a uniprocessor. It would become totally unacceptable if the entire process state and address space needed to be copied to another processor as well. A distribution decision could also be made when **exec()** is called. This is more acceptable, since the address space is to be replaced and the process state is at its minimum. However, there are still problems with this. The new process is created in the Kernel of one processor, which must then transfer it to another, yet it must retain some information for the benefit of the parent process. Such a mechanism still incurs the cost of creating a new process address space, only to destroy it soon afterwards. A process which **exec()**s without forking will move yet again, introducing a further indirection between it and its parent.

Another main use of **fork()** is to create a new thread of control in the existing program. This is usually necessary to overcome the purely sequential nature of Unix processes and perform some form of multiplexing. This is most frequently present in programs written before **select()** (or **poll()** in System V) was introduced. The multi-threading introduced by this mechanism is of limited use since the processes have disjoint address spaces, and any communication between them must be achieved by using signals or pipes which were set up before the **fork()**. This is more adequately catered for by the use of process-internal threads. When a standard for internal multi-threading emerges (POSIX 1003.4 is working towards such a standard), this use of **fork()** can be expected to cease.

Another use of **fork()** is by programs such as mailers and printer daemons to continue processing in the background. In this case, the program forks a child and then terminates. The original parent, which is usually a shell, sees its child finish and can continue. Meanwhile, the grandchild can do the job in the background without forcing the user to wait for it to finish. This is better dealt with by an explicit **detach()** call, which is a more portable mechanism, and can be implemented much more cheaply than by using **fork()**.

The obvious conclusion is that **fork()** is not appropriate for process creation in a multi-threaded, distributed environment. The only function of **fork()** which cannot be provided by other, more efficient and more powerful, mechanisms is its role in running new programs. It is undeniably attractive to be able to manipulate the environment of a new program before it is entered by means of normal C code between **fork()** and **exec()** calls. This is the major use of the call, and it must be retained if Unix compatibility is to be preserved. For this reason, **vfork()** is retained in Helios solely as a prefix to **exec()**, to allow this environment manipulation to take place. The same restrictions apply to Helios **vfork()** as to the BSD version. The child may not return from the calling procedure, and it must terminate with **exec()** or **_exit()**. For source compatibility, **fork()** is defined as a macro which executes **vfork()**.

5.5 Signals

In Unix, a signal is delivered to a process by invoking a signal handling procedure on the top of the stack of the process. This is possible because the Kernel explicitly schedules processes, and it is aware of all process states at all times. Transputer versions of Helios use the processor's scheduler, and have no direct involvement in either the creation or the scheduling of processes. For this reason, Helios is unable to deliver signal handlers onto the top of the stack of a process, because it has no way of knowing where this is.

Helios delivers signals to a separate signal delivery process. This means that signal handlers will execute in parallel with the other threads in the process. It also means that the Unix practice of exiting a signal handler by longjumping back to some recovery code will not work. To preserve such behaviour, the Posix library allows signal handlers to be marked as synchronous or asynchronous. An asynchronous signal handler is called in the signal delivery process. A synchronous signal is only called when the Posix library is entered. By default, SIGABRT, SIGHUP, SIGINT, SIGKILL, SIGQUIT and SIGTERM only are delivered asynchronously. If a user signal handler is installed for any of these it reverts to being called synchronously. Only if the handler is installed using **sigaction()** with the SA_ASYNC bit set in the flags will the user handler be called asynchronously. This does not guarantee that the longjump trick mentioned earlier will work in a multi-threaded program, but it will work in a Unix compatible single-threaded program.

Like **fork()**, Unix signal handling has fundamental problems in a multi-threaded distributed environment which are not related to Helios or to the Transputer. Signals are used for three purposes in Unix.

1. Reporting synchronous hardware traps such as SIGSEGV, SIGKILL and SIGFPE.

2. Reporting asynchronous events sent explicitly by other processes, or the system, such as SIGINT, SIGABRT, and SIGHUP, etc.

3. Reporting asynchronous events which are used to multiplex a single thread between several activities. Signals in this class are SIGCHLD, SIGALRM, SIGIO etc.

The **synchronous** signals are associated with the thread which causes the hardware trap. In a multi-threaded environment, there are two alternatives for handling these. In the first option, the signal is delivered directly to the faulting thread. The problem with this is that the thread may have corrupted its stack, and this may be the reason why it trapped. In the second option, the thread is suspended and the signal is delivered to another nominated thread, together with the original thread's state. The signal handler can then inspect and modify the trapped thread, and possibly resume it. This could become highly machine specific, and it is at variance with current practice, although it is used by MACH and Topaz. There is no standard practice at present, although the first option is preferable for the sake of compatibility. Currently there is no mechanism for raising these traps on the Transputer, so the Transputer version of Helios does not support these signals.

Of the two types of **asynchronous** signal, the provision of multi-threading eliminates the need for the second class, although they will need to be supported for the sake of compatibility. The alternatives for handling asynchronous signals are as follows.

1. Deliver all signals to all threads.

2. Deliver all signals to a nominated thread.

3. Deliver each signal to a thread nominated on a per-signal basis.

4. Introduce a function which suspends the calling thread until one of a specified set of
   signals is pending.

5. Create a new thread for each signal as it is delivered.

6. Deliver all signals to a special signal handling thread.

7. Deliver each signal to an arbitrary thread.

Clearly option one leads to chaos, option two is close to current practice, option three results in contention between threads over the single per-process signal mask, and option four is too different from current practice. Option five is the mechanism used in Helios until version 1.2, and options six and seven are combined in the signal mechanism for Helios version 1.2 onwards.

The POSIX 1003.4 Realtime Extensions Committee in Draft 8 (August 1989) has chosen option two, where the nominated thread is always the initial thread of the process, since it is backwards compatible with existing systems. However, the committee has documented a function called **sigwait()** to implement option four, in the hope of encouraging its use and eventual standardization. The advantages of this mechanism are that it removes all restrictions on the signal handling code, it allows multiple signals to be handled simultaneously, and it allows different signals to be handled at different priority levels. With multi-threading, there is no longer any need for signals to be treated analogously to interrupts, and a more controlled mechanism for handling them is preferable.

5.6 Process identifiers

The single Unix Kernel is in sole control of all processes in its system, and it can choose process ids which are unique. In Helios, there is an arbitrary number of processors, each running a different Kernel. It is impossible to ensure that any particular process id is unique throughout the entire system. Posix restricts *pid_t* to being a signed integer type. This means that conventional mechanisms for concatenating site id with a site unique value will rapidly run out of bits. If, as in Helios, subnetwork may be disconnected and reconnected at will, the system must cope with the possibility that previously disjoint networks are using the same set of process ids. Also, a primary design objective of Helios was that all objects should have textual names with meaning in human terms, rather than arbitrary 'magic numbers'. For this reason, all running tasks are entered in the **/tasks** directory, with a name derived from the name of the program in use.

In the Helios Posix library, the process ids returned by **getpid()**, **vfork()** and **wait()**, which should be supplied as arguments to **kill()** and **waitpid()**, are entirely local to the calling task. They are used merely as tokens by the library, to represent internal data structures. The process id of any task is 42, and its parent is 41. Each new invocation of **vfork()** yields a monotonically increasing sequence starting from 43. Between **vfork()** and **exec(), getpid()** returns the new value and **getppid()** returns 42. A task may only **wait()** for or **kill()** its own children through the Posix library. Control may be exercised over other tasks by means of the Helios mechanisms, particularly **SendSignal()**.

5.7 User and group identifiers

Protection of resources in Unix is achieved by assigning an owner's user identifier and a group identifier to all objects. Access rights for the owner, for the group members and for the general public are also stored. In the same way, processes are given user and group ids. When a process attempts to operate on an object, the identifiers are compared and, depending on the results, the appropriate set of access rights is used to determine whether the operation is allowed. This mechanism relies on the Kernel to protect the identifiers stored with objects and processes against tampering. This is achieved by storing them outside any process address space, and by closely controlling the functions which manipulate them. Helios for the T400 and T800 series of Transputer has no memory protection, so cannot prevent users from tampering with their user identifiers. However, as already mentioned, there are fundamental flaws in the Unix mechanism which make it unsuitable for a distributed system. In brief, to duplicate the Unix mechanism in a distributed system requires a centralised user authentication server which must be consulted before every operation on every protected object, to validate the user. This slows down all operations and vastly complicates the implementation of all servers. It is a centralised system, with all the consequent problems of reliability, and it could become a serious bottleneck. If a user could fake an authentication server, the whole system would be open to them.

The Helios mechanism for resource protection and access controls is based on encrypted capabilities. This is a distributed system, and the implementation of the

protection mechanism is entirely in the hands of the servers containing the protected objects. Consequently, the overheads are minimal, and there is no problem with reliability beyond that of the original server. Users cannot fake a capability in the same way as they can fake a user id. The first is a 64 bit number chosen from a sparse number space, and the second is listed in the password file. No special means of storing the capabilities is needed in the operating system. They can be stored in user memory, and the user can gain no advantage by tampering with them.

In choosing a protection mechanism which is suited to a distributed system rather than to Unix emulation, a large degree of compatibility has been sacrificed in this area. However, experience suggests that, with the exception of a few system programs, portable Unix programs do not manipulate their own ids, or object user and group ids. A simple translation between Unix access modes and Helios access matrices is sufficient to support programs like **tar** and even NFS.

The Helios Posix library from version 1.2 stores a single user and group id for a task which can be manipulated with the usual calls. Unix programs normally have both real and effective user and group ids, but since the only mechanism for making these different is the set-user-id option on executable programs, which Helios does not support, there is no need to store both. Similar advantage has been taken of the Posix option to define the number of subsidiary group ids as zero, so nothing need be stored.

Normally Helios does not pass the uid and gid from parent to child, since they serve no useful function. However, if the environment variables _UID and _GID are set when a program is entered, uid and gid are set from them. The values of these variables must be 8 digit hexadecimal numbers. In the same way, the current uid and gid will be passed on by **exec()** to any child programs if these variables exist.

5.8 BSD compatibility

A new addition in Helios version 1.2 was a Berkeley Unix compatibility library. This is a link library of routines which have been written while porting a number of public domain programs. These routines are a mixture of code written by Perihelion, public domain sources and a few genuine BSD sources. Again, the emphasis here is on minimising changes to source code rather than complete emulation of BSD. No documentation is supplied for these routines. They are all documented by Berkeley. The following is a list of the contents of this library.

```
alloca()          bcmp()          bcopy()           bzero()
closelog()        ffs()           ftruncate()       getopt()
getpass()         getw()          getwd()           index()
inet_addr()       inet_lnaof()    inet_makeaddr()   inet_netof()
inet_network()    inet_ntoa()     initgroups()      insque()
ioctl()           mktemp()        openlog()         pclose()
perror()          popen()         psignal()         putw()
rcmd()            readlink()      readv()           remque()
rexec()           rindex()        rresvport()       ruserok()
seekdir()         setegid()       seteuid()         setgroups()
setlinebuf()      setlogmask()    setrgid()         setruid()
sigblock()        sigpause()      sigsetmask()      sigstack()
sigvec()          strcasecmp()    strncasecmp()     syslog()
```

```
telldir()       truncate()      writev()
```

The library also contains the following variables or tables.

```
sys_errlist      sys_nerr      sys_siglist
```

For implementation purposes, the following routines are in the Posix library.

```
getdtablesize()   gettimeofday()   lstat()

wait2()           wait3()
```

Some extra headers have been added to the **include** directories to support BSD compatibility. These are listed below.

```
sgtty.h        strings.h         sys/dir.h       sys/errno.h
sys/file.h     sys/ioctl.h       sys/param.h     sys/resource.h
sys/time.h     sys/ttychars.h    sys/ttydev.h    sys/un.h
varargs.h
```

BSD compatibility features have been added to the following existing headers.

```
errno.h         fcntl.h         pwd.h         signal.h
stdio.h         string.h        sys/stat.h    sys/types.h
sys/wait.h      unistd.h
```

The BSD compatibility features in the existing headers are only enabled if the macro **_BSD** is defined. The BSD only headers will generate an error message if they are included when **_BSD** has not been defined. Care must be taken when mixing Posix and BSD specific code, because the headers redefine some things for the BSD option.

The support for **ioctl()** is limited. For terminals it is implemented on top of the Posix termios system, and it only implements the common features. The most commonly used attributes: RAW, ECHO and TANDEM are implemented. The ability to change the special characters is not currently implemented. All ioctls to internet sockets are implemented with the exception of SIOCGIFCONF. The following ioctls each correspond to their BSD equivalent.

```
FIONREAD      FIONBIO      TIOCGETD      TIOCGETP
TIOCSETP      TIOCSETN     TIOCFLUSH     TIOCGETC
TIOCLSET      TIOCLGET     TIOCGPGRP     TIOCSPGRP
TIOCGLTC      TIOCSTOP     TIOCSTART     TIOCGWINSZ
TIOCOUTQ
```

Some of these (TIOCFLUSH for instance) simply call the equivalent Posix routine, which may not be implemented. Those ioctls mentioned in the headers but not in this list will do nothing, and return zero.

5.9 Porting techniques

This section gives some helpful hints on porting programs to Helios. It only covers programs written in C for Unix, although programs written in C for the PC, Amiga, Atari ST and even the Macintosh should port as easily if they make few machine specific assumptions. Programs which are genuinely written for portability present few problems. In many cases they simply need to be compiled and linked.

There are two potential areas of difficulty in porting a program: the C language, and system functionality. The Helios C compiler is ANSI standard, while most Unix programs are written to K&R standard. The ANSI standard includes the K&R features for compatibility, but some of these are known as 'deprecated' features, and their use is discouraged. An example of this is the use of a function declaration without giving any argument types. These deprecated features all generate warning messages, but they can be turned off with the **-w** flag to the compiler. In version 1.3 of Helios, the C compiler driver can take the flag **-wA** to turn off all optional warning and error messages. This is recommended on all K&R style programs.

CAUTION: This option can sometimes suppress too many error messages, in particular the message concerning the use of nested comments.

Once the sources of a program have been compiled, they must be linked. Putting aside references to library code for a moment, there is a potential problem with linking the parts of a program together. This problem arises from the practice of declaring all references to a variable as **extern**, without explicitly defining it and allocating storage. The Unix linker is then responsible for allocating the storage in the data part of the final program image. Under Helios, all programs are composed of modules which export a set of variables and functions to their peers. If there is no definition of a variable, it cannot be allocated to a module, and therefore it does not fit the Helios model. There are two simple solutions to this problem. The first is to selectively remove the **extern** directive from one declaration of the variable, thus allocating it to that module. The second solution is to add an extra source file containing definitions for all such variables. The advantage of this is that the original sources need not be altered. Fortunately, it appears that few Unix programmers use this feature, and most take a modular view of their programs, with explicit definitions and clearly defined interfaces.

The remaining area of difficulty is in system functionality. A program written purely in terms of the runtime system should have no problems, but such programs are rare. More frequently, programs use a number of Unix system calls. Portable programs will have conditional compilation for different Unix systems. If this is the case, try the POSIX option, if it exists. Otherwise, try **-wA -D_BSD** in the compilation, and **-lbsd** when linking with the C compiler driver. This has been found to work for a large number of programs.

If some routines are still undefined, your only option is to write them. It is preferable to add some stubs to implement the missing functions, rather than to change the sources. If it becomes necessary to change the sources in any way, ensure that the change is made in conditional compilation flags. The C compiler defines the macro **__STDC__** for marking ANSI C code. The compiler driver defines the macros **__HELIOS**, **__proc** and **__HELIOSproc**, where **proc** is one of the processor types: **TRAN**, **ARM**, **i860** etc.

The only area where significant changes may be needed is in the use of **fork()**. If it is used as a prefix to **exec()**, nothing need be done, although you must ensure that **fork()** is called at the same level as, or at a higher **fork()** level in the calling tree than **exec()**. It is not permitted to call a procedure to call **fork()**. If the **fork()** is used to detach the program from its invoker, it must be removed and the invocation must be altered. This may be as simple as writing an alias or a shell script to call the program in the background. When a **fork()** is used to obtain a new thread, it can usually be replaced easily by using internal threads. Care must be taken with any assumptions made about the separation of the process address spaces.

There are five further problems when porting code. The first problem is that Helios does not provide a version of the Bourne shell, so utilities and programs which rely on it will not necessarily work. The second problem is that file system limitations can seriously affect the ease of porting large pieces of code. In particular, if Helios is hosted by the MS-DOS or TOS filing systems, then the eight character filename, three character extension case insensitivity limitations can cause problems, as can the lack of hard or symbolic links.

The third problem is with function name clashes. Helios has its own specific system call names which differ from Unix system calls, so the function names of application programs can clash with Helios system function names such as **Malloc()**, **Exit()**, etc. The fourth problem is with changed **socket** domains. Programmers using **socket** should note that the **AF_UNIX** domain does not exist. Instead, they should use **AF_HELIOS**. Also, Helios sockets do not need to be unlinked before or after use, since they are not created into the filing system, whereas Unix sockets are so created.

The fifth problem is with memory corruption. The greatest problem encountered when porting Unix code will be the lack of memory protection in a Helios environment. Such practices as using memory after it has been freed, accessing negative elements of arrays and placing large arrays on the stack can all cause programs to crash.

5.10 Multi-threaded library access

Helios is designed to be a multi-threaded system. Unfortunately, the standards for the C library and the Posix library were not designed as such. This means that you must be careful when using these libraries in a multi-threaded environment.

The ANSI C library contains no interlocks, so simultaneous calls into it are not protected from interference. However, if no common data structures are used, two threads can call C library routines simultaneously. For example, I/O on different FILE structures is quite safe, but calling **printf()** from two threads simultaneously can produce unpredictable results. The use of the Helios system call **IOdebug()** to produce debugging output is recommended. This lack of interlocking is largely a result of the implementation of many C library functions as macros, particularly **getc()** and **putc()**. They are implemented in this way for the sake of speed. To call procedures to claim and release interlocks on data structures would defeat the purpose.

The Posix library is almost entirely procedural, so it is possible to perform some interlocking. In general it is safe to call most Posix routines simultaneously from several threads. Problems may arise where routines are defined to use static data areas, for example: **getpwnam()**, **ctermid()** and so on. Such areas are allocated once per

task, and not on a per-thread basis. Similarly, **errno** is a single variable for the entire task, so errors in two threads simultaneously may result in the loss of one error code. The Helios system libraries are totally reentrant, with full interlocking on all **Streams** and **Objects**. This means that they are always safe to use in all situations.

Chapter 6

Communication and performance

6.1 Communication

Helios runs on a wide variety of system configurations. The only assumptions made about the underlying hardware is that it comprises a network of processors that are inter-connected by some means. The physical connection medium can assume various forms – for example, serial links or ethernet.

Typically, Helios runs on Transputer networks. As described in the networks chapter, the fundamental features of the T400 and T800 series Transputer are a 16 or 32-bit processor (CPU) and high-speed communication links that provide point-to-point inter-processor connections. These components reside on a single chip and can operate concurrently. The IMS T414 Transputer is an example of this basic model – other special purpose members of the Transputer family feature additional circuitry, microcode and interfaces that support specific tasks (for example, disc and memory controllers). The IMS T800 includes an on-chip floating-point unit (FPU). The CPU, memory, communications links and FPU all share a 32-bit data and address bus. An external memory interface is provided to allow access to additional, off-chip memory.

This chapter concerns the communication mechanisms provided by Helios, and it is therefore worthwhile to briefly examine the underlying hardware communications system. The Transputer has a number of serial links (typically 4) that provide full duplex, synchronous inter-processor communications. Each link has an input and output channel. A connection between two Transputers is implemented by connecting these channels through a pair of uni-directional signal lines.

Data that is sent along a links output channel is acknowledged on the input channel, and synchronisation is implemented by a handshaking technique. The IMS T800 allows messages to be pre-acknowledged, and each of the links are capable of maximum uni- and bi-directional data rates of 1740 and 2350 Kbytes per second respectively (see the Inmos handbook for further details).[1] The fact that each link has its own DMA controller means that any specific link can operate independently and in parallel with the other links, the CPU and the FPU. Communication, therefore, does not involve processor overhead. Networks of Transputers can be configured through the communications links to specific topologies – for example, a pipe, ring, mesh or hypercube.

[1] *Transputer Reference Manual* published by Prentice Hall, ISBN 0-13-929001-X

Message routing is not supported by current Transputer hardware.[2] This does not pose a problem when communicating between adjacent, directly linked Transputers – data can be simply sent and received over the connecting link. If, however, the data must traverse a number of intermediate processors, the message must be suitably routed. Message routing must be explicitly provided as a function of the software running on each Transputer.

The foundation of the Helios communication system is a message passing mechanism resident within the Kernel. Message passing is implemented by means of two Kernel primitive operations, **PutMsg()** and **GetMsg()**. It is important to note, however, that these routines are not reliable – they do not provide error detection or recovery, and there is no guarantee that messages will arrive at their respective destinations. Messages can become 'lost', for example if a processor in the network crashes, or if there is insufficient buffer space in an intermediate processor for a message. Message routes can become unusable if, for example, one or more intermediate processors are switched into a native mode, requiring a higher-level reconnection operation. Although the probability of messages not reaching their intended destinations is low, it is obviously essential to provide higher-level communication mechanisms that ensure some degree of reliability. Under Helios, these higher-level mechanisms take the form of pipes and sockets – these mechanisms support reliable, fault tolerant inter-process communications.

This chapter describes the use of Helios pipes and sockets. The utilisation of the low-level message passing primitives is also explained, although it is stressed that the use of these routines is not encouraged. The remainder of the chapter focuses on performance issues, and examines the extent to which the functionality of Helios compromises the raw computational and communications capabilities of the underlying hardware. There is a cost in terms of processing overhead associated with the provision of operating system services such as transparent and fault tolerant communications, and it is obviously crucial to be aware of the magnitude of this cost. It is shown by a series of experiments that Helios imposes a negligible amount of computational overhead, and that the message passing mechanism is highly efficient. For the sake of completeness, a short overview of the Helios model is included below.

6.1.1 Helios overview

The Helios parallel operating system is based on the Communicating Sequential Processes model of parallel programming.[3] Application programs are composed of tasks which communicate with each other to complete the application's objective. Tasks may be run on one processor or distributed between several processors. A collection of related tasks is called a task force and it is the responsibility of a Task Force Manager to map and load a task force onto the Helios network. Task forces can be executed either directly from the shell or by using the Helios Component Distribution Language (CDL), which defines how tasks are connected and mapped.

Typically a Helios processing node consists of a high performance processor, local

[2] The IMS T9000 series supports message routing through the internal VCP (Virtual Channel Process) and external C104 packet routing switch(es).

[3] *Communicating Sequential Processes*, C.A.R. Hoare. Published by Prentice Hall. ISBN 0-13 153271-5, ISBN 0-13-153289-8 PBK.

memory, and communication channels to other processors. Each Helios node runs a small Nucleus containing a Kernel, basic libraries, Process Manager and Loader. Additional services such as an X window system server, debugger server, and TCP/IP server may be added to any node on the Helios network; these services are, however, optional.

6.1.2 Pipes

Pipes are the most commonly used high-level communication facility in Helios. This is because Helios provides various means of automatically setting up pipes when running programs, without requiring any effort on the programmer's part. For example, the shell command `ls -l | more` will run the two programs **ls** and **more** in such a way that the two programs can communicate immediately over the connecting pipe. Similarly the CDL script `master [10] ||| slave` will run twelve programs connected in a farm topology such that all the required communication pipes exist by the time the programs start up.

The conventional way of communicating over pipes is with the Posix **read()**, **write()** and **select()** calls.

```
result =  write(1, buffer, sizeof(Packet));
result =   read(0, buffer, sizeof(Packet));
result = select(n, reads, writes, exceptions, timeout);
```

The exact means of calling the Posix routines vary from language to language. For example, when programming in C the routines can be called directly. When programming in Fortran there are jacket routines **POS_READ()** and **POS_WRITE()**. In addition the pipes may be used through language-level I/O facilities such as the standard I/O library supported by C, with its **printf()** and related routines. When programming at the language-level it may be necessary to cope with any buffering that is performed automatically at these higher levels. Chapter 4, *CDL*, describes this topic in more detail. Language-level I/O usually ends up going through the Posix library calls.

The Posix file descriptors 0, 1 and so on all have underlying Helios **Stream** structures, and the Posix calls utilise the System library routines **Read()** and **Write()** with these stream structures. Application programmers can make use of these lower-level routines if desired, but they do not offer any greater functionality than the Posix calls and only a small amount of performance benefit. Helios provides a routine **fdstream()** which, given a file descriptor, returns the underlying stream structure.

Read() and **Write()** calls acting on pipes, whether called directly or through Posix calls, are implemented using message passing directly from one program to another without going through an intermediate server. Pipes provide a reliable point to point communication facility, with automatic recovery from failures such as lost messages and broken message routes. It does not matter whether the two ends of the pipe are on the same processor or on different processors. Data written into a pipe can be read from the pipe at the other end in exactly the same order, and will not be discarded. To achieve this reliability, a protocol is implemented on top of the basic message passing, and additional threads are **Fork()**'ed off automatically by the system to maintain the pipe in a consistent state irrespective of the behaviour of the applications, for example

to cope with one side writing to a pipe before the other side is ready to read. This involves some overhead, as described later in this chapter.

The behaviour of pipes is fairly straightforward: one end of a pipe writes to it, and the other end reads from it. Nevertheless, there are a few subtleties:

1. The initial pipe connection happens when both programs attempt to perform some I/O on the pipe. Hence the first pipe I/O operation takes somewhat longer than subsequent ones.

2. A **read** operation need not return the amount of data requested. Even if one end of the pipe performs a write of 100K and the other end performs a read for 100K it is not guaranteed that the entire amount will be transferred in one go, because the data may have to be split into several messages. To cope with this it is usually desirable to make use of a **fullread()** routine, defined as follows:

```
bool full_read(int fd, char *buff, int amount)
{ int result;
  while (amount > 0)
   { result = read(fd, buff, amount);
     if (result <= 0) return(FALSE);
     amount -= result;
     buff += result;
   }
  return(TRUE);
}
```

3. A call to **read()** will return the amount of data actually read, if successful. It will return 0 if the end of the file has been reached, in other words if the other end of the pipe has closed the stream or terminated. It will return -1 to indicate an error. The above **fullread()** routine loops as long as data is being transferred, succeeding if all of the requested data has been read and failing if an error occurs or the pipe is closed.

4. For many applications it is desirable to ensure that both ends of the pipe exchange the same amount of data. Typically this is achieved by enforcing a simple application-specific protocol. For example, when using the Helios load balancer, pipe communication always involves a fixed size packet header and a variable size amount of data, with the size of the data held in the packet.

5. Write operations usually do not return until the entire amount of data has been transferred. Even this is not guaranteed, because certain events such as asynchronous signals may interrupt a call to **write()**. Hence applications may need to use a **fullwrite()** routine as well as a **fullread()** routine.

6. If the reader end of a pipe closes the stream or terminates then any attempts to write to the pipe will fail, and will cause a **SIGPIPE** signal.

7. Pipes can be used bi-directionally, in other words a program can both read from and write to the same pipe. However it is essential that the two ends of the pipe agree on the current operation, in other words if both ends attempt to write at the

same time the results are undefined. To avoid such problems it is recommended that most pipes are used uni-directionally.

If two programs are already running and they are not yet connected by a pipe then it is difficult to establish a new pipe between them. Instead the socket calls described later should be used. If a program needs to run another program, usually referred to as a child program, and it needs to communicate with that child through a pipe then this is possible. The simplest way to do this is with the **popen()** and **pclose()** routines provided in the BSD compatibility library.

```
FILE *to_child = popen("child_program", "w");
int   fd       = fileno(to_child);

  /*interact with the child using file descriptor fd */

pclose(to_child);
```

The **popen()** routine runs the specified program such that the program's standard input stream, file descriptor 0, is a pipe back to the parent. The C library **to_child** pointer refers to the other end of this pipe, allowing the parent program to interact with the child. The call to **pclose()** closes the pipe and hence the child can no longer read from the pipe; this will normally cause the child to exit. If the second argument to **popen()** is `"r"` instead of `"w"` then the child's program standard output stream, file descriptor 1, is used instead of the standard input stream.

If the **popen()** routine is too high-level for the application's requirements then it is possible to achieve the same effect with the Posix calls **pipe()**, **vfork()**, and **execvp()**. However, there is a restriction when passing pipes on from parent to child. At any one time any given pipe may be in use by only two programs. The **pipe()** call creates a single pipe, and provides two file descriptors for this pipe. One of these file descriptors should be used in the parent, and the other in the child. It is necessary to ensure that the extra file descriptor is closed. This is illustrated by the following code fragment:

```
int simple_popen(char *program, char **argv)
{ int   pdes[2], pid;

  if (pipe(pdes) < 0) return(-1);

  if ((pid = vfork()) == 0)
   {    /* In child */
     dup2(pdes[0], 0);
     close(pdes[0]);
     close(pdes[1]);
     execvp(program, argv);
     _exit(1);
   }

   /* In parent */
  close(pdes[0]);
  return(pdes[1]);
}
```

The first file descriptor, `pdes[0]`, is used only in the child program. It replaces the standard input stream for that program using a call to **dup2()**. The parent program used `pdes[1]` as the file descriptor to be used for communicating with the child. All unnecessary file descriptors are closed.

If the parent program will start more than one child then it is necessary to ensure that the child processes do not inherit file descriptors for the same pipe, since this would break the rule that a pipe may be used only by two programs. The following line achieves this:

```
fcntl(pdes[1], F_SETFD, FD_CLOEXEC);
```

6.1.3 Sockets

If two programs need to communicate with each other but are not yet connected by a pipe then it is necessary to establish a connection between them dynamically. The recommended way to do this is through the socket library calls. Sockets are more general than pipes, for example they allow communication between programs running on different machines connected only through ethernet or a similar network, and furthermore it is not necessary for both machines to be running Helios. The usual Unix networking software such as NFS, rlogin, ftp, and so on, is implemented using sockets.

First, this section describes how it is possible for two Helios programs to establish a socket connection which behaves in exactly the same way as the pipes described already. Next a description is given of the extra work involved in establishing a connection with a program running on a remote machine, which may or may not be running Helios. Finally some of the various socket options are described.

Unlike pipes, sockets are not symmetrical. It is not possible for both programs to call exactly the same routines to establish a connection between them. Instead one of the program must become a **receiver** and the other a **sender**, and the application programmer must decide somehow which one is which. Note that the programs cannot communicate with each other first to establish which one should be the receiver, since if the programs were able to communicate already it would not be necessary to establish a socket connection. Within the context of Unix networking software this does not matter; usually the receiver program is a server or daemon, and the sender is a client. The work that has to be done by the receiver is shown below:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/hel.h>

#define Socket_Name "SomeMagicString"

...
struct sockaddr_hel socket_addr;
int    addr_len = sizeof(socket_addr);
int    srv_socket, client_socket;

srv_socket = socket(AF_HELIOS, SOCK_STREAM, 0);
if (srv_socket < 0) /* error */
...
```

```
socket_addr.sh_family = AF_HELIOS;
strcpy(socket_addr.sh_path, Socket_Name);
if (bind(srv_socket, (struct sockaddr *)
         &socket_addr, addr_len) < 0)
 /* error */
...
if (listen(srv_socket, 1) < 0) /* error */
...
client_socket = accept(srv_socket, (struct sockaddr *)
                       &socket_addr, &addr_len);
if (client_socket < 0) /* error */
...
/* Interact with other program using */
/* file descriptor client_socket */

close(client_socket); /* when finished */
```

Most programmers do not need to know exactly what all this code does, since it can be put into a library and forgotten about. Nevertheless a brief description may be useful:

1. The first call to **socket()** creates a socket which, essentially, just initialises some data structures that can be used for subsequent operations. The routine returns a file descriptor, but this file descriptor cannot be used for reading or writing.

2. The socket is defined to be internal to the Helios network, in other words it cannot be used to access a remote program over the ethernet. The socket is a stream socket, and hence it will behave just like a pipe once the connection is fully established.

3. It is necessary to register the socket with the system, before other programs can interact with it. This is the purpose of the **bind()** routine. The second argument to **bind()** gives the identity of this socket, using a data structure containing a textual name. The name should be specific to this application and should not overlap with socket names used by other, unrelated, applications.

4. The call to **listen()** informs the system that other programs are now permitted to connect to this socket.

5. The **accept()** routine waits for other programs to connect to this socket. It returns another file descriptor which can be used for **read()** and **write()** operations, allowing communication with the other program in exactly the same way as communication over pipes.

6. Instead of the call to **accept()** it is possible to use **select()**, with **srv_socket** as one of the file descriptors in the **Read** vector. The **select()** routine will return when another application attempts to connect to that socket. This technique is typically used by Unix daemons which need to service requests from several clients, by reading the request data from appropriate sockets, while at the same time accepting connections from new clients.

The other side of the socket responsible for connecting to the receiver is more simple.

```
struct sockaddr_hel socket_addr;
int    addr_len = sizeof(socket_addr);
int    client_socket;

client_socket = socket(AF_HELIOS, SOCK_STREAM, 0);
if (client_socket < 0) /* error */

socket_addr.sh_family = AF_HELIOS;
strcpy(socket_addr.sh_path, Socket_Name);

while (connect(client_socket, (struct sockaddr *)
               &socket_addr, addr_len) < 0)
 { fputs("Sender: trying to connect socket...\n", stderr);
   sleep(2);
 }

 /* Interact with other program using */
 /* file descriptor client_socket */

 close(client_socket); /* when finished */
```

1. The sender program must also create a socket with the right characteristic, so that it can connect to the other program. The call to **socket()** returns a file descriptor, but this cannot be used yet for I/O operations.

2. The call to **connect()** matches the call to **accept()** in the receiver program. The second argument identifies the socket to connect to, again by using a unique name. Both programs must use the same name.

3. **connect()** may not succeed immediately, if the receiver has not yet called **listen()**, **bind()** and **accept()**. Hence the above code loops until the connection is established.

4. The **accept()** routine in the receiver returned a new file descriptor which is used for communication. The **connect()** routine, rather confusingly, turns the existing file descriptor into something that can be used for communication.

Once the two programs have performed the above steps they both have a file descriptor that can be used for communication. This file descriptor behaves in exactly the same way as the pipes described earlier, and the same restrictions about reading and writing operations apply.

Two aspects of the above code make it specific to communication within a single Helios machine: the address family type **AF_HELIOS** and the socket address structure **sockaddr_hel**. For socket based communication over the ethernet it is necessary to change the use of both of these. The address family should be **AF_INET**, and the socket address structure should be **sockaddr_in** as defined in the header file **/helios/include/netinet/in.h**. The contents of an internet address structure is different

to a Helios address structure: instead of specifying a simple string the address contains two numbers, a service number and a host number. To avoid embedding these numbers in binaries there are library routines which extract the information from the system configuration files **/helios/etc/hosts** and **/helios/etc/services**.

Suppose a program wishes to establish itself as a socket for the service "**hydra**". Typically this program would be a networking daemon. The code required would look something like this:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

...
struct sockaddr_in socket_addr;
int    addr_len = sizeof(socket_addr);
int    srv_socket, client_socket;
struct servent *sp;

srv_socket = socket(AF_INET, SOCK_STREAM, 0);
if (srv_socket < 0) /* error */
...
sp = getservbyname("hydra", "tcp");
if (sp == NULL) /* error */

memset(&socket_addr, 0, addr_len);
socket_addr.sin_addr.s_addr = INADDR_ANY;
socket_addr.sin_port        = sp->s_port;
socket_addr.sin_family      = AF_INET;
if (bind(srv_socket, (struct sockaddr *)
          &socket_addr, addr_len) < 0)
 /* error */

if (listen(srv_socket, 1) < 0) /* error */

client_socket = accept(srv_socket, (struct sockaddr *)
                       &socket_addr, &addr_len);
if (client_socket < 0) /* error */

/* Interact with other program using */
/* file descriptor client_socket */

close(client_socket); /* when finished */
```

The differences between this and the previous example are the use of **AF_INET** and the contents of the address structure. Note that when binding a socket it is not necessary to specify the machine to which the socket should be bound, since it must always be the local machine. For the sender side things are rather different, and it is necessary to look up the name of the target machine as well as the port number.

```
struct sockaddr_in socket_addr;
int    addr_len = sizeof(socket_addr);
int    client_socket;
```

```
struct servent *sp;
struct hostent *hp;

client_socket = socket(AF_INET, SOCK_STREAM, 0);
if (client_socket < 0) /* error */

sp = getservbyname("hydra", "tcp");
if (sp == NULL) /* error */

hp = gethostbyname(<machine-name>);
if (hp == NULL) /* error */

memset(&socket_addr, 0, addr_len);
memcpy(&socket_addr.sin_addr, hp->h_addr, hp->h_length);
socket_addr.sin_family = AF_INET;
socket_addr.sin_port   = sp->s_port;

while (connect(client_socket, (struct sockaddr *) socket_addr,
               addr_len) < 0)
 { fputs("Sender: trying to connect socket...\n", stderr);
   sleep(2);
 }

 /* Interact with other program using */
 /* file descriptor client_socket */

 close(client_socket); /* when finished */
```

If the appropriate entries for the target machine name or the service name are missing from the configuration files **hosts** or **services** then the routines **gethostbyname()** and **getservbyname()** will fail. These files are held in the **/helios/etc** directory under Helios and they are held under the **/usr/etc** directory under Unix. Once the socket connection has been established, the file descriptors can be used in exactly the same way as those for pipes. The sockets described so far have all been stream sockets. In fact there are several different types of sockets:

tcp or **SOCK_STREAM** sockets provide reliable point to point communication, like pipes.

udp or **SOCK_DGRAM** sockets provide a datagram service. Such a service provides unreliable communication. When a program performs a write operation the entire data is packed into a single message, if possible, and sent off as one message. This message may or may not arrive at its destination, it may arrive more than once, and it may arrive out of order relative to other messages.

raw or **SOCK_RAW** sockets exist to provide direct access to the underlying communication hardware. When using the Helios family of sockets these are equivalent to message passing, and it is possible to extract message ports from the appropriate Stream structures if applications need to perform message passing.

To make use of the different types of sockets it is necessary to change two parts of the above code fragments: in the call to **socket()** the desired socket type should

be specified, instead of **SOCK_STREAM**; in the call to **getservbyname()** the second argument should be "**tcp**", "**udp**" or "**raw**" as desired.

6.1.4 Message passing

The foundation of all Helios communications is direct message passing. Message passing is the lowest level of communication. The higher-level communication mechanisms are built on top of this, and feature protocols that provide aspects such reliability and portability. Typically, users need not concern themselves with message passing as this is handled internally by Helios. Indeed, although message passing is more efficient than the higher-level communication mechanisms, its use is not encouraged. This is because message passing makes no provision for error detection and recovery. At this level of communication, there is no guarantee that messages will reach their intended destinations and although the probability of a message getting lost is small, it does nevertheless exist. Message passing should therefore only be considered in circumstances where communication speed is critical and, importantly, where reliability is not crucial. In all other cases, use should be made of pipes or sockets in conjunction with higher-level library routines that feature built-in error detection and recovery facilities (Posix **read()** and **write()**, or Helios **Read** and **Write()**).

There are two message passing primitives, **GetMsg()** and **PutMsg()**. Both of these routines take as their argument a pointer to a message control block (MCB) :

```
typedef struct MCB {
  MsgHdr  MsgHdr;      /* message header buffer     */
  word    Timeout;     /* message timeout           */
  word    *Control;    /* pointer to control buffer */
  byte    *Data;       /* pointer to data buffer    */
} MCB;
```

MsgHdr is defined as follows :

```
typedef struct MsgHdr {
  unsigned short  DataSize;   /* 16 bit data size            */
  unsigned char   ContSize;   /* control vector size         */
  unsigned char   Flags;      /* flag byte                   */
  Port            Dest;       /* destination port descriptor */
  Port            Reply;      /* reply port descriptor       */
  word            FnRc;       /* function/return code        */
} MsgHdr;
```

PutMsg() and **GetMsg()** respectively transmit and receive the contents of the MCB on the ports defined within the message header. A timeout is associated with the implementation of both routines. If the message header is not delivered or received within the specified duration, the entire transfer fails. Message passing is described more completely in chapter 9, *The Kernel*.

A simple example of message passing is given in appendix C. The program is designed to measure the rate of data transmission between two processes (a sender and a receiver) which communicate using the **PutMsg()** and **GetMsg()** primitives. The connection between the sender and receiver is established using raw sockets in the typical asymetric client/server fashion. The receiver (server) creates a socket, binds an address to it and it then listens for and accepts a connection from the sender (client).

The processes then loop for a number of iterations, transmitting a series of messages of varying lengths to each other.

It is useful to isolate and identify the components of the program that are involved in the message passing process. As shown below, these components are actually simple. The sender and receiver both make use of two MCBs, **txmcb** and **rxmcb**, the contents of which are sent and received on the ports identified by **tx** and **rx**. The mapping between the respective destination and reply ports is established as follows :

```
/* Sender */                          /* Receiver */
tx = fdstream(sock)->Server ;      tx = fdstream(msg_sock)->Server ;
rx = fdstream(sock)->Reply ;       rx = fdstream(msg_sock)->Reply ;
```

The control and data vectors within the MCBs must be initialised prior to use, and, in particular, the destination and reply port descriptors set. This is performed using the **InitMCB()** procedure :

```
InitMCB (&txmcb, MsgHdr_Flags_preserve, tx, NullPort, 0) ;
InitMCB (&rxmcb, MsgHdr_Flags_preserve, rx, NullPort, 0) ;
```

InitMCB() also sets the value of **MCB.Timeout** to **IOCTimeout** (20 seconds).

The data field of the MCB is a pointer to a data buffer. This buffer either contains the information to be transmitted, or is used to store the incoming message. The data and size fields must be set accordingly :

```
txmcb.Data =
rxmcb.Data = buf ;
rxmcb.MsgHdr.DataSize =
txmcb.MsgHdr.DataSize = buf_size ;
```

Finally, pointers to the respective MCBs are passed to **PutMsg()** and **GetMsg()** to effect the data transfer :

```
PutMsg(&txmcb) ;
GetMsg(&rxmcb) ;
```

As stated above, it is necessary to build protocols on top of the message passing primitives to provide fault tolerance and reliability. These protocols do, of course, impose processing overheads. The effects of these overheads on the performance of Helios are examined in the following section.

6.2 Performance

The raw performance of multi-processor message passing machines is dependent upon the computational speed of each processor, and the bandwidth and latency of inter-processor communication. The provision of operating system services, such as transparent message routing, imposes some degree of processing overhead. There is an obvious and unavoidable trade-off between functionality and performance – aspects such as ease of programming and portability can only be provided at the risk of obscuring the available processing power.

This chapter examines the extent to which the functionality of Helios affects the two performance criteria indicated above – computation and communication. These

aspects are investigated by a series of experiments designed to evaluate the overhead of running computationally and communications intensive application programs under Helios. It is shown that Helios imposes a negligible amount of computational overhead, and that the message passing mechanism is highly efficient.

Computational performance is evaluated using the standard Whetstone and Dhrystone benchmarks. The communications performance is examined at the various levels provided under Helios (from low-level message passing primitive operations through to Posix library function calls). At each level, the respective inter-process data transmission rates are analysed, illustrating the gains in performance that can be attained at the expense of functionality and portability. The effects of routing messages through intermediate nodes is also considered, highlighting the efficiency of the Helios message passing mechanism.

Finally, the extraction of performance data from the Helios Kernel is described for users who desire to obtain run-time performance statistics.

Helios runs on a wide variety of different processors and communications hardware. Performance figures will, quite naturally, differ with respect to specific hardware technologies. The aim here is to help Helios users achieve the maximum from whatever hardware technology is being used. The performance analysis conducted in this chapter pertains to a typical system against which other systems may be easily compared.

6.2.1 Test conditions

The benchmarks given in this chapter were performed using medium performance Transputers (20MHz IMS T800C-G20S with access to 1 Mbyte of 4 cycle (200 ns) external RAM). The link speeds of the processors were set at 20 Mbits/second. For systems using newer technologies with faster processor speeds, memory interfaces, or better communication bandwidth, the benchmark figures may be extrapolated or repeated using the test methods described in this chapter.

The processor network used for performance testing comprised a root Transputer linked to a pipeline of Transputers. The test programs were executed on the pipeline, and the root Transputer was responsible for running various system services. The programs were all coded in C, compiled using version 2.01 of the Helios C compiler, and run under Helios version 1.2.1.

6.2.2 Computational benchmarks

The computational benchmarks used consist of the standard Dhrystone and Whetstone tests. Although the merits of these tests are debatable for measuring processor power, they are useful in demonstrating techniques for improving speed. Each benchmark was repeated four times and each time the level of performance was increased by adding an extra enhancement.

Test 1 Standard benchmark with stack checking.

Test 2 Stack checking disabled.

Test 3 Fast on-chip RAM used (Transputer feature).

Benchmark	Test 1	Test 2	Test 3	Test 4
Whetstones	1,205,200	1,219,600	1,636,300	1,638,500
Dhrystones	3,600	3,868	5,555	5,539

Table 6.1: Computational benchmarks

Test 4 Standalone environment (no Nucleus).

The tests shown in Table 6.1 illustrate the relative effects on the performance of a program using different environments, and should only be taken as the benchmarks for the specific type of hardware used in the test.

Stack checking is performed on the entry of each new procedure. Programs which call procedures infrequently will have little benefit from the removal of the check. The dhrystone benchmark attempts to be a representative program in the ratio of procedure calls to computation. Removing stack checking from this program (Test 2) improved the speed by just under 8%. This would only be recommended for fully debugged programs. Unchecked stack overflow could potentially destroy system memory. If a program uses any recursive functions then it is probably best never to disable stack checking.

Fast 'on-chip' RAM is a feature of most Transputers and may be utilised from Helios by using the **Accelerate()** system routine. This places the stack of a function in fast memory. Prior to use, the memory must be allocated using **AllocFast()**. The performance increase can often be dramatic. Fast memory is a scarce resource, however, and should be used carefully to obtain optimum performance. Normally only computationally intensive functions should be accelerated. For all fast memory test results in Table 6.1, Test 3, 1.5 Kbytes from the available 4 Kbytes of fast memory was used. Not all processors which run Helios have fast memory. To ensure portability of the code, calls like **Accelerate()** should be conditionally compiled.

Test 4 runs the benchmark programs on a standalone processor. This is a processor which does not run the Helios Nucleus and is usually called a **native node**. Library functions are linked into the program by a standalone linker and the resulting code is booted into native nodes by a bootstrap utility. There are several reasons why a standalone may be used: there may be only a limited amount of memory, the program may need total control of all processor resources, or the user may want maximum performance.

The benchmark figures show that only a very small performance increase can be obtained, less than 0.2%! The conclusion from this is that the basic operating system maintenance takes little computational power away from an application. The main reason why this figure is so low is because scheduling is effectively performed by the Transputer hardware rather than by a software executive. This may not be the case for other processors. However, all processor technologies running as standalone will benefit from single-thread execution and the absence of message routing tasks.

6.2.3 Communication benchmarks

A limiting factor in efforts to achieve high performance and processor efficiency on processor networks is that of communication. The rate of computation is significantly higher than that of communication – communication is consequently a potential performance bottle-neck. It is therefore important that the overhead imposed by an operating system on communications performance be kept to a minimum.

Helios provides four levels of communication. The lowest level is used in the Nucleus: **PutMsg()** and **GetMsg()** are the message passing primitives that provide the basis of all Helios communication. The level above this is provided by System library functions **Read()** and **Write()**. These calls operate on streams and have timeouts associated with each requested transaction. At the next level are the Posix **read()** and **write()** functions. Calls to the Posix functions are based on Posix file descriptors. The highest level of communication is at the language level; here, the range and calling conventions depend on the programming language used.

In this section, communication performance is investigated at the message passing, System and Posix levels. At each level, the respective rates of inter-process data transfer are evaluated between processes residing on

1. Adjacent processors.
2. Processors separated by varying link distances.
3. The same processor.

To provide a comparison for the respective performance figures, the experiments were also implemented using direct link utilisation (that is, directly transmitting messages over the physical links through in-line calls to assembler macros). For the sake of clarity, the following notation is used to refer to each of the above message passing mechanisms :

Direct	Direct link usage using in-line assembler macros.
Primitive	Message passing primitives (**GetMsg()** and **PutMsg()**).
System	System library functions (**Read()** and **Write()**).
Posix	Posix library functions (**read()**, **write()**).

The full code listings and results of the experiments conducted in this section are located in appendix C.

As described previously, the test network was configured in a pipeline, and a task force of two components (the sender and the receiver) was used to measure the time to transmit a message to a node and reflect it back again. The need to utilise two-way transmission is evident when one considers that the clocks on the individual processors are not synchronised. It is consequently not possible to obtain a direct measurement of the time taken to perform uni-directional message transmission. Two-way message transmission was therefore performed, and the total transmission time halved. This is illustrated in Figure 6.1.

Inter-processor communication

Inter-processor message transmission times over Transputer links are characterised by a relationship of the form :

message time from P_0 to $P_{n-1} = \frac{tstart - tend}{2}$

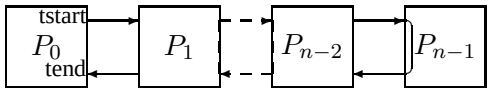


Figure 6.1: Bi-directional message transmission

t_{total} = t_{init} + N.t_{tx}, where

t_{total} = message transmission time
t_{init} = message initialisation or setup time
N = number of bytes in message
t_{tx} = transmission time for 1 byte

The t_{init} is independent of the message length. The number of processor cycles required to transmit a message comprising w words is 2w + 19. A T800 Transputer running at 20 MHz executes 19 cycles in 0.95 microseconds – this figure is consequently a lower bound for t_{init}. A minimum value for t_{tx} can likewise be established. The maximum uni-directional data rate attainable on 20 Mbit/second Transputer links is 1740 Kbytes/second. This corresponds to a t_{tx} value of 0.5612 microseconds.

In this section, values for t_{init} and t_{tx} are derived with respect to each of the Helios communication mechanisms. It will be seen that, at each level of communication, t_{tx} is closely related to the physical link bandwidth. The value of t_{init}, however, increases with the level of communication – there is a definite trade-off between performance and functionality. To investigate the performance of data transmission between two directly connected processors, the sender and receiver processes were placed on adjacent nodes in the pipeline. The transmission times for messages of various sizes are shown in Table 6.2. The rate of data transmission in Kbytes/seconds is also depicted.

The transmission times listed in Table 6.2 are characterised by the following relationship :

t_{total} = $t_{overhead}$ + t_{init} + N.t_{tx}, where

$t_{overhead}$ = loop overhead on each test iteration
= 1.686 microseconds

Linear regression can be applied with respect to the values of t_{total} and N given in Table 6.2 to derive a relationship of the form t_{total} = A + BN. These relationships are shown below, with the correlation coefficient (r) obtained in each case :

Direct	t_{total} =	3.869263 +	0.563853.N	(r = 1.000000)
Primitive	t_{total} =	126.120129 +	0.562684.N	(r = 1.000000)
System	t_{total} =	1446.708396 +	0.567681.N	(r = 0.999956)
Posix	t_{total} =	1461.421142 +	0.567642.N	(r = 0.999956)

	Direct		Primitive		System		Posix	
N	t_{total}	Rate	t_{total}	Rate	t_{total}	Rate	t_{total}	Rate
1	5	195	125	7	1430	0	1440	0
2	10	195	125	15	1405	1	1415	1
4	5	781	130	30	1400	2	1415	2
8	10	781	130	60	1405	5	1420	5
16	10	1562	135	115	1410	11	1420	11
32	20	1562	145	215	1410	22	1425	21
64	40	1562	160	390	1420	44	1435	43
128	75	1666	195	641	1435	87	1450	86
256	145	1724	270	925	1465	170	1480	168
512	290	1724	415	1204	1675	298	1700	294
1024	580	1724	705	1418	2185	457	2200	454
2048	1160	1724	1280	1562	2760	724	2775	720
4096	2315	1727	2430	1646	3915	1021	3925	1019
8192	4625	1729	4735	1689	6215	1287	6230	1284
16384	9245	1730	9350	1711	10825	1478	10845	1475
32768	18480	1731	18565	1723	20040	1596	20055	1595
65535	36955	1731	37000	1729	38605	1657	38615	1657

Table 6.2: Data transmission times and rates with respect to message size

N = message size (bytes)
t_{total} = transmission time (microseconds)
Rate = transmission rate (Kbytes/second)

Level of communication	t_{init}	t_{tx}
Direct	2.18	0.5639
Primitive	124.43	0.5627
System	1445.02	0.5677
Posix	1459.74	0.5676

Table 6.3: Derived t_{init} and t_{tx} values (microseconds)

Table 6.3 shows the corresponding values for t_{init} (given by A – $t_{overhead}$) and t_{tx} (equivalent to B). The value t_{tx} closely relates to the bandwidth of the interconnecting hardware. It is interesting to note that the lowest t_{tx} value is associated with **Primitive** and not **Direct** communication as one would expect – the difference is, however, negligible. As stated previously, the maximum uni-directional data rate that can be attained on 20 Mbit/second Transputer links is 1740 Kbytes/second. Helios achieves a maximum transfer rate that closely approaches this physical maximum – 1729 Kbytes/second.

To place the performance figures of Table 6.2 in perspective, the respective rates of transmission are shown graphically in Figure 6.2. In this graph, and where applicable to the other graphs in this section, the **Posix** curve is plotted with the **System** curve as the respective coordinates are almost identical.

These graphs exhibit a number of observable features upon which some comment is appropriate. Firstly, it is evident that although the basic shape of the curves is similar, they differ widely with respect to their displacement along the horizontal axis. Obviously, this is directly attributable to the respective t_{init} (constant) components associated with each of the functions. Secondly, it can be seen that the transmission of

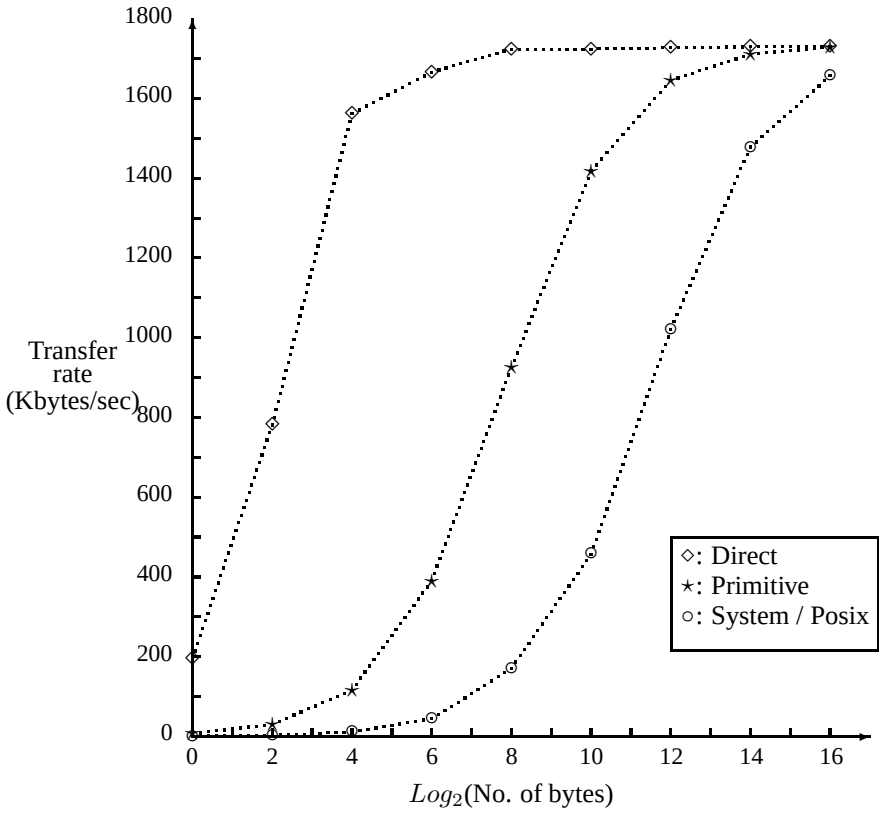


Figure 6.2: Rates of data transmission with respect to message size

small messages is inefficient. As previously indicated the value of t_{init} is independent of the message size; consequently, it is more efficient to transmit a single large message than a number of smaller ones. The third noticeable attribute of the graph is that the respective rates of transmission all converge towards a maximum rate. This maximum rate is dictated by the physical link bandwidth, and it can only be approached when the messages are sufficiently large for t_{init} to become negligible. It is interesting to note that, with respect to messages of this size (16K or more), the actual communication mechanism is immaterial; there is little difference between the use of message passing primitive operations and higher-level library routines.

The effect of the C004 link switch on data transmission

The IMS C004 link switch is a programmable crossbar switch that can be used to electronically configure Transputer networks. Routing data through the C004 (as opposed to utilising directly connected Transputer links) incurs a 1.75 bit time delay. It is of interest to ascertain the effect of the C004 on the performance of the Helios communication mechanisms. Figure 6.3 illustrates the rates of data communication (Kbytes/second) attained using message passing primitives (**PutMsg()** and **GetMsg()**) between two Transputers that were

1. Directly linked and
2. Connected through a C004 link switch.

It is evident from Figure 6.3 that the effect of the C004 link switch on the rate of communication is far from negligible. The overhead imposed by the link switch increases with the size of the message. In the worst case (64 Kbyte message), transmission through the C004 is 23 % slower than sending data over directly connected links.

Through-routing message transmission

Another important aspect of inter-processor communication is the effect of routing messages through intermediate nodes. Figure 6.4 shows the rates of communication associated with the transmission of a 64 Kbyte message from a source to a destination node through varying numbers of intermediate processors.

The **Primitive** and **System / Posix** curves indicate a rather steep drop in performance as soon as messages are through-routed through a second processor (2 intermediate links). However, the addition of further intermediate links does not lead to significant performance degradation.

Conversely, the appalling performance exhibited by the **Direct** curve as more links are traversed is striking, and requires some comment. The test program used to evaluate the performance associated with direct link usage over intermediate nodes was implemented such that each node would receive the message in its entirety before sending it on. It is a well established fact that “pipelining” message transmission through intermediate nodes has a tremendous effect on the overall rate of transmission. This technique entails breaking up the message into smaller packets which are then received and transmitted concurrently at each intermediate node (this is made possible by the

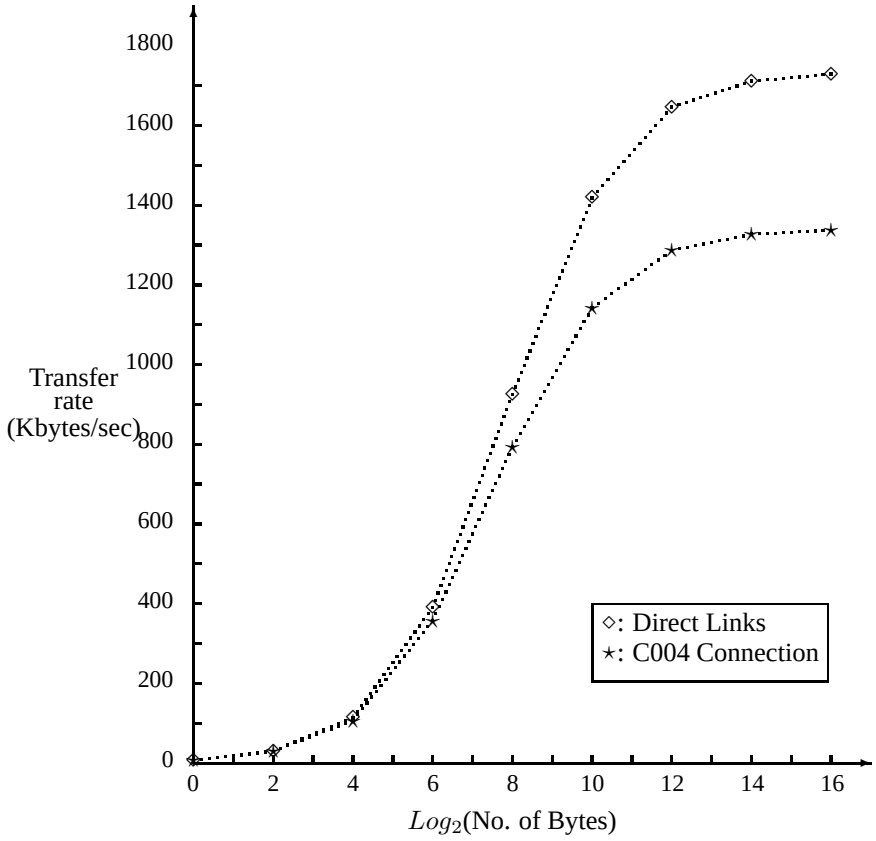


Figure 6.3: Effect of C004 link switch on rate of data transmission

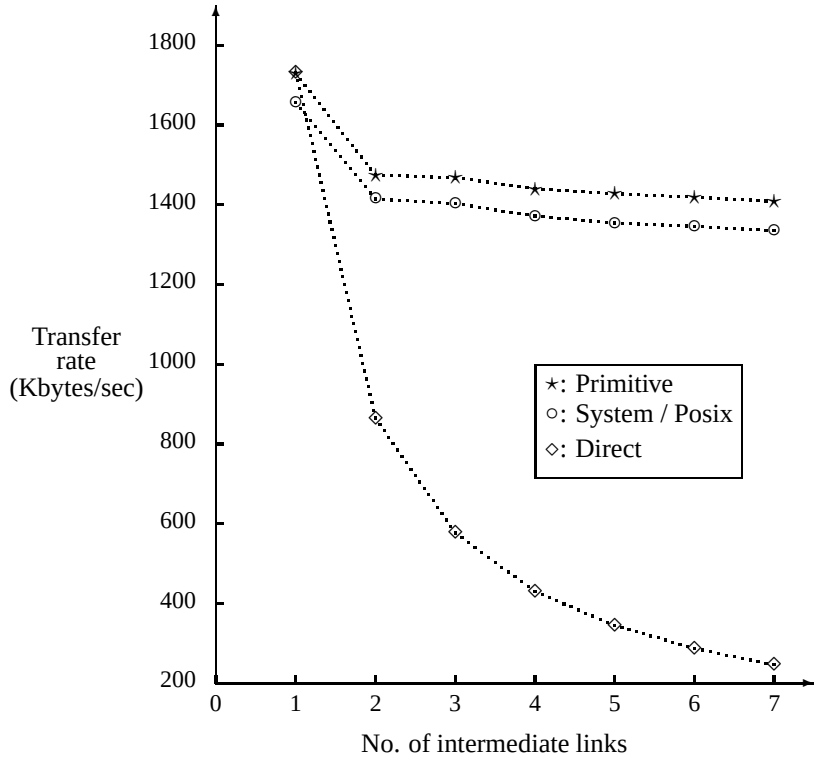


Figure 6.4: Rates of data transmission (64 Kbyte message) with respect to number of intermediate links

Message Size (bytes)	System		Posix	
	Software Channels	Hardware Links	Software Channels	Hardware Links
1	0	0	0	0
2	1	1	1	1
4	2	2	2	2
8	4	5	4	5
16	9	11	9	11
32	19	22	18	21
64	37	44	37	43
128	75	87	74	86
256	148	170	145	168
512	288	298	283	294
1024	529	457	520	454

Table 6.4: Rates of transmission using system and posix routines (Kbytes/second)

fact that Transputer links have their own DMA controllers and hence can operate in parallel). Message pipelining was, however, deliberately not implemented. Instead, the objective was to provide a comparison against which the effect of message pipelining as implemented (transparently) under Helios could be observed.

The results depicted in Figure 6.4 are largely self-evident. Helios takes full advantage of message pipelining by splitting up messages into packets of an optimal size (the minimum transfer size that is cost effective in terms of link setup overhead). The Kernel then reads in new packets at the same time as sending on the previous ones. The packets are received and transmitted concurrently by the intermediate nodes.

Internal communication

Pipes not only form a communication channel between processes on different processors, but also between processes on the same processor. The programmer interface is identical but the transfer medium is shared memory rather than hardware links.

Figure 6.5 shows the rates of data transmission attained using the **GetMsg()** and **PutMsg()** primitives between two processes residing on

1. The same processor (communicating through memory).

2. Adjacent processors (communicating over physical links).

As one would expect, the bandwidth is dramatically higher (9.23 Mbytes per second maximum) for message transmission through local memory. However, what is gained in transfer rate may be lost if both communicating processes are competing for the computational power of the processor.

This is clearly evident with regard to the transmission of small messages using the higher-level System and Posix calls. Data transmission rates between processes residing on the same and adjacent processors attained through the System (**Read()** and **Write()**) and Posix (**read()** and **write()**) library routines are shown in Table 6.4.

These figures indicate that, for messages of less than 1 Kbyte, it is faster to communicate over a physical link than through memory. This is due to the implementation

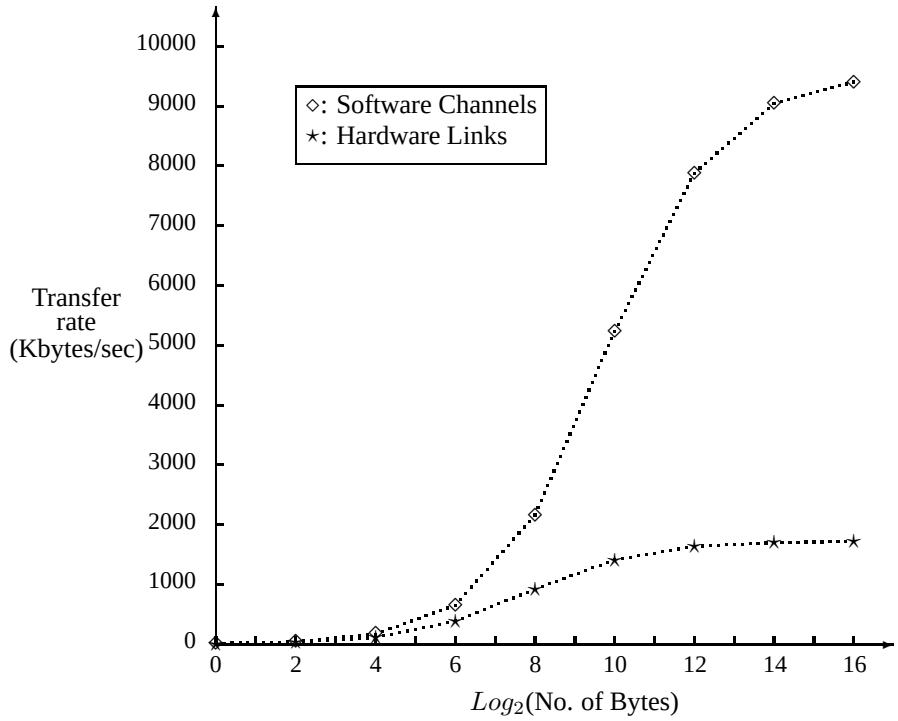


Figure 6.5: Rates of data transmission over software channels and hardware links

of the pipe protocol. The operation of this protocol requires some degree of computational overhead, carried out at each end of the pipe by separate system threads. Therefore the passage of data through the pipe requires several suspensions and resumptions of both the user and pipe worker processes through semaphores and message exchanges. When the ends of the pipe are on different processors much of this protocol processing can be carried out in parallel by the two processors. However, when both ends are on the same processor, they must share CPU time and therefore get in each other's way. Since the relative protocol processing cost for small data transfers is large anyway, this only makes things worse, hence the lower performance for local pipes. However, as the data transfer size increases, the relative cost of the protocol processing becomes smaller and the local pipe soon overtakes the remote one as a result of the higher message transfer rate.

Observations

The Helios communications mechanism is highly efficient. There is little overhead associated with the use of primitive message passing operations, and it is possible to achieve rates of data transmission that approach the limits imposed by the link hardware.

For applications that require fault tolerant communications, Helios provides higher-level library routines with built-in error detection and recovery. Although these routines impose a communications overhead, this is not prohibitive. Indeed it has been shown that, with regard to the transmission of large messages, the performance associated with the respective communications mechanisms is fundamentally identical.

Finally, the implementation of data transmission through intermediate nodes takes full advantage of the parallel capabilities of the communications hardware.

The decision concerning which communication mechanism to use is obviously application specific. It is important, however, to bear in mind the trade-offs between performance and functionality.

6.2.4 Obtaining performance data from Helios

During normal operation, Helios monitors its own performance and maintains a record. Users wishing to access run-time performance statistics may do so with a **ServerInfo** request to the processor's **tasks** directory. The following procedure can be used to retrieve the relevant information to a named processor :

```
int getstats(char *name, ProcStats *pstats)
{
        char pname[100];
        Object *proc;
        int err;

        strcpy(pname,"/");
        strcat(pname,name);
        strcat(pname,"/tasks");

        proc = Locate(NULL,pname);
        if( proc == NULL ) return FALSE;
```

```
        err = ServerInfo(proc,pstats);
        Close(proc);
        return err>=0;
}
```

On successful completion, the **ProcStats** structure will have been filled with status information from the processor. This structure, which may be found in <**root.h**>, comprises the following fields :

Type The processor type. Currently 414, 425, 800, 801 or 805.

Load The average load on the processor. This ranges from 0 to about 2000 and it is a time averaged estimate of the number of microseconds for which each currently running process has been scheduled.

Latency A time averaged estimate in microseconds of how long a high priority process would wait on the run queue.

MaxLatency The maximum latency seen.

MemMax The total size of the available system memory pool in bytes.

MemFree The number of bytes currently available in the system memory pool.

LocalTraffic The total size of all messages exchanged between local processes.

Buffered The total size of all messages buffered pending delivery by the Kernel.

Errors The number of times the processor error flags have been set.

Events The number of times the processor event channel has been triggered.

NLinks The number of links this processor has, and the number of structures in the following **Link** vector.

Link A vector of **NLinks** sixteen-byte structures, each of which contains the following fields:

Conf is a LinkConf structure showing the state of the link.

In is the number of bytes received from the link.

Out is the number of bytes transmitted through the link.

Lost is the total sizes of all the messages received but destroyed as a result of congestion.

Name The current network name of the processor. The exact offset of this field depends upon the value of **NLinks.**

Chapter 7

The Resource Management library

7.1 Introduction

This chapter describes the Helios Resource Management library. The purpose of this library is to make a Helios network of processors readily accessible to advanced application programmers, and in particular to permit task forces to be run within the network. It can be considered to be an extension to the Helios CDL language, giving more control to any applications that need it. The chapter assumes familiarity with Helios networking and the CDL language. Typical jobs that can be performed with the Resource Management library include:

1. The examination of the current network, and the use of the information provided, for example by displaying it on a screen.
2. The construction of the binary resource map used by the Helios networking software.
3. The construction and execution of a task force.
4. The mapping of a task force onto the current network, and its execution.

7.2 The Resource Management library

There are many different ways of looking at computer systems. One approach treats the hardware as a set of facilities or **resources**. These are controlled by **system software**, which makes them available to end-user **application software** and hence to the users of the system.

For example, consider a hard disc. At one level, a hard disc consists of one or more platters coated with a suitable magnetic material, spinning around a common axis, together with a number of heads capable of detecting and changing the magnetic state of the platters. At a slightly higher level, a hard disc consists of a number of disc blocks, subdivided into sectors and tracks, with each disc block holding typically 512 bytes of data. The application programmer sees the disc as a hierarchical collection of files. Depending on the application the end user might see word processing documents, pretty pictures of an engineering design, or a speadsheet for the company's

accounts. System software sits between the hardware and the application program: it is responsible for taking the collection of disc blocks, turning them into a hierarchical filing system, and making this filing system readily accessible to application programs. This is illustrated in Figure 7.1.

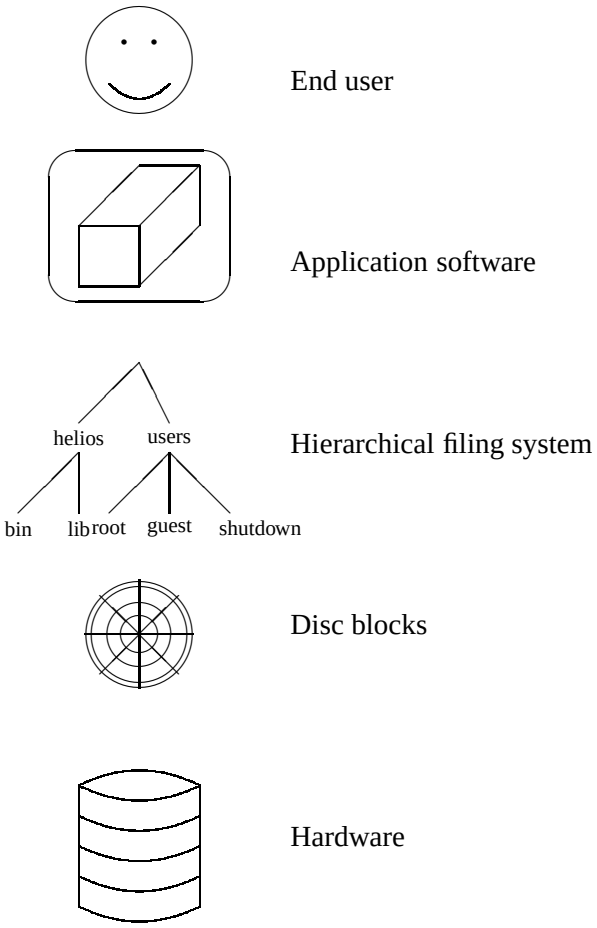


Figure 7.1: The system hierarchy

When designing an operating system interface for hardware, two things have to be considered. First a model is needed, which abstracts away from the hardware without losing hardware functionality. Second a set of library routines is needed to allow access to this model. Again consider the hard disc. The abstract model defines two main primitive objects (files and directories), which can be manipulated by application programs. This means that application programmers lose some functionality, for example they cannot control where on the disc their data is held nor optimise the disc organisation for their application. On Unix systems the appropriate library calls are **open()**, **read()**, **write()**, **close()** and so on. The exact implementation of the system software,

the filing system itself, is not important provided it implements the model and provides the library routines. Details such as cache size and the disc block allocation strategy affect performance, but not (usually) the application programmer.

Where appropriate the system software should implement existing standards. Thus the interface to a hardware graphics display should usually be the X window system, unless there are very good reasons why this is unsuitable. Other graphics standards such as GKS can be implemented on top of X. The interface to a processor's memory is provided by the routines **malloc()**, **free()**, and **realloc()**. The interface to a filing system is provided by the Unix file I/O routines.

In multi-processor or distributed processor machines there are two relatively new hardware facilities: the network of processors, and the communication hardware between the processors. The purpose of processors is to run programs, so the system software must provide some way of running programs on the various processors in the network. The purpose of communication hardware such as an ethernet or Transputer links is to let programs communicate with each other. The Resource Management library is the interface provided by the system software for the first of these, control over the network of processors.

It can be argued that existing Unix standards already have library routines to do the work of the Resource Management library: **fork()**, **execve()** and so on. However, these routines are designed for single- processor systems rather than the target hardware of multi-processor and distributed processor machines. Controlling a collection of processors is rather more complicated than controlling a single processor, and application programmers need much more flexibility than is provided by the Unix routines.

7.2.1 The abstract model

As with all system software an abstract model is required to model the hardware: the network of processors. The Resource Management library defines seven basic primitives.

1. **Processor**: something which can do work.
2. **Network**: a collection of processors.
3. **Link**: a connection between processors.
4. **Task**: a program which can be run on a processor.
5. **Task force**: a collection of tasks.
6. **Channel**: a connection between tasks.
7. **Session**: a collection of tasks and task forces.

Processors

In most networks it is fairly obvious what the processors are: the Transputer, the i860, the 680x0, etc. They have certain characteristics:

1. Processors are of a given type, for example they may be T800, i860 or a number of other types.

2. Processors have a purpose. Some processors may be reserved for use by the system, while others are available to users' sessions. Some processors may be dedicated to performing I/O or routing messages. Some processors run the Helios Nucleus in addition to the user programs, relying on the system software to control message routing, memory management and so on. Other processors such as **native** processors, for example, contain no system software at all and leave the application full access to all the processor's hardware.

3. Processors have a certain amount of memory. More interestingly, they have an **address space**. For processors without virtual memory, including Transputers, this address space is the amount of physical memory attached to the processor, for example four megabytes. For processors with virtual memory four gigabytes is more likely.

4. Processors have a current state. For example, a processor may have crashed recently because an application program wrote over the wrong piece of memory.

5. A processor should have at least one piece of hardware allowing it to communicate with other processors. This hardware may or may not be connected at any one moment in time.

6. Processors may currently be allocated to a user, or they may be in a free pool.

7. Processors usually have a logical id. This logical id can be separate from the physical id which specifies where in the system the processor resides.

8. The system software may have partial or full control over the processor. In particular, depending on the hardware in use the system software may or may not be able to reset processors or reconfigure the connections between the processors.

9. Processors may have any number of other characteristics, for example an attached signal processing chip.

For example, consider the simple network in Figure 7.2.

1. Most of the processors are the T800 type, except for **/IO** which is an 80286.

2. Most of the processors are available to users. **/IO** is dedicated to performing I/O operations only. **/00** may be reserved for system software.

3. All processors of the T800 type have two megabytes of memory.

4. All the processors are currently running normally.

5. The T800 processors each have four links allowing them to communicate. The I/O processor has a single link allowing it to communicate with just one other processor.

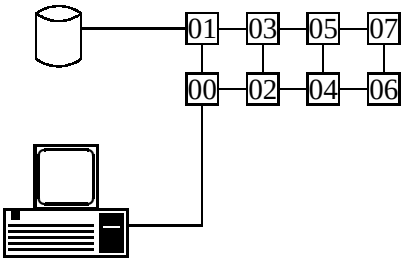


Figure 7.2: A simple network

6. In a single-user network like the one shown it is likely that most processors are permanently allocated to one user.

7. The processors' logical ids are the strings **IO**, **00**, **01** and so on.

8. Processor **01** has an attached SCSI hard disc unit.

Networks

Some sort of data structure is needed to collect processors together. This is known as a **network**. In theory a network can contain any number of processors. It must be emphasised that the term **network** does not necessarily refer to every processor in the system. For example, if a particular user currently has processors **00**, **01** and **02** allocated then these three processors form a network: they are a collection of processors.

The majority of networks will not contain any form of hierarchy: there is just one network, and all relevant processors are inside this network. However at times it may be convenient to subdivide a collection of processors into smaller groups or subnetworks. This can be achieved by having subnetworks, as shown in Figure 7.3. For example, if the machine contains hardware produced by various different manufacturers with incompatible reset schemes then it may be desirable to have separate subnets for each scheme.

Links

In the Resource Management library a **link** is simply a connection between two processors. Links can take many forms, including:

1. A high-speed serial RS232 line.

2. Shared memory between two processors, for example between an i860 and a Transputer.

3. A connection into a hardware routing network.

4. An internet TCP stream socket across an ethernet.

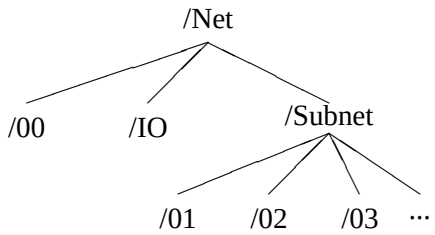


Figure 7.3: Subnetworks

5. A lower-level ethernet connection permitting broadcasts and so on.

6. A Transputer link.

For example, consider a Transputer which has a shared memory interface to an i860. The i860 has just one link, the shared memory. The Transputer has five links, the four Transputer links implemented on-chip plus the shared memory. The first four links will be quite easy to use, whereas the fifth link will need some extra software.

Connections between processors can be divided into two types. Some connections are point-to-point, for example a Transputer link, a TCP socket, or an RS232 line. Other connections allow a processor to communicate with many other processors using a single piece of hardware. For example routing chips allow every processor to communicate with every other processor attached to the router. An ethernet can allow every processor to communicate with every other processor attached to the ethernet. Representing point-to-point connections only is relatively easy, as the resulting network is a simple graph with the processors as vertices and the links as edges. This is shown in Figure 7.4.

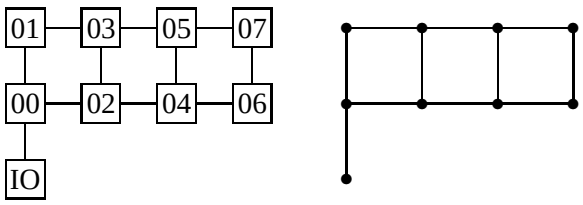


Figure 7.4: A network as a graph

Representing many to many connections is more complicated. One approach is to treat a routing network as a large number of point-to-point connections. Thus if the router connects sixteen processors then each processor might be considered to have fifteen point-to-point connections to other processors. This does not work for large

numbers of processors, because the number of connections becomes unmanageable. An alternative approach is to treat a router as a special type of processor. All the 'real' processors have one connection to the router processor, and the router processor has many connections to many processors. This is shown in Figure 7.5.

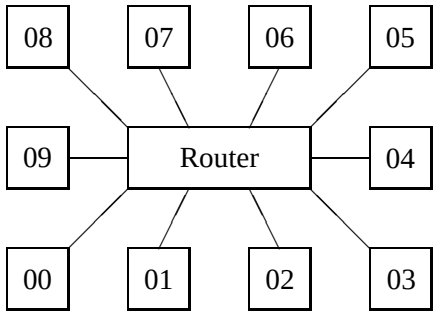


Figure 7.5: A network with a router

Some processor connections may go outside the hardware administered by the Resource Management library. For example, if a network is based around an ethernet, there may be a gateway to another network, and to the external world as a whole. To cope with this the library has the concept of **external** links.

Tasks

Processors exist in order to run applications. This means that it is necessary to have a representation of applications, and in particular applications that can be distributed over a network of processors. Applications consist of one or more **tasks**, communicating with each other to achieve parallelism, and interacting with the outside world to obtain data and produce results. A single task runs on a single processor, but a processor may run more than one task.

Like processors, tasks have certain characteristics.

1. They are compiled for a particular processor or set of processors. For example, a task could be compiled to run on just T800 processors, or on any Transputer. In future, with the advent of architecture neutral distribution formats, it may be possible to have binary files that run on any processor.

2. Tasks are associated with a specific piece of code, usually held in a filing system. Each distinct task may be associated with a separate file. Alternatively a single file might hold the code for all the tasks in a task force.

3. Tasks may run alongside system software, which means they get the benefits of automatic message routing, memory management and so on. Alternatively tasks may run on native processors with full access to all the hardware facilities

and the associated problems. If native tasks are used, only one task can run on a processor. If system software is present, this may support multiple tasks running on a single processor.

4. Tasks have a certain memory requirement. This may be so small that it can be ignored. Alternatively a task may need most of a processor's memory, preventing that processor from running any other tasks.

5. Tasks may have specific hardware requirements, for example they may need a particular signal processing chip.

Task forces

To exploit multiple processors it is necessary to have multiple tasks distributed over the processors. As with processors a data structure is needed to collect tasks together, and this structure is known as the task force. As with networks, a hierarchy of sub task forces is permitted. For example, Figure 7.6 represents a task force in the form of a farm. There is a master task, a load balancer task, and there are a number of worker tasks.

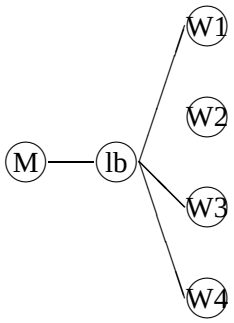


Figure 7.6: A farm task force

Channels

Just as a processor is connected by one or more links to other processors, a task is connected by one or more **channels** to other tasks. Like links, channels are numbered from 0 onwards. The channel numbers are in fact equivalent to Unix file descriptors. Hence channel 0 corresponds to Unix's **stdin** stream, channel 1 corresponds to **stdout** and so on. Some channels may be used for interaction with the outside world, for example to write data to a disc, and such channels are known as external channels. In Figure 7.6 there are two channels between the master and the load balancer, one for communication in each direction. There are also two channels between the load balancer and each worker.

Sessions

The Resource Management library assumes that the processor network may be shared by multiple users. Every processor running system software can run multiple tasks, with the system software taking care of sharing the processor between the tasks. At present the library assumes that a processor cannot be shared by more than one user, but this restriction may be relaxed in a future version. If a network can have multiple users it is necessary to define some things about a user's session, and the way that the processors are shared.

1. The Resource Management library is used to control a fixed number of processors in the machine. There may be additional processors but these are not accessible through the library. The collection of processors administered by the library is known as the **Network**, with a capital letter N.

2. At any one moment in time a subset of the Network may have been allocated to a particular session, and this subset is known as the session's **domain**. A processor can be in only one domain.

3. Processors can be added to a user's domain at any time. They can also be removed from the user's domain provided they are not currently running any application software.

4. Some processors in the Network may be reserved for use by the system software and hence cannot be allocated. For example, a processor running a file server would normally be reserved to protect the file server program from accidental or deliberate corruption. Any processors not reserved and not currently allocated to a domain are in the free pool.

5. A user may run one or more applications inside the domain of processors associated with their session. A processor may be used by more than one application.

In future there may be some extensions to the Resource Management library. These might include routines to create and abort sessions, and to perform various types of accounting.

7.3 Outline of the library calls

This section outlines the calls provided by the Resource Management library. There are a considerable number of these calls, so they are grouped together in the following sections:

1. Programming conventions.
2. Constructing a network.
3. Examining a network.
4. Obtaining a network.
5. Constructing a task force.
6. Examining a task force.
7. A program's environment.
8. Executing a task.
9. Executing a task force.
10. Modifying a network.
11. File I/O.
12. Error handling.
13. Miscellaneous.

The sort of job for which the Resource Management library might be used is as follows. There is an existing application in the form of a task force, written in Fortran or another language. It is necessary to execute this task force subject to certain restrictions, and in particular the task force must be mapped carefully onto the existing network. One approach is to use a CDL script describing the task force, but this may not suffice for a variety of reasons. Instead the user can write an additional control program, linked with the Resource Management library, to run the task force. This program can be as simple or as complicated as required, and supersedes the CDL compiler.

7.3.1 Programming conventions

The Resource Management library must not conflict with any existing libraries, either System libraries or User libraries. In particular it is essential that the library does not use any routine names that clash with existing names, thus preventing the library from being linked with existing programs. To do this various conventions must be adopted, and these are described in this section. Only a C interface to the library is defined in this chapter, although interfaces to other languages should be straightforward. For C programmers there is a new system header file:

```
#include <rmlib.h>
```

All routine names start with the two letters **Rm**, to guarantee that the name is unique. These two letters are usually followed by a verb specifying the operation to be performed, starting with a capital letter. This verb is followed by a noun indicating the kind of object on which the operation is to be performed. There may be an additional field if more information is needed. For example, the routine to find out how much memory a particular processor has takes the following form:

```
unsigned long int RmGetProcessorMemory(RmProcessor);
```

Similarly, the routine to break a processor link is:

```
int RmBreakLink(RmProcessor, int);
```

All constants defined by the library take a similar form. They start with the two letters **Rm**, then an additional letter indicating the nature of the constant, an underscore, and the actual name. For example, the error code indicating that one of the arguments passed was an invalid link number is:

```
#define  RmE_BadLink     <some number>
```

Similarly, the constant indicating a processor type takes the form:

```
#define  RmT_T9000          <some number>
```

Similar naming conventions have been adopted in the past, for example by the X window system. All X library routines and constants start with the letter **X**. X toolkit routines start with **Xt**, and the X Athena Widget library uses **Xaw**. The library has four main data structures: for processors, networks, tasks, and task forces. The layout of these data structures is not known to the programmer, it is internal to the library. The library header file defines four data types which are probably, but not necessarily, pointers to the internal data structures.

1. **RmProcessor**
2. **RmNetwork**
3. **RmTask**
4. **RmTaskforce**

The library has its own error codes, independent of the error codes used by other systems. It is based on the Unix error code model: there is a global variable **RmErrno**, which is set to one of a small number of error codes. When an error occurs this variable is set to a suitable code. In addition some of the library routines return error codes when appropriate.

7.3.2 Building a network

The first operation which can be performed by the Resource Management library involves building an internal representation of a network. Consider the network in Figure 7.7.

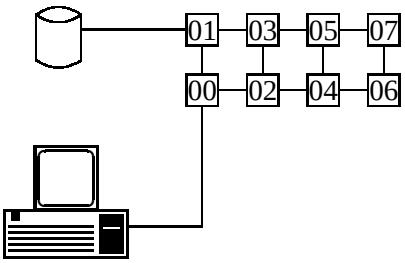


Figure 7.7: Another simple network

Building the representation involves three stages: building the processors, collecting the processors together in a network and making the links between the processors.

Building the processors

The following piece of code is used to build a representation of the root processor, **00**.

```
RmProcessor Proc00 = RmNewProcessor();

RmSetProcessorId(         Proc00, "00");
RmSetProcessorType(       Proc00, RmT_T800);
RmSetProcessorMemory(     Proc00, 4 * 1024 * 1024);
RmSetProcessorPurpose(    Proc00, RmP_System | RmP_Normal);
RmAddProcessorAttribute( Proc00, "scsi");
```

First of all, a new data structure is allocated, and then the library returns an object of the type **RmProcessor**. This might be a pointer to the actual data structure. It could also be an index into a pre-allocated table of these data structures, or some more exotic object. Since **Proc00** will only be used for further calls into the library, its exact data type does not matter. The inverse action is **RmFreeProcessor()**.

Assuming that there was enough free memory to allocate the data structure, and that for simplicity this section ignores the possible error conditions, some of the details for this processor can now be filled in. The processor name, type, and memory size are relatively straightforward. The purpose is the result of using **or** on two pieces of information. The first is either **RmP_System** or **RmP_User**, indicating whether the processor is reserved for use by the system software or whether it can be allocated to users. The second piece of information should be one of the following.

1. **RmP_Normal**. This processor runs some system software to perform basic processor administration, for example message routing and memory allocation.

2. **RmP_Native**. This processor does not run any system software, and hence all the hardware resources are accessible to application software. However, the application software is responsible for administering all the hardware.

3. **RmP_IO**. This processor is reserved for performing I/O operations and is incapable of running any application software. It may still be allocated to users as a means of preventing other users from accessing facilities provided by that processor.

4. **RmP_Router**. This processor is reserved for routing messages. This routing may be performed directly by the hardware, for example a routing chip or an ethernet. Alternatively the network may contain a backbone of cheap processors such as 16 bit Transputers to perform message routing without going through processors running application software.

For example, the processor **/IO** in the network would have the purpose

```
RmP_User | RmP_IO
```

and processor **/01** would have the purpose

```
RmP_User | RmP_Normal
```

In general a processor's purpose is fixed. The only change that the software may allow is from normal to native mode, and back to normal when the application has finished. Any other changes will tend to involve significant changes to the system configuration files, and hence shutting down and rebooting the machine.

The final routine, **RmAddProcessorAttribute()**, is used for other attributes such as attached signal processing chips. In this case the **scsi** attribute indicates that the processor has an SCSI disc interface unit. Such information must be stored with the network because applications may need it: a mapping program might choose to map an I/O intensive task to a processor near the one with the disc interface. Because of the varied hardware available, little can be done about defining the format of these strings. As a general rule, attributes should take either the form *xx*, a simple string, or the form *yy=zz*, specifying both a name and a value. This value may be a number or just another string. Should it be necessary to remove an attribute, there is an inverse routine called **RmRemoveProcessorAttribute()**.

Collecting the processors in a network

To create a network, processors have to be collected together in a data structure. The following code achieves this:

```
RmNetwork  network = RmNewNetwork();

RmSetNetworkId(        network, "example");
RmAddheadProcessor(    network, Proc00);
RmPostinsertProcessor( Proc00,  Proc01);
RmPreinsertProcessor(  Proc01,  ProcIO);
RmAddtailProcessor(    network, Proc02);
RmAddtailProcessor(    network, Proc03);
   ...
```

Effectively the processors are held in a linked list, although the network data structure will contain more information than just a list header. Processors can be added to the head or tail of the list, or inserted before or after processors already in the data structure. Hence the above code would leave the processors in the order shown in Figure 7.8.

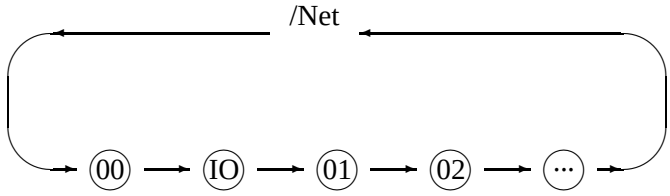


Figure 7.8: Processor ordering within a network

If subnetworks are required, the same routines may be used.

```
RmNetwork      whole_net = RmNewNetwork();
```

```
RmNetwork       subnet    = RmNewNetwork();

RmAddtailProcessor(whole_net, Proc00);
RmAddtailProcessor(whole_net, ProcIO);
RmAddtailProcessor(whole_net, (RmProcessor) subnet);
RmAddtailProcessor(subnet,    Proc01);
RmAddtailProcessor(subnet,    Proc02);
RmAddtailProcessor(subnet,    Proc03);
     ...
```

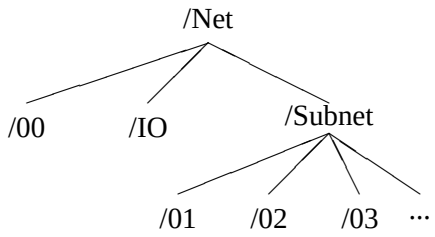


Figure 7.9: A processor tree

This would give a tree as shown in Figure 7.9. The system does not attach any particular significance to the order of processors within a network, but an application might do so. For example, an application might choose to order processors depending on how suitable they are for running a particular application. The routine to remove processors from a network is:

```
RmProcessor RmRemoveProcessor(RmProcessor);
```

The routine returns the RmProcessor value itself, or NULL for failure. This can be useful, for example:

```
RmAddtailProcessor(new_network, RmRemoveProcessor(proc));
```

This line removes a processor from its current network and adds it to a new one. There is another routine, **RmFreeNetwork()**, which releases a network and all the processors contained within it.

Making the connecting links

Just one routine is needed to establish the connections between the processors. This is shown by the following code.

```
RmMakeLink(Proc00, 0, ProcIO, 0);
RmMakeLink(Proc00, 2, Proc01, 0);
RmMakeLink(Proc00, 3, Proc02, 1);
RmMakeLink(Proc01, 3, Proc03, 1);
RmMakeLink(Proc02, 2, Proc03, 0);
RmMakeLink(Proc02, 3, Proc04, 1);
     ...
```

The routine takes four arguments: two processors and two link numbers. Note that it affects two processors simultaneously. Making a link from processor **00** to processor **IO** also makes a link from processor **IO** to processor **00**. To specify external links there is a constant which can be substituted for the second processor.

```
RmMakeLink(Proc00, 1, RmM_ExternalProcessor, 1);
```

The exact meaning of the link number for external processors depends on the system. Typically this link number identifies a specific connector on the machine. For example a Transputer system may have eight link connections coming out of the backplane, allowing other machines to be connected to it through links, with the link number identifying the connector. If a link is not connected, this can be specified explicitly as shown below. However by default the library assumes that all links are not connected so this is not usually necessary.

```
RmMakeLink(Proc00, 1, (RmProcesssor) NULL, 0);
```

Another constant is available if desired.

```
RmMakeLink(Proc00, 1, RmM_NoProcessor, 0);
```

Once a link has been made it can be broken again by using another routine.

```
RmBreakLink(Proc00, 2);
```

Again, breaking a link is symmetrical. The processors at both ends of the link are affected, although only one of the processors has to be specified.

7.3.3 Examining a network

The usefulness of constructing a network is limited. A rather more interesting job is examining an existing network. There are three types of networks worth examining: the whole Network (all processors in the current machine that can be controlled by the library); the user's domain of processors and a network obtained by the application.

Getting information

There are four routines which can be used to get details of the whole Network and the user's domain:

```
RmNetwork  RmGetNetwork();
int        RmLastChangeNetwork(void);
RmNetwork  RmGetDomain();
int        RmLastChangeDomain(void);
```

The first routine obtains a copy of the current Network so that it can be examined by an application. In a multi-user, multi-processor, multi-tasking system the Network may change at any time, but **RmGetNetwork()** gives a current snapshot of the Network. To find out whether or not there has been a recent change to the Network another routine provides a time stamp. The Network is said to have changed under the following conditions:

1. One or more processors have been allocated to a user.

2. One or more processors have been returned to the free pool.

3. Processors have changed from normal to native mode, or vice versa, affecting message routing.

4. The Network topology has changed because an application has reconfigured the connections between processors it has obtained.

5. A fault has been detected, for example a processor crash.

Any applications that rely on up to date information about the network can check the time stamp at regular intervals. Similar routines are available to obtain a copy of the current domain and an appropriate time stamp.

Walking through the linked list of processors

Given that the processors in a network are held in a linked list, some routines must be needed to walk through this list and generally to find out what is in the list.

1. `RmProcessor RmFirstProcessor(RmNetwork);`

   This routine returns the first processor or subnetwork in the specified network, in other words the head of the list.

2. `RmProcessor RmLastProcessor(RmNetwork);`

   This routine returns the last processor or subnetwork.

3. ```
   RmProcessor RmNextProcessor(RmProcessor);
   RmProcessor RmPreviousProcessor(RmProcessor);
   ```

   Given the head or the tail of the list, these two routines can be used to move forwards or backwards through the linked list. When the end is reached these routines return **NULL**.

4. ```
   bool RmIsProcessor(RmProcessor);
   bool RmIsNetwork(RmProcessor);
   ```

   These two routines can be used to check whether or not an entry in the linked list is a processor or a subnet.

5. `bool RmIsNetworkEmpty(RmNetwork);`

   This routine returns true if the specified network does not contain any processors or subnetworks.

6. `int RmSizeofNetwork(RmNetwork);`

   This routine returns the number of processors and subnetworks in the specified network, without recursing down into the subnetworks.

7. `int RmCountProcessors(RmNetwork);`

   This routine returns the total number of processors in the specified network including all its subnets. If the routine returns the same value as **RmSizeofNetwork()**, the network being examined does not contain any subnetworks.

8. `RmNetwork RmParentNetwork(RmProcessor);`

   Given a processor or subnetwork this routine returns the parent network. If the argument passed is the root of the network tree, the routine returns **NULL**.

9. `RmNetwork RmRootNetwork(RmProcessor);`

   Given any processor or subnetwork this routine returns the root of the network tree.

List walking routines provided by the library

Walking through the list of processors in a network is a very common operation, and the library provides a number of routines to make it easier. The application programmer using the library can write routines of the form:

```
int fn(RmProcessor processor, ...);
```

This function can be passed as an argument to the library's list walking routines, which apply the function to the contents of a network passed as argument. The routine is specified as having a variable number of arguments, which is somewhat specific to the C language. To facilitate defining other language interfaces, the library does not guarantee to pass more than three integer-sized arguments.

1. `int RmApplyNetwork(RmNetwork, fn, ...);`

   This routine applies the specified function to all the processors and subnetworks in the specified network, without recursing into the subnetworks. At most three integer-sized arguments should be passed after the function. The routine returns the sum of the results of applying the function to all entries.

2. `int RmApplyProcessors(RmNetwork, fn, ...);`

   This routine is similar to the above, differing only in the way subnetworks are handled. The first routine applies the function to any subnetworks, without recursing down to processors inside the subnetworks. Hence the user-supplied function should check whether it has been passed a processor or a subnetwork as an argument, before it attempts to process the argument. The second routine does not apply the user-defined function to subnetworks, but implicitly recurses down into the subnetworks.

3. `int RmSearchNetwork(RmNetwork, fn, ...);`

   The **apply** routines always act on every object in the network passed as an argument. An alternative requirement is searching the network, applying the function to successive entries until a suitable one is found. This routine applies the function to successive entries until the function returns a non-zero result. The routine then returns this non-zero result. Typically this result would be the **RmProcessor** value itself.

4. `int RmSearchProcessors(RmNetwork, fn, ...);`

   This routine is similar to the previous one, but recurses into subnetworks rather than applying the function to the subnetworks.

Some examples may prove useful at this point. This code counts the number of T800 processors in a network.

```
int walk_fn1(RmProcessor processor, ...)
{ if (RmGetProcessorType(processor) == RmT_T800)
   return(1);
  else
   return(0);
}

int main(void)
{ RmNetwork Network = RmGetNetwork();
  printf("Number of T800 in the current network is %d.\n",
        RmApplyProcessors(network, &walk_fn1));
}
```

RmApplyProcessors() is used so that all processors inside subnetworks are counted as well. Similar code using **RmApplyNetwork()** would look like this:

```
int walk_fn2(RmProcessor processor, ...)
{ if (RmIsNetwork(processor))
   return(RmApplyNetwork((RmNetwork) processor, &walk_fn2));
  if (RmGetProcessorType(processor) == RmT_T800)
   return(1);
  else
   return(0);
}

int main(void)
{ RmNetwork Network = RmGetNetwork();

  printf("Number of T800 in the current network is %d.\n",
    RmApplyNetwork(Network, &walk_fn2));
}
```

A third example finds the first processor of a specified type.

```
int search_fn1(RmProcessor processor, ...)
{ va_list     args;
  int         processor_type;

  va_start(args, processor);
  processor_type = va_arg(args, int);
  va_end(args);

  if (RmGetProcessorType(processor) == processor_type)
   return((int) processor);
  else
   return(0);
}

int main(void)
```

```
{ RmNetwork   Network = RmGetNetwork();
  RmProcessor result;

  result = (RmProcessor)
       RmSearchProcessors(Network, &search_fn1, RmT_T9000);
  if (result == (RmProcessor) NULL)
   puts("There are no T9000 in this network.");
  else
   printf("The first T9000 in the network is %s\n",
     RmGetProcessorId(result));
}
```

Examining a processor

The routines described so far can be used to get hold of the processors in a network. Another set is required to examine the actual processors, and most of these are straightforward.

```
1. const    char *RmGetProcessorId(RmProcessor)
2.          int   RmGetProcessorType(RmProcessor);
3. unsigned long  RmGetProcessorMemory(RmProcessor);
4.          int   RmGetProcessorPurpose(RmProcessor);
5.          int   RmGetProcessorState(RmProcessor);
6.          int   RmGetProcessorOwner(RmProcessor);
7. const    char *RmWhoIs(int);
8.          int   RmWhoAmI(void);
9.          int   RmGetProcessorControl(RmProcessor);
10.         bool  RmTestProcessorAttribute(RmProcessor, char *);
11.         int   RmCountProcessorAttributes(RmProcessor);
12.         int   RmListProcessorAttributes(RmProcessor, char **);
13. const   char *RmGetProcessorAttribute(RmProcessor, char *);
```

The processor purpose again consists of two fields. There is one field to control whether the processor is reserved for use by the system, or whether the processor is available to users. The other field specifies the processor's role in the network. A constant is available to isolate this second field.

```
int purpose = RmGetProcessorPurpose(proc);

printf("Processor %s : ", RmGetProcessorId(proc));
if (purpose & RmP_System)
 fputs("system, ", stdout);
else
 fputs("user, ", stdout);

switch(purpose & RmP_Mask)
 { case RmP_Normal : puts("normal");  break;
   case RmP_Native : puts("native");  break;
```

```
    case RmP_Router : puts("router");  break;
    case RmP_IO     : puts("I/O");     break;
    default         : puts("unknown"); break;
  }
```

The processor state consists of a number of flags including the following:

1. **RmS_Reset**. The processor is believed to be reset.

2. **RmS_Running**. The processor is currently running the default software, providing message routing, memory management and so on.

3. **RmS_Suspicious**. There may be a problem with this processor, for example it has failed to receive a message down a link.

4. **RmS_Crashed**. The processor appears to have crashed with all communication failing.

5. **RmS_Dead**. The processor has crashed and the system was unable to recover it. This might happen because the hardware does not provide sufficient control over the processor, particularly an individual reset, or because of a hardware failure.

The routine **RmGetProcessorOwner()** returns the current owner of the processor, if any. The number returned is likely to be the same as the Unix user identifier, but this is not essential. There are two special identifiers: **RmO_System** specifies that the processor is reserved for use by the system software; **RmO_FreePool** specifies that the processor is currently free, and has not been allocated to any user's domain. The routine **RmWhoAmI()** returns a user identifier for the current session. The routine **RmWhoIs()** takes a user identifier and translates it into a string. The routine **RmGetProcessorControl()** returns a word indicating the amount of control which the system software has over this processor. Again, this word consists of a number of flags:

1. **RmC_Native**. The system may be able to switch this processor into native mode, removing any system software currenly running on it. This is not always possible, for example a network might consist of a number of existing Unix workstations attached to an ethernet, and it is not usually possible to stop Unix running on such processors.

2. **RmC_Reset**. The system can definitely reset the target processor.

3. **RmC_PossibleReset**. The system may be able to reset the target processor, but it cannot guarantee it. Unfortunately this is possible with some hardware.

4. **RmC_FixedMapping**. The mapping of the logical processors administered by the Resource Management library onto the physical processors inside the box is fixed. This is important in systems which support link switching.

5. **RmC_FixedLinks**. The processor links are fixed and cannot be reconfigured. If this bit is clear, some of the links can be reconfigured, but not necessarily all links.

The final four routines examine the additional processor attributes that may be defined, to indicate facilities like signal processing chips or disc interface units. The first, **RmTestProcessorAttribute()**, checks if the string provided is one of the known attributes. The second routine, **RmCountProcessorAttributes()**, returns the number of attributes for this processor, which may be zero. To find all the attributes, a suitable array should be allocated and this is filled in by the routine **RmListProcessorAttributes()**. The final routine searches for strings of the form *aa=bb*. If the application gives the string *aa*, the routine returns the string *bb*. The use of these routines is illustrated by the following code fragment.

```
if (RmTestProcessorAttribute(proc, "scsi"))
 { char *drive_count = RmGetProcessorAttribute(proc, "drive_count");
   int  no_drives;
   if (drive_count == NULL)
    no_drives = 1;
   else
    no_drives = atoi(drive_count);

   printf("Processor %s has a SCSI unit with %d drives\n",
        RmGetProcessorId(proc), no_drives);
 }
int number_attribs = RmCountProcessorAttributes(proc);

if (number_attribs > 0)
 { char **attrib_table = malloc(number_attribs * sizeof(char *));
   int    i;
   char   *id = RmGetProcessorId(proc);

   RmListProcessorAttributes(proc, attrib_table);
   for (i = 0; i < number_attribs; i++)
    printf("Processor %s has attribute %s\n", id,
           attrib_table[i]);

   free(attrib_table);
 }
```

Examining links

Given a network of processors it is useful to be able to work out which processor is connected by which links to which other processors. There are two routines for this purpose. The first is:

```
int    RmCountLinks(RmProcessor);
```

This routine simply returns the number of links attached to this processor. The second routine is better:

```
RmProcessor RmFollowLink(RmProcessor, int link, int *destlink);
```

Given a processor and a link number on that processor, the library follows the link to the other side and returns the connecting processor. The third argument should be a pointer to an integer variable, which is filled in with the destination link number. If the specified link is not connected, the routine returns the value NULL. The library

defines a constant **RmM_NoProcessor**, equivalent to NULL. If the specified link is connected to something outside the network containing the processor, the routine returns **RmM_ExternalProcessor**. This can be quite common. For example, when examining a user's domain any connection to a processor inside the Network but not allocated to the domain will be external to the domain. The following code fragment illustrates the use of this routine.

```
int         number_links = RmCountLinks(proc);
int         i;
RmProcessor neighbour;
int         destlink;

for (i = 0; i < number_links; i++)
 { neighbour = RmFollowLink(proc, i, &destlink);
   if (neighbour == RmM_NoProcessor)
    { printf("link %d is not connected.\n", i);
      continue;
    }
   if (neighbour == RmM_ExternalProcessor)
    { printf("link %d goes outside this network.\n", i);
      continue;
    }
   printf("link %d is connected to link %d of processor %s\n",
          i, destlink, RmGetProcessorId(neighbour));
 }
```

7.3.4 Obtaining a network

If a network is to be shared by multiple users and multiple applications per user, there must be some mechanism to prevent tasks from being run without authorisation, for example on another user's processor. This is fairly easy. All processors to be used by an application must be obtained first.

A user's domain is the set of processors obtained by any applications running in the user's session and not yet returned to the free pool. A given processor can only be in one domain, in other words allocated to only one user. A processor inside a domain can be allocated to more than one application, subject to some restrictions. An application can run more than one task in a processor it has obtained, again subject to some restrictions.

Actually obtaining the processors

There are two main ways of obtaining access to processors. The first method is to construct a template describing the application's requirements, for example sixteen T800 processors with at least four megabytes of memory. The system software will perform some pattern matching between the template and the available processors, and if a suitable match is found, those processors are obtained. The second method is to examine an existing network, either the whole Network or the current user's domain, choose some specific processors, and attempt to obtain them. The first of these approaches tends to involve less code because constructing a template network

is quite easy. The second method involves more code, but gives greater control. The following routines are used to obtain and release processors.

1. `RmProcessor RmObtainProcessor(RmProcessor);`

   Given an **RmProcessor** value, which may correspond to either an existing processor or a newly created template, this routine returns a new **RmProcessor** value or **NULL**. The returned processor is now owned by the application, and can be used for running applications.

2. `int RmReleaseProcessor(RmProcessor);`

   If a processor is no longer needed and is not running any user tasks, this routine releases the application's ownership of the processor. This may involve returning the processor to the free pool, depending on whether or not other applications are using it. Normally any resources obtained by an application will be released automatically when it exits, but this routine gives greater control.

3. `RmNetwork RmObtainNetwork(RmNetwork, bool exact);`

   Given a network template describing the application's requirements, this routine attempts to get access to a suitable matching set of processors. Alternatively the template might consist of a copy of an existing network, for example that obtained by **RmGetDomain()**, with the unwanted processors removed. The second argument specifies whether the routine should succeed only if an exact match is found for all the processors in the template, or whether a partial match will suffice.

4. `RmNetwork RmObtainProcessors(int, RmProcessor *, bool);`

   When attempting to obtain some existing processors it is usually convenient to search through the existing network and fill in a table, rather than build a new network to serve as a template for **RmObtainNetwork()**. This routine takes a table size, a pointer to a table of **RmProcessor** values, and a boolean to indicate whether or not partial success will suffice.

5. `int RmReleaseNetwork(RmNetwork);`

   Again, any network of processors obtained by an application will be released automatically when that application exits, but this routine enables applications to change their resources easily at any time.

6. `bool RmIsFree(RmProcessor, bool exclusive);`

   When examining an existing network of processors it is necessary to know whether or not a processor can be obtained. One way is to check whether it is in the free pool or in the user's current domain, which can be done with routines already described. However the examination routines act on a local copy of the network or a subset of the network, which may be out of date. Hence this routine performs a spot-check on the specified processor to determine whether or not it can be obtained at present. Of course the routine cannot guarantee that the processor can be obtained, because some other application may obtain the processor a fraction of a second after the spot check. The second argument depends on whether the application needs exclusive access to the processor or whether it is willing to share the processor with other applications.

Administering the obtained processors

Another four routines are useful to administer the controlled processors, but they merely involve administering the local data structures.

1. `RmProcessor RmFindMatchingProcessor(RmProcessor, RmNetwork);`

   When a network is obtained by specifying a template of, for example, four T800 and four T9000, it is necessary to find out which processor in the template was matched with which processor in the obtained network. This routine can be used to follow the mapping. The processor argument should correspond to part of the template which is used in the call to **RmObtainNetwork()** or **RmObtainProcessors()**, and the network argument should be the result of that routine. The return value is NULL if no match was found for that processor, or the matching processor.

2. `int RmMergeNetworks(RmNetwork neta, RmNetwork netb);`

   Suppose that an application obtains two or more different networks of processors. For example, halfway through a run it may decide that it needs another sixteen processors. Rather than keeping all the different sets of obtained processors in separate **RmNetwork** structures it is usually desirable to maintain only one, and merge newly obtained networks into the current set. This routine takes all the processors in set **netb**, merges them into set **neta**, and leaves **netb** devoid of processors.

3. `RmNetwork RmGetNetworkHierarchy(void);`

   If the application obtains its processors using **RmObtainProcessor()**, that is to say one at a time rather than getting sets of processors at a time, usually these processors should still be collected together into a network. This can be a newly created **RmNetwork**. However, using a new structure might cause confusion because the resulting network of obtained processors does not match the subnet hierarchy of the global Network. Hence it may be preferable to obtain a copy of the Network hierarchy: just the subnetworks and not the processors, and have the library insert any obtained processors into the right position in the library.

4. `int RmInsertProcessor(RmNetwork, RmProcessor);`

   Given a newly obtained processor and a network matching the Network hierarchy, this routine inserts the processor into the correct position in the hierarchy.

Sharing processors

Applications may need finer control over processor allocation than is provided by the routines described so far. For example, an application may require exclusive access to the processors they have obtained while others are willing to share processors with other applications (although not with other users). The following routines control this.

```
int   RmSetProcessorShareable(RmProcessor);
int   RmSetProcessorExclusive(RmProcessor);
int   RmIsProcessorShareable( RmProcessor);
int   RmIsProcessorExclusive( RmProcessor);
```

The first two routines may be be applied to the template processors used for the actual **obtain** calls, and provide information for the system software to help it perform the required matching. For example, if an application specifies that one of the processors in the template of the requirements should be exclusive, the system will not match this with any processors already allocated to other applications. Furthermore, once the processor has been obtained the system will not allocate it to other applications. Alternatively, the first two routines may be applied to processors already obtained.

For example, an application may need exclusive access to a processor during the first part of a run, but is then willing to share the processor with other applications. Changing a processor from shared to exclusive may fail if another application was also given access to the processor while it was still shared. The final two routines can be used to test the current allocation strategy for a processor. To avoid total confusion, a given processor cannot be allocated twice to the same application.

Administering the domain

The administration of a user's domain needs some additional routines. In particular, an application may need to obtain some processors from the free pool, implicitly adding them to the user's domain, then to mark them as **sticky** and release the processors. Normally, releasing processors would cause them to leave the domain and return to the free pool, but in this case they would remain in the user's domain until a subsequent application marks them as no longer sticky. There are four routines to control this. These routines are unlikely to be used by user-written applications, but will be used by utilities provided with the system.

```
int   RmSetProcessorPermanent(RmProcessor);
int   RmSetProcessorTemporary(RmProcessor);
bool  RmIsProcessorPermanent( RmProcessor);
bool  RmIsProcessorTemporary( RmProcessor);
```

7.3.5 Constructing a task force

In the Resource Management library, networks consist of processors connected by point to point links. Similarly, task forces consist of tasks connected by point-to-point channels. Hence, not surprisingly, the routines to construct a task force are very similar to the routines to construct a processor network. Obviously there are significant differences between processors and the programs that run on them, so the routines are not identical. As an example, consider the task force shown in Figure 7.10. The task force is a simple farm. There is a master component or task `M`, a load balancing component `lb`, and there are a number of worker components `W`.

Building the tasks

The following code fragment can be used to build the master component.

```
RmTask master = RmNewTask();

RmSetTaskId(     master, "master");
RmSetTaskType(   master, RmT_T414);
RmSetTaskNormal( master);
```

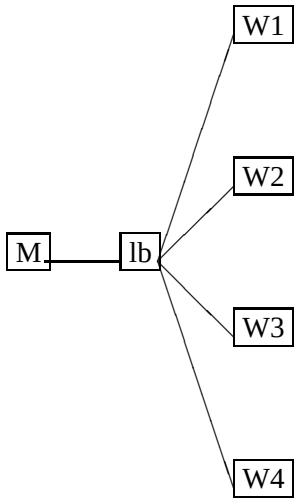


Figure 7.10: A simple task force

```
RmSetTaskMemory( master, 2 * 1024 * 1024);
RmSetTaskCode(   master, "../bin/master");
```

The routines available for creating tasks are:

1. `RmTask RmNewTask(void);`

   This routine returns a new **RmTask** value. Again this may be a pointer, an index into a table, or something else. The task data structure can only be manipulated indirectly, through the library. There is an inverse routine, **RmFreeTask()**.

2. `int RmSetTaskId(RmTask, char *);`

   This routine fills in the task identifier. Usually, but not necessarily, this will be related to the name of the file holding the program.

3. `int RmSetTaskMemory(RmTask, unsigned long);`

   This specifies the memory requirements of the program. It is important when the application lets the system map the task, rather than performing its own mapping.

4. `int RmSetTaskCode(RmTask, char *filename);`

   A task is associated with a piece of binary code, which usually resides on a disc. This routine associates the task with a particular file name.

5. `int RmSetTaskType(RmTask, int);`

   This routine specifies the processor type needed by the program. It is used when the application lets the system map a task force onto the available network. For

flexibility some additional routines are provided to find out the processor type, given a file name.

6. `int RmGetProgramType(char *);`

   Given a file name this routine returns an integer describing the type of processor that can be used to run this program.

7. `int RmSetTaskNative(RmTask);`

   This routine is used to specify that the task should be run on a native processor only, in the absence of any system software. The task will need full access to all the hardware resources, and cannot share the processor with any other tasks.

8. `int RmSetTaskNormal(RmTask);`

   This routine specifies that the task should run on a processor with the normal system software, providing message routing and other facilities. Also, such a task can usually share the processor with other tasks.

9. `int RmAddTaskAttribute(RmTask, char *);`

   This routine is used to associate an arbitrary string with the task. If the system is responsible for mapping the task onto the available network, the string is assumed to define some special hardware facility that must be available on the target processor. There is an inverse routine, **RmRemoveTaskAttribute()**.

10. `int RmAddTaskArgument(RmTask, int, char *);`

    This is described later, in the subsection on program environments.

Collecting the tasks in a task force

Collecting tasks together into a task force is almost identical to collecting processors together in a network.

Networks	Task forces
`RmNewNetwork()`	`RmNewTaskforce()`
`RmFreeNetwork()`	`RmFreeTaskforce()`
`RmSetNetworkId()`	`RmSetTaskforceId()`
`RmAddtailProcessor()`	`RmAddtailTask()`
`RmAddheadProcessor()`	`RmAddheadTask()`
`RmPreinsertProcessor()`	`RmPreinsertTask()`
`RmPostinsertProcessor()`	`RmPostinsertTask()`
`RmRemoveProcessor()`	`RmRemoveTask()`

For example, the following code fragment collects together the various tasks of the farm shown earlier.

```
RmTaskforce     farm = RmNewTaskforce();

RmSetTaskforceId(farm, "mandelbrot");
RmAddtailTask(farm,    master);
```

```
RmAddtailTask(farm,    load_balancer);
for (i = 0; i < number_workers; i++)
 RmAddtailTask(farm, worker[i]);
```

Making the connecting channels

Within a network the processors are connected by point-to-point links. Similarly, within a task force the tasks are connected by point-to-point channels. Links within a processor are identified by numbers starting from 0. Similarly channels between tasks are identified by numbers starting from 0. These channels are in fact identical to the conventional Unix file descriptors, so channel 0 corresponds to a program's standard input stream **stdin**. More details of this can be found in chapter 4 on the component distribution language, *CDL*. The routine for creating links between processors is **RmMakeLink()**. Similarly the routine for making channels between tasks is **RmMakeChannel()**. The following code fragment establishes all the channels for the farm example.

```
RmMakeChannel(master, 5, load_balancer, 0);
RmMakeChannel(load_balancer, 1, master, 4);

for (i = 0; i < number_workers; i++)
 { RmMakeChannel(load_balancer, 5 + (2 * i), worker[i], 0);
   RmMakeChannel(worker[i], 1, load_balancer, 4 + (2 * i));
 }
```

In this code fragment the conventional direction of Unix file descriptors is preserved. For example the first line connects an extra output file descriptor within the master component to the standard input of the load balancer, and the second line connects the standard output of the load balancer to an extra input file descriptor in the master component. In practice channels are bi-directional, so the following two lines are totally equivalent:

```
RmMakeChannel(master, 4, load_balancer, 1);
RmMakeChannel(load_balancer, 1, master, 4);
```

External channels correspond to file descriptors not used for communicating with other tasks. For example, by default, channel 3 of every task is an external channel corresponding to the standard error stream **stderr**. It is possible to redirect certain channels to particular files or named devices. For example, the following command line:

```
ls -l > listing
```

can be implemented using the following code fragment:

```
RmTask ls = RmNewTask();

RmSetTaskId(            ls, "ls");
RmSetTaskCode(          ls, "/helios/bin/ls");
RmSetTaskArgument(      ls, 1, "-l");
RmConnectChannelToFile( ls, 1, "listing", O_WRONLY);
```

The final argument should be a conventional Unix open mode.
The routine to break a channel is:

```
RmBreakChannel(RmTask, int);
```

Again this routine is symmetrical and affects both ends of the channel.

7.3.6 Examining a task force

Again, the routines available for examining a task force are very similar to the routines for examining a network. The following table indicates one to one matches.

Network	Taskforce
RmGetNetworkId()	RmGetTaskforceId
RmFirstProcessor()	RmFirstTask()
RmLastProcessor()	RmLastTask()
RmNextProcessor()	RmNextTask()
RmPreviousProcessor()	RmPreviousTask()
RmIsNetworkEmpty()	RmIsTaskforceEmpty()
RmSizeofNetwork()	RmSizeofTaskforce()
RmCountProcessors()	RmCountTasks()
RmParentNetwork()	RmParentTaskforce()
RmRootNetwork()	RmRootTaskforce()
RmApplyNetwork()	RmApplyTaskforce()
RmSearchNetwork()	RmSearchTaskforce()
RmApplyProcessors()	RmApplyTasks()
RmSearchProcessors()	RmSearchTasks()
RmCountLinks()	RmCountChannels()
RmFollowLink()	RmFollowChannel()
RmGetProcessorMemory()	RmGetTaskMemory()
RmGetProcessorId()	RmGetTaskId()
RmGetProcessorType()	RmGetTaskType()
RmTestProcessorAttribute()	RmTestTaskAttribute()
RmCountProcessorAttributes()	RmCountTaskAttributes()
RmListProcessorAttributes()	RmListTaskAttributes()
RmGetProcessorAttribute()	RmGetTaskAttribute()
RmIsProcessor()	RmIsTask()
RmIsNetwork()	RmIsTaskforce()

In addition there are a number of routines specific to tasks and task forces.

1.
```
bool   RmIsTaskNative(RmTask);
bool   RmIsTaskNormal(RmTask);
```

   These routines check whether a specific task should run on a native processor or not.

2.
```
const char *RmGetTaskCode(RmTask);
```

   This routine returns the name of the file containing the code for this task.

3.
```
const char *RmGetTaskArgument(RmTask, int);
int         RmCountTaskArguments(RmTask);
```

   These are described below in section 7.3.7 on a program's environment.

4. `const char *RmFollowChannelToFile(RmTask, int, int *);`

   If a call to **RmFollowChannel()** returns the constant **RmM_ExternalChannel**, this channel is in fact connected to a file or a named device. The name can be obtained by this library routine. The third argument should be a pointer to a suitable integer value, which will be set to the Unix **open** mode.

7.3.7 A program's environment

A native task executes without any operating system support. It cannot perform file I/O, display graphics, read data from a keyboard, and so on except by communicating through its processor's links to reach a server program which can perform those operations. Hence concepts such as a current directory are irrelevant to native tasks, which are said to operate without an environment. However, conventional programs do have an environment consisting of the following pieces of information:

1. One or more argument strings `argv[]`. For example, when the **ls** command is executed with the command line

   ```
   ls -l doc
   ```

   the **ls** program is given three arguments. The first argument is **ls**, the id of the program. The second and third arguments are **-l** and **doc**.

2. Some environment strings. A typical environment string might be

   ```
   TERM=ansi
   ```

3. Standard streams. Usually these refer to the current terminal, but standard streams may be redirected.

4. A **context** of miscellaneous items of information. This includes a current directory, protection information such as user id and group id, a controlling terminal, signal processing information and so on.

Consider the Unix **execve()** family of routines, which are used to start other programs. When one of the Unix routines is used the newly started program inherits its context and standard streams from its parent. The environment strings are usually inherited. The arguments to the new program are supplied by the library routine. The routines provided by the Resource Management library to execute tasks and task forces are similar. The context, standard streams, and environment strings are always inherited. The arguments are passed on. However, a task force with a large number of component tasks is rather different from a simple program such as **ls**, so the actual rules are somewhat more complicated. Consider the following example. The user types the following command:

```
mapfarm 10 100000 200000 > log
```

The **mapfarm** program is a user application linked with the Resource Management library. It examines the current network and obtains some suitable processors. Then it constructs a task force consisting of a master, load balancer, and ten workers like the first argument. This task force is mapped onto the obtained processors, attempting

to produce an efficient mapping. This mapped task force is then executed with two arguments, 100000 and 200000. The Unix file descriptors for the **mapfarm** program are as follows:

File descriptor	C equivalent	where to/from
0	**stdin**	terminal
1	**stdout**	file **log**
2	**stderr**	terminal

The **master** component of the farm has channels 4 and 5 connected to the load balancer. The remaining channels are inherited from the parent, which in this case is the **mapfarm** program.

File descriptor	C equivalent	where to/from
0	**stdin**	terminal
1	**stdout**	file **log**
2	**stderr**	terminal
3		not used
4		from load balancer
5		to load balancer

For the load balancer, standard input and output are connected to the master component. Channels 4 onwards are connected to the various workers.

File descriptor	C equivalent	where to/from
0	**stdin**	from master
1	**stdout**	to master
2	**stderr**	terminal
3		not used
4		from worker 0
5		to worker 0
6		from worker 1
7		to worker 1
. . .		. . .

For the worker components, all workers have the same channel allocation. Channel 0 comes from the load balancer, channel 1 goes to the load balancer, and the rest are inherited from the parent.

File descriptor	C equivalent	where to/from
0	**stdin**	from load balancer
1	**stdout**	to load balancer
2	**stderr**	terminal

Note: the destination of each component and of each channel in the task force is inherited from the parent program, unless a channel has been redirected using **RmMakechannel()** or **RmConnectChannelToFile()**.

The **mapfarm** program is given three arguments: 10 for the number of workers and 100000 and 200000 for the parameters of the job. It is necessary to pass the number of workers on to the load balancer, and the job parameters to the master component. The worker components do not need any arguments. The simple way is to build all arguments into the task force data structure, using the code below. Please note that `argv[]` refers to the arguments passed to the **mapfarm** program.

```
RmSetTaskArgument(load_balancer, 1, argv[1]);
RmSetTaskArgument(master,        1, argv[2]);
RmSetTaskArgument(master,        2, argv[3]);
```

Note that the argument numbers start at one, not zero, because conventionally argument zero is reserved for the program name, or in Resource Management library terms the task id. The arguments passed to the various tasks can be examined by two other routines:

```
const char *RmGetTaskArgument(RmTask, int);
      int   RmCountTaskArguments(RmTask);
```

For example, counting the arguments for the load balancer task would return the value 2: argument 0 is the task id, and argument 1 is a string corresponding to the number of workers. Getting argument 0 is equivalent to getting the task id, and getting any other argument returns the appropriate string.

An alternative way is to use inheritance. Previously all arguments were built into the task force structure. The task force as a whole does not have to be given any arguments. Consider a scenario where a complex task force is constructed and mapped, written to a file, and then has to be executed a number of times with different arguments every time. It is possible to give these arguments to the task force as a whole, and have a way for the component tasks to inherit a suitable subset. To do this special strings can be used. For example,

```
RmSetTaskArgument(load_balancer, 1, "$1");
RmSetTaskArgument(master, 1, "$2");
RmSetTaskArgument(master, 2, "$3");
```

This specifies that the load balancer inherits the first argument passed to the task force as a whole, and the master inherits arguments two and three. Any task force argument can be inherited by any number (including zero) of component tasks.

7.3.8 Executing a task

Once a task has been constructed in memory, the simplest way to execute it is:

```
RmTask RmExecuteTask(NULL, RmTask, char **argv);
```

This routine causes the system to execute the specified task on a suitable processor, possibly the one running the current program, and passes this task the array of arguments. This array is filtered as described in the previous subsection. The routine returns a new copy of the task structure which can be used for the remaining task execution commands. An alternative version of this routine is:

```
RmTask RmExecuteTask(RmProcessor, RmTask, char **argv);
```

This permits the application to run a task on a specific processor that it has obtained. Essentially this gives the functionality of the Helios **remote** command. Other routines relating to task execution are:

1. `int RmWaitforTask(RmTask);`

   This routine is roughly equivalent to the Posix **waitpid()** routines. It waits for the specified task to terminate, normally or abnormally, and returns the task return code.

2. `int RmWaitforTasks(int, RmTask *);`

   Given a table of tasks and a count of the number of tasks this routine waits for one of the tasks to stop running and returns its return code.

3. `bool RmIsTaskRunning(RmTask);`

   When the previous routine returns, this one can be used to find out which one of the various tasks in the table terminated. In practice there are timing problems: two tasks might have terminated at approximately the same time; one of these would have caused the return of **RmWaitforTasks()**, but both tasks would be listed as non-running by the time the application examines the table.

4. `int RmGetTaskReturncode(RmTask);`

   Once a task has finished running, this routine can be used to retrieve its return code.

5. `int RmSendTaskSignal(RmTask, int);`

   While a program is running, the conventional Unix way to control it is by sending it signals. Useful signals include **SIGINT** for user attention, **SIGKILL** to force program termination, and **SIGUSR1** for user-defined operations.

6. `int RmLeaveTask(RmTask);`

   This routine has no real Unix equivalent. It detaches the specified task from the current program, so that the current program can exit without affecting the task. The task will continue running in the background until it terminates normally.

7.3.9 Executing a task force

The routines for executing whole task forces are very similar to those for executing single tasks.

1. `RmTaskforce RmExecuteTaskforce(NULL, RmTaskforce, char **);`

   This routine causes the system to attempt to map the task force onto the existing network, obtaining processors as required, and then execute it with the arguments specified. Note that the system makes no guarantee at all about the mapping quality, so the performance of the task force may not be optimal.

2. `RmTaskforce RmExecuteTaskforce(RmNetwork, RmTaskforce, char **);`

   This routine causes a mapped task force to be executed in a network of obtained processors, with the given arguments. It is not necessary for every component

task to be mapped onto an obtained processor as the system will take care of the remaining ones, but again the system makes no guarantee about the quality of the mapping. The routines for mapping tasks onto processors are listed below.

3. `int RmWaitforTaskforce(RmTaskforce);`

   Given an executing task force this routine waits for the whole task force to terminate. A task force is said to have terminated when all of its component tasks have terminated or have been aborted in some way. The return code for a task force is zero if all component tasks have terminated with a zero return code, otherwise it is the return code of the first task to terminate with a non-zero result. The routine **RmWaitforTasks()** may be used (with suitable casting) for task forces or a mixture of tasks and task forces, as desired.

4. `bool RmIsTaskforceRunning(RmTaskforce);`

   This routine returns **true** if any of the task force components are still running, **false** otherwise. Alternatively the **RmIsTaskRunning()** routine can be applied to any of the component tasks.

5. `int RmGetTaskforceReturncode(RmTaskforce);`

   Once a task force has stopped running, this routine can be used to retrieve the return code.

6. `int RmSendTaskforceSignal(RmTaskforce, int);`

   This routine is used to send the signal to all component tasks in the task force.

7. `int RmLeaveTaskforce(RmTaskforce);`

   This routine can be used to leave the task force running in the background, while the current program exits.

The **execute** routine returns a new **RmTaskforce** structure for the executing task force, which is different from the **RmTaskforce** template used in the call. This can be quite useful, for example an application might construct a single task force template and then execute this several times with different arguments. As the calculations progress, some of the executing task forces might be aborted because they are failing in some way, while new task forces might be started up with some refined data. Another reason for keeping the template task force and the executing task force separate is to enforce some protection: it does not make much sense to send a signal to a task force that is not yet executing; also, changing a task force while it is running is rather more involved than changing a simple template.

Note: the the current version of the library prevents the changing of task forces while they are running.

7.3.10 Mapping a task force

The previous subsection showed how a task force that had been fully or partially mapped onto a network of processors could then be executed. This subsection describes how the mapping can be done. The mapping of tasks onto processors is essentially one to one. A given component task in a task force can be mapped onto only one processor. It must be mapped onto a processor before the task can be executed, either by the user's application or by the system software. The basic routine to do this is:

```
int RmMapTask(RmProcessor, RmTask);
```

Given an existing processor and a task that is not yet executing, this routine associates the task with the processor. Before the task, or the task force which contains it, can be executed the processor must be obtained. To undo the mapping, the following routine can be used:

```
int RmUnmapTask(RmTask);
```

Given a network of processors, you can find out which processor a task is mapped onto.

```
RmProcessor RmFollowTaskMapping(RmNetwork, RmTask);
```

For example, the following technique is possible:

1. Construct the target task force.
2. Examine the global network, and choose a set of processors which provide an efficient mapping of the task force onto the network. This can be very expensive. Remove any processors that are not required.
3. Map the task force onto the network.
4. Save both the network and the task force to a file.

Later, whenever the task force must be executed:

1. Read the network and the task force from the file.
2. Obtain the processors specified in the network.
3. Execute the task force in the obtained network.

This procedure will fail unless all the processors in the optimal network are available when the task force has to be executed. The probability of this is difficult to estimate. For a large system with tens of users logged in the chances of a specific set of processors all being available are small. On the other hand, if such a system can also run batch jobs overnight with only one or two batch jobs sharing the network, the chances are quite good.

7.3.11 Modifying a network

Having the Helios Nucleus present on every processor tends to make the writing of application programs much easier, because Helios takes care of fairly complex jobs such as message routing. It also has disadvantages. In particular, the system software takes over all the Transputer links and adds an overhead to all communication.

Some applications need full access to all the processor hardware, and cannot run efficiently unless they can take over all the processor links and use all of the available bandwidth. Such applications have to run on **native** processors, which do not have any system software at all. Also, some existing languages such as occam are designed to run only on native processors. To cope with these the Resource Management library provides facilities for controlling native processors.

Note: there is a strong argument that as processors become more complicated, for example an i860 with on-board memory management hardware, it becomes much more difficult to run code without any system software. Hence the use of native networks will decline. However, with the current generation of Transputers, and possibly also the T9000, native processors are important.

At any one time a network may contain several groups of processors running in native mode, with the remainder running Helios normally. Care must be taken when switching processors to native mode because there is a chance that parts of the network will become disconnected from each other. Essentially the facilities provided are as follows:

1. Obtain a network of processors suitable for running in native mode. If a template is used for one of the **obtain** routines, rather than details of existing processors, some or all of the processors may have had their purpose set to **RmP_Native**. This causes the system to attempt to find a set of processors suitable for switching into native mode. The system takes into account both the hardware facilities available for controlling the processors and the current connectivity of the network.

2. Switch one or more processors into native mode. This involves a number of steps:

   (a) Check that this would not cause the network to become disconnected. The system cannot assume that the user's controlling program is well behaved.

   (b) Disable any links connecting the native processors to the rest of the network. The system will automatically start re-routing messages. The system will not respond to any data coming from native processors, so as far as the native network is concerned it is completely isolated.

   (c) Cause the Nucleus to terminate on the native processors.

3. Reset one or more processors, to prepare them for running native code. Processors can be reset any number of times, in order to run different standalone programs.

4. Reconfigure the links between the native processors, subject to hardware restrictions, in order to make the native network topology match the application topology.

5. Reboot native processors when the program has finished with them.

6. Automatically clean up native processors if the program exits without rebooting and releasing the native processors.

The routines to switch some processors from normal to native mode are:

```
int RmSetNetworkNative(RmNetwork);
int RmSetProcessorsNative(int count, RmProcessor *);
```

These routines only affect those processors which have their purpose set to **RmP_Native**. It is quite common to have one processor running under the system software, acting as a server program of some sort, and booting the native code into its network. Similar routines are used to reset processors:

```
int RmResetNetwork(RmNetwork);
int RmResetProcessors(int count, RmProcessor *);
```

Once again, these routines will only affect those processors which have their purpose set to **RmP_Native**. The first version is useful when the native code is first booted into the native network. The second is useful when some of the native processors have to be reset in order to run a different native code. The rebooting of processors involves much the same code.

```
int RmRebootNetwork(RmNetwork);
int RmRebootProcessors(int, RmProcessor *);
```

It is somewhat more complicated to reconfigure links. Once a program has obtained a network and switched it to native mode, it can attempt to reconfigure the links. To do this it must first modify its copy of the network, using routines **RmBreakLink()** and **RmMakeLink()**, to establish the new topology. Then it calls one of the following routines:

```
int RmReconfigureNetwork(RmNetwork, bool, bool);
int RmReconfigureProcessors(int, RmProcessor, bool, bool);
```

Two additional boolean arguments are provided. The first describes how strictly the system should interpret the connections in the network:

1. **FALSE**. Suppose processor `a`, link `x`, is supposed to be connected to processor `b`, link `y`. If the first boolean argument is false, the routine succeeds if a link of processor `a` is connected to a link of processor `b`. This establishes the basic connections between the processor, but not the exact connections.

2. **TRUE**. Suppose processor `a`, link `x,`, is supposed to go to processor `b`, link `y`. The routine will succeed only if exactly that connection can be made.

Some hardware provides full switching, whereby any link of any processor can be connected to any link of any other processor. However there is a lot of hardware which supports only limited switching. For example links 0 and 2 of all processors can be connected to each other in any way, links 1 and 3 can be connected in any way, but it is not possible to connect a link 0 to a link 1 or link 3.

The second boolean argument controls the logical to physical mapping of processors. If two physical processors are not currently executing any code, or are executing identical code, and all their links are about to be reconfigured, it is possible to switch a physical processor from one logical processor to another. This can be extended to any number of processors. If the second argument is TRUE, the logical to physical mapping of processors must be preserved, for example because code is still running on some of the processors. If the second argument is FALSE, the system software is allowed to change the logical to physical mapping if this is necessary.

Irrespective of whether the reconfiguration calls fail completely, succeed by rearranging some of the specified links if permitted, or succeed completely, the various connections will be updated by the routine to reflect the new state of the machine. To undo a link reconfiguration, the library provides two routines.

```
int RmRevertNetwork(RmNetwork);
int RmRevertProcessors(int, RmProcessor *);
```

These routines attempt to revert the processors' connections to their default setup. This is not always possible. For example, in the original setup link `x` of processor `a` might have been connected to link `y` of processor `b`. Processor `a` has been allocated to one application, and its links were reconfigured such that the connection to processor `b` was broken. Processor `b` has been allocated to a different application and its links have also been reconfigured. Even if the connections for processor `a` are supposed to revert to the default state, the connection to processor `b` cannot be made again until that processor's links should also revert.

Given the possible limitations in link switching, it can be desirable to test the possibility of making one or more links before actually changing the configuration. This is possible without having obtained the processors in connection, for example if the program needs to allocate a group of processors which it knows it can reconfigure as desired. The routines for this are:

```
bool RmIsLinkPossible(RmProcessor, int, RmProcessor, int);
bool RmIsNetworkPossible(RmNetwork, bool, bool);
bool RmAreProcessorsPossible(int, RmProcessor *, bool, bool);
```

The first routine tests a single link, and the other two test a whole network. The two boolean arguments are the same as for **RmReconfigureNetwork()**.

7.3.12 File I/O

To avoid unnecessary recalculation it is desirable to be able to save data to a file and retrieve it later. For example, a program might be used once to map a complex task force onto a network and write the resulting mapping to a file. Another program can then use this file repeatedly to obtain the processors and execute the task force. There is a problem with this: since the actual data structures used by the Resource Management library are not known to the application programmer, it is the library that has to perform the file I/O. Hence there are a number of routines to do this.

1. `int RmWrite(char *filename, RmNetwork, RmTaskforce);`

   This routine opens the specified file and writes the network and the task force to this file. Either of the network or the task force, but not both, can be NULL. This routine is used by the Helios resource map compiler **rmgen** to produce the binary resource map. The file is closed at the end of the operation.

2. `int RmRead(char *filename, RmNetwork *, RmTaskforce *);`

   This routine opens the specified file for reading and extracts a network and a task force. If the **RmNetwork** pointer argument is NULL, any network in the file is skipped, otherwise the pointer is filled in. If a non-NULL pointer is used but the file does not contain a network, the pointer will be filled in with NULL. The same applies to the task force.

3. ```
   int RmWritefd(int fd, RmNetwork, RmTaskforce);
   int RmReadfd( int fd, RmNetwork *, RmTaskforce);
   ```

   These two routines are similar to the previous ones but instead of a filename argument they take a Unix file descriptor as argument. Hence the network and
   ```

task force can be made part of a file rather than a whole file. Alternatively the network and task force can be shipped between programs through pipes or sockets subject to protection restrictions: if an obtained processor is shipped from one program to a second, this second program does not have access to the processor.

4. ```
   int RmWritefdNetwork(  int fd, RmNetwork);
   int RmReadfdNetwork(   int fd, RmNetwork *);
   int RmWritefdTaskforce(int fd, RmTaskforce);
   int RmReadfdTaskforce( int fd, RmTaskforce *);
   ```

   These routines are similar but apply to networks only or to task forces only, not to both networks and task forces.

5. ```
   int RmWritefdProcessor(int fd, RmProcessor);
   int RmReadfdProcessor( int fd, RmProcessor *);
   int RmWritefdTask(     int fd, RmTask);
   int RmReadfdTask(      int fd, RmTask *);
   ```

   Instead of affecting a whole network with all its processors, or a whole task force with all its component tasks, these four routines apply to single processors or single tasks.

6. ```
   int RmWritefdNetworkOnly(   int fd, RmNetwork);
   int RmReadfdNetworkOnly(    int fd, RmNetwork *);
   int RmWritefdTaskforceOnly( int fd, RmTaskforce);
   int RmReadfdTaskforceOnly(  int fd, RmTaskforce *);
   ```

   These routines are available to perform I/O on the network or task force structures only, without automatically performing I/O on the processors contained by the network or the tasks contained by the task force.

7.3.13 Miscellaneous

For some applications it is necessary to associate some additional data with every object controlled by the Resource Management library. For example, given a task it may be desirable to store the most suitable processor to date for running that task. The library permits two additional integer-sized data values to be associated with every object.

```
int RmSetProcessorPrivate(  RmProcessor, int);
int RmGetProcessorPrivate(  RmProcessor);
int RmSetProcessorPrivate2( RmProcessor, int);
int RmGetProcessorPrivate2( RmProcessor);
int RmSetNetworkPrivate(    RmProcessor, int);
int RmGetNetworkPrivate(    RmProcessor);
int RmSetNetworkPrivate2(   RmProcessor, int);
int RmGetNetworkPrivate2(   RmProcessor);
int RmSetTaskPrivate(       RmProcessor, int);
int RmGetTaskPrivate(       RmProcessor);
int RmSetTaskPrivate2(      RmProcessor, int);
int RmGetTaskPrivate2(      RmProcessor);
int RmSetTaskforcePrivate(  RmProcessor, int);
int RmGetTaskforcePrivate(  RmProcessor);
```
```

```
int RmSetTaskforcePrivate2( RmProcessor, int);
int RmGetTaskforcePrivate2( RmProcessor);
```

It is common practice to use a pointer to an application-specific data structure as the private value, allowing arbitrary data to be associated with the object.

7.3.14 Error handling

The Resource Management library has its own error handling system. This is based around a small number of error codes, which are simple integers like the Unix error codes. The library has an integer variable **RmErrno** which is set to one of the codes when an error occurs. The available error codes are given in the header file **rmlib.h**, but include the following.

1. **RmE_Success**. The operation was successful. This number is always 0.
2. **RmE_NotProcessor**. An argument passed to the routine was not a valid **RmProcessor** value.
3. **RmE_NotNetwork**. An argument was not a valid **RmNetwork** value.
4. **RmE_NotTask**. An argument was not a valid **RmTask** value.
5. **RmE_NotTaskforce**. An argument was not a valid **RmTaskforce** structure.
6. **RmE_WrongNetwork**. A network argument was inappropriate. For example, an attempt was made to connect a processor in one network to a processor in a different network.
7. **RmE_WrongTaskforce**. A task force argument was inappropriate. For example, an attempt was made to connect a task in one task force to a task in a different task force.
8. **RmE_InUse**. A particular object is currently in use. For example, an attempt might be made to release a processor while the program still has a task running on that processor.
9. **RmE_Corruption**. Some corruption of the library's data structures has been detected.
10. **RmE_BadArgument**. One of the arguments was invalid. For example, a processor type used in a call to **RmSetProcessorType()** was a number not corresponding to any known processors.
11. **RmE_ReadOnly**. An attempt was made to write to a file to which the program does not have access.
12. **RmE_NoMemory**. The local processor does not have enough memory to perform the requested operation.
13. **RmE_NotFound**. A search failed, for example a search is made for a particular processor attribute which did not match any of the existing attributes.

Many of the library routines return an error code, or **RmE_Success** to indicate success. The value of **RmE_Success** is defined to be zero, so that a non-zero return code indicates an error. When an error occurs the error code is usually stored with the object, in addition to being returned. This allows the error code to be retrieved at a later stage. It is particularly useful when an operation on a collection of objects has failed. For example, suppose a program attempts to execute a mapped task force but two of the component tasks cannot be started because there is insufficient memory on the target processor. The routine can return at most one error code, so it can be difficult to analyse what happened after a failure. Storing the various error codes with the tasks allows the program to search the task force after a failure and find out which tasks could not be started and why. The routines for extracting and clearing error codes in an object are:

```
int RmGetProcessorError(   RmProcessor);
int RmClearProcessorError( RmProcessor);
int RmGetNetworkError(     RmNetwork);
int RmClearNetworkError(   RmNetwork);
int RmGetTaskError(        RmTask);
int RmClearTaskError(      RmTask);
int RmGetTaskforceError(   RmTaskforce);
int RmClearTaskforceError( RmTaskforce);
```

A final routine is provided to map the error codes onto strings. This is useful when producing diagnostics.

```
const char *RmMapErrorToString(int);
```

7.4 Example programs

This section gives two example programs to illustrate the use of the Resource Management library. Full sources of these and other examples are shipped with the Helios system, in the directory **/helios/users/guest/examples/rmlib**.

The first example program is **owners**, which performs basically the same function as the **network owners** command. It examines the current network state and displays usage information about the network.

The second program is **mappipe**. This program can be used to map pipeline task forces optimally in an arbitrary network. The program constructs a task force in the form of a pipeline. Then it obtains a copy of the current network and searches this network for a set of processors connected in a pipeline of the required length. Finally it executes the task force in this pipeline of processors. This program illustrates one of the ways in which user applications can attempt to solve the mapping problem.

7.5 Owners

In a multi-user network the various processors can be owned by various different users at any one time. In addition some of the processors may be in the free pool, and others may be reserved for use by the system. The **owners** program is a cut-down version of the **network owners** command. Its operation is defined as follows.

1. A data structure is needed to hold details of the various users, including the user id and the number of processors currently owned. There can be any number of users so these details are best held in a linked list. The following data structure is used:

```
typedef struct OwnerDetails {
        Node   Node;
        int    Owner;
        int    Count;
} OwnerDetails;
```

   A static variable **OwnerList** is a linked list holding all the individual user list nodes. Helios supplies various useful linked list routines automatically, and these are defined in the header file **queue.h**.

2. A copy of the current network state is obtained from the Network server, so that this copy can be analysed.

3. A routine is applied to every processor in the network, to update the owners list.

4. The results in the owner list are displayed.

This program can be compiled with the command line shown below. The other programs in this section can be compiled in much the same way.

```
c -oowners -lRm owners.c
```

The first part of the program would look something like this:

```
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <queue.h>
#include <rmlib.h>

typedef struct OwnerDetails {
        Node Node;
        int  Owner;
        int  Count;
} OwnerDetails;

static  List    OwnerList;
static  int     NumberProcessors;
static  int     NetworkWalk(RmProcessor Processor, ...);
static  WORD    ShowOwners(Node *node, WORD arg);
static  WORD    MatchOwner(Node *node, WORD arg);

int main(void)
{ RmNetwork Network;

  InitList(&(OwnerList));
    /* Get details of the current network into local memory */
```

```
  Network = RmGetNetwork();

  NumberProcessors = RmCountProcessors(Network);
      /* Walk down the current network examining every processor */
      /* Build the ownership list as the program goes along.     */
  (void) RmApplyProcessors(Network, &NetworkWalk);
      /* Output the results by walking down the owner list. */
  (void) WalkList(&OwnerList, &ShowOwners);
  return(EXIT_SUCCESS);
}
```

This part is rather straightforward. A number of C header files are needed to avoid compilation warnings or errors. The data structure used to hold the results is defined, and two static variables are declared. The first variable is a linked list to hold the various user nodes. The second variable is a count of all the processors in the network so that the final display of results can print out percentages as well as absolute numbers. Next there are three function declarations. The first is used as an argument to **RmApplyProcessors()**, and the next two are used as arguments to the Kernel's list walking routines.

Routine **main()** does very little. The linked list has to be initialised before it can be used, or memory corruption will result. The Resource Management library is used to obtain a copy of the current network state. This is then examined using two more **RmLib** calls, and finally the results are displayed.

The routine **NetworkWalk()** is applied to every processor in the network. It extracts the current owner of the processor, which might be a real user, the network's free pool, or the system itself. Then it examines the list of currently known owners using one of the Kernel's list walking routines. If the program has already encountered a processor owned by the same user, there will be an existing **OwnerDetails** structure that can be updated. Otherwise a new structure must be obtained dynamically, initialised, and added to the list of currently known users.

```
  /* This routine is called for every processor in the network. */
static int NetworkWalk(RmProcessor Processor, ...)
{ int          Owner;
  OwnerDetails *details;

    /* Get the current processor owner, and see if this owner */
    /* is already known.                                      */
  Owner    = RmGetProcessorOwner(Processor);
  details  = (OwnerDetails *)
             SearchList(&OwnerList, &MatchOwner, Owner);

    /* If the user is already known, the search will have  */
    /* found an OwnerDetails structure that can be updated.    */
    /* Else a new structure must be allocated and initialised. */
  if (details != (OwnerDetails *) NULL)
   details->Count++;
  else
   { details = (OwnerDetails *) malloc(sizeof(OwnerDetails));
```

```
      details->Owner = Owner;
      details->Count = 1;
      AddTail(&(OwnerList), &(details->Node));
    }

  return(0);
}
```

Given an owner identifier returned by **RmGetProcessorOwner()** it is necessary to check whether or not this owner has been encountered already. If so, there will be a list node with a matching identifier, so it is necessary to search the list of known owners for this identifier. The Kernel's **SearchList()** routine can be used for this.

```
   /* Match a processor's owner with an entry in the current */
   /* list of owners.                                        */
static WORD MatchOwner(Node *node, WORD arg)
{ OwnerDetails *details = (OwnerDetails *) node;

  if (details->Owner == arg)
   return(1);
  else
   return(0);
}
```

Finally, once every processor in the network has been processed, it is necessary to output the results. This is achieved by walking down the list of owners that have been encountered and printing out a suitable line. The information held in the **OwnerDetails** data structure is an integer user identifier, but the Resource Management library contains a routine to translate this to a text string. The routine below displays this string, a count of the number of owned processors which is assumed to be less than a thousand, and a percentage of the network. Note that this percentage is approximate as no attempt is made to cope with rounding errors, and hence the total number of processors may not add up to 100 percent.

```
  /* Print out the results of the search. */
static WORD ShowOwners(Node *node, WORD arg)
{ OwnerDetails *details = (OwnerDetails *) node;

  printf("%-10s : %3d processors, %2d%% of the network\n",
        RmWhoIs(details->Owner), details->Count,
        (details->Count * 100) / NumberProcessors);
  return(0);
}
```

The **network** command contains essentially this code to handle the **owners** option. It is slightly more advanced in that it treats I/O processors and router processors separately, but that feature could be added fairly easily to the program.

7.6 Mappipe

One of the most important problems in parallel programming is the mapping problem. Given a task force (a collection of tasks or programs that communicate with each

other) and given a network or collection of processors that are connected in some arbitrary topology, work out which tasks to put on to which processors. Usually it is necessary to match the channel connections between the component tasks as closely as possible with the link connections between the processors. For example, if a task A communicates with three other tasks, the task force should be mapped so that the three other tasks are on processors connected directly to the processor running task A.

For the general case, solving the mapping problem is very difficult. Specifically, the problem is NP-complete: adding one more task to the task force, or one more processor to the network, could double the time taken to perform the mapping. This means that the time taken to do a mapping becomes very long for all but trivial problems, and it is necessary to use heuristics in order to get a mapping that is reasonably efficient, but not optimal, within a reasonable time.

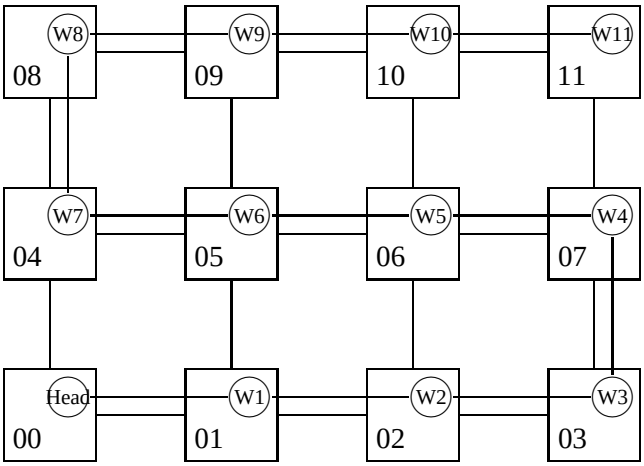


Figure 7.11: Mapping a pipeline of tasks

Under Helios, solving the mapping problem is mostly a user problem. The Task Force Manager does have a simple mapping algorithm built-in, and hence it can run user task forces without any extra effort on the user's part. However, the Task Force Manager makes no guarantees about performance and it is possible that the default mapping is not optimal for the application. One of the jobs of the Resource Management library is to permit users to produce their own mapping algorithms, typically tuned specifically to one task force topology. The **mappipe** program gives an example of how this could be done. It can be used for task forces in the form of a pipeline, which have a special program at the start of the pipeline and then a number of workers. It is necessary to find a set of processors also connected in a pipeline. This is illustrated in Figure 7.11. The diagram illustrates a pipeline of twelve tasks, with one program as the head and eleven workers. This pipeline is mapped onto a network of 12 processors in an optimal fashion, because there is a separate link to handle the traffic for each channel. A mapping such as this could be achieved by running **mappipe** with the argument 12.

The initial part of **mappipe** is shown below.

```
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <queue.h>
#include <rmlib.h>

static RmTaskforce build_taskforce(int);
static RmNetwork   obtain_network(RmTaskforce);
static int         execute_taskforce(RmNetwork, RmTaskforce);

int main(int argc, char **argv)
{ RmTaskforce pipeline;
  RmNetwork   processors;
  int         number_tasks;

  number_tasks = atoi(argv[1]);

  pipeline = build_taskforce(number_tasks);

  processors = obtain_network(pipeline);

  return(execute_taskforce(processors, pipeline));
}
```

Again there is the usual set of header files which must be included, followed by a number of forward declarations. The real work is done in three separate routines. The first constructs an **RmTaskforce** structure representing the application to be executed. The second does most of the work, examining the network for a set of processors meeting the requirements, obtaining these processors, and specifying which parts of the application should run where. The final routine simply executes the task force and waits for it to finish. Of course a full program should check for failures such as being unable to map the pipeline onto the current network.

The code that constructs the task force is shown below.

```
static RmTaskforce build_taskforce(int number_tasks)
{ RmTaskforce result = RmNewTaskforce();
  RmTask      previous, current;
  int         i;

  RmSetTaskforceId(result, "pipeline");

  previous = RmNewTask();
  RmSetTaskId(previous, "start");
  RmSetTaskCode(previous, "/helios/bin/ps");
  RmAddTaskArgument(previous, 1, "all");
  RmAddtailTask(result, previous);

  for (i = 1; i < number_tasks; i++)
   { char buf[16];
     current = RmNewTask();
     sprintf(buf, "worker%d", i);
```

```
      RmSetTaskId(current, buf);
      RmSetTaskCode(current, "/helios/bin/cat");
      RmAddtailTask(result, current);
      RmMakeChannel(previous, 1, current, 0);
      previous = current;
    }
  return(result);
}
```

The component tasks have to be held in an **RmTaskforce** structure which is allocated and given an arbitrary name. Then an **RmTask** structure is allocated for the head of the pipeline, and this is added to the task force. In this example the head of the pipeline is the program **ps**, and it is given a single argument **all**. Of course in practice the head of the pipeline would be a user program, probably passed as an extra argument to the **mappipe** program. Finally the various workers are built and added to the pipeline. These workers currently consist of the program **cat** but again a user program would normally be substituted here. For every worker the standard input stream is connected to the standard output of the previous worker.

Building the task force pipeline is relatively easy. Finding a matching pipeline of processors is not so easy. The basic algorithm is as follows:

1. Work out how big the pipeline of processors should be.

2. Build an **RmNetwork** data structure to hold the obtained network, plus an additional vector.

3. Obtain a copy of the current state of the network from the Network server. Without this information it is not possible to perform a sensible mapping because the program has no idea about the network topology, which processor is currently free and so on.

4. Check that the network is big enough to run the pipeline. Attempting to map 65 programs in a 64-processor network is doomed to failure, but it might take the mapping program a long time to work this out.

5. Find a possible starting place for the pipeline. This must be a processor which has not been tried before and which must meet certain requirements, for example it must be a T800 with at least one megabyte of memory.

6. An attempt is made to obtain the starting place. Note that by the time the search reaches this processor it may not be possible to obtain the processor anymore because another user may have claimed it. The copy of the network obtained earlier was a snap-shot which may become out of date at any time.

7. A vector is used to hold the pipeline of processors as it is being built. The starting place is put at the start of the pipeline. Then a call is made to **add_to_pipeline()** which looks for processors connected to the current starting place and adds them to the pipeline. This routine is recursive and does the main job of the network search.

8. If **add_to_pipeline()** succeeds, the whole pipeline has been obtained. It is now possible to map the task force on to the pipeline of processors, so that it can be executed.

9. If the pipeline cannot be built from this starting place, it is necessary to try again with a different processor, releasing any resources.

```
typedef struct {
        RmProcessor Template;
        RmProcessor Obtained;
} ProcessorVector;

static bool add_to_pipeline(RmNetwork, ProcessorVector *,
                            int, int);
static int  find_starting_place(RmProcessor, ...);
static void map_taskforce(ProcessorVector *, RmTaskforce);

static RmNetwork obtain_network(RmTaskforce taskforce)
{ RmNetwork       result;
  ProcessorVector *vector;
  int             number_processors;
  RmNetwork       whole_network;
  RmProcessor     starting_place;

  number_processors = RmCountTasks(taskforce);
  vector = Malloc(number_processors * sizeof(ProcessorVector));

  result = RmNewNetwork();
  RmSetNetworkId(result, "Obtained");

  whole_network = RmGetNetwork();

  if (RmCountProcessors(whole_network) < number_processors)
   { fputs("mappipe: not enough processors in the network.\n",
           stderr);
     exit(EXIT_FAILURE);
   }

  for (starting_place = (RmProcessor)
          RmSearchProcessors(whole_network, &find_starting_place);
       starting_place != (RmProcessor) NULL;
       starting_place = (RmProcessor)
          RmSearchProcessors(whole_network, &find_starting_place))
   { vector[0].Template = starting_place;
     vector[0].Obtained = RmObtainProcessor(starting_place);
     if (vector[0].Obtained == (RmProcessor) NULL) continue;
     RmAddtailProcessor(result, vector[0].Obtained);

        /* If this recursive call succeeds, the whole */
        /* pipeline has been mapped                   */
     if (add_to_pipeline(result, vector, 1, number_processors))
      { map_taskforce(vector, taskforce);
```

```
        return(result);
      }

        /* Failure, clean up and try another starting place */
     RmRemoveProcessor(vector[0].Obtained);
     RmReleaseProcessor(vector[0].Obtained);
     RmFreeProcessor(vector[0].Obtained);
   }

  return((RmNetwork) NULL);
}
```

One of the problems in a mapping program is finding somewhere to start. This routine is applied to every processor in the network, every time around the main loop, to check whether or not the processor is suitable. Several requirements must be met.

1. The processor must not have been used as a starting place before. The private field functions are used to keep track of this.

2. In this example a starting processor should be a T800 with at least one megabyte of memory. The exact requirements are specific to the application.

3. The processor must be an ordinary Helios processor, not a native processor or an I/O processor.

4. The processor should be in the free pool at the moment. In fact this restriction is too severe because it does not allow for processors already in the user's domain, but that is a minor complication.

If the current processor is a suitable starting place, its private field is updated to reflect this, to ensure that it will not be used as a starting place at some future time.

```
static int find_starting_place(RmProcessor processor, ...)
{
  if ((RmGetProcessorPrivate(processor) != 0)          ||
      (RmGetProcessorType(processor)    != RmT_T800)   ||
      (RmGetProcessorMemory(processor)  <  0x100000)   ||
      ((RmGetProcessorPurpose(processor) & RmP_Mask)
                                        != RmP_Helios) ||
      (RmGetProcessorOwner(processor)   != RmO_FreePool))
   return(0);

  RmSetProcessorPrivate(processor, 1);
  return((int) processor);
}
```

The main search routine of the **mappipe** program is **add_to_pipeline()**. This routine is called when there are one more processors already in the pipeline and it is necessary to find the next one.

1. If the end of the pipeline has been reached, the search has completed successfully and the application can be run.

2. Otherwise check the various neighbours of the processor currently at the end of the vector.

3. If a link is not connected or goes to a processor in another network ignore it.

4. If the neighbour at the end of a link does not meet the requirements of the various workers, ignore it.

5. If the neighbour is already in the pipeline ignore it, because it is undesirable to run two workers on the same processor.

6. If the neighbour cannot be obtained, for example because another user has obtained it while the search has been going on, ignore it.

7. Add the processor to the end of the pipeline and call this routine recursively, to find the next processor in the pipeline. If the rest of the search succeeds, the whole search has succeeded. Otherwise it is necessary to undo some work, try the next neighbour if any, and finally report failure.

Clearly this search mechanism is not very useful except for the pipelines. It could be extended to rings by adding a test at the end of the pipeline, to ensure that the final processor is adjacent to the first one. Grids, farms, and other topologies would need very different search algorithms.

```
static bool add_to_pipeline(RmNetwork result,
       ProcessorVector *vector, int position, int max)
{ RmProcessor previous = vector[position - 1].Template;
  int         number_links = RmCountLinks(previous);
  int         i, j, destlink;
  RmProcessor neighbour;

  if (position == max) return(TRUE);

  for (i = 0; i < number_links; i++)
   { neighbour = RmFollowLink(previous, i, &destlink);
     if ((neighbour == RmM_NoProcessor) ||
         (neighbour == RmM_ExternalProcessor))
      continue;

     if ((RmGetProcessorType(neighbour)    != RmT_T800)    ||
         (RmGetProcessorMemory(neighbour)  <  0x100000)    ||
         ((RmGetProcessorPurpose(neighbour) & RmP_Mask)
                                           != RmP_Helios) ||
         (RmGetProcessorOwner(neighbour)   != RmO_FreePool))
      continue;

     for (j = 0; j < position; j++)
      if (neighbour == vector[j].Template)
```

```
       goto skip;

     vector[position].Template = neighbour;
     vector[position].Obtained = RmObtainProcessor(neighbour);
     if (vector[position].Obtained == (RmProcessor) NULL)
      continue;

     RmAddtailProcessor(result, vector[position].Obtained);
     if (add_to_pipeline(result, vector, position + 1, max))
      return(TRUE);
     RmRemoveProcessor(vector[position].Obtained);
     RmReleaseProcessor(vector[position].Obtained);
     RmFreeProcessor(vector[position].Obtained);
skip:
     continue;
   }

  return(FALSE);
}
```

Once all the required processors have been obtained it is possible to map the task force onto the processors. The processors have been put into a vector, and the tasks have been placed in the right order inside the task force. Hence the mapping is relatively straightforward, and simply involves walking down the component tasks within the task force.

```
static void map_taskforce(ProcessorVector *vector,
                          RmTaskforce taskforce)
{ RmTask task = RmFirstTask(taskforce);
  int    i = 0;

  for ( ; task != (RmTask) NULL; task = RmNextTask(task))
   { printf("Mapped task %s onto processor %s\n", RmGetTaskId(task),
            RmGetProcessorId(vector[i++].Obtained));
     RmMapTask(vector[i].Obtained, task);
   }
}
```

Finally it is necessary to execute the task force. The most important conditions have been met: a suitable processor has been obtained from the system for every component task; the processors and the tasks have been collected together into the right data structures; the tasks have all been mapped onto processors. Hence the description of the application and processors can now be passed on to the Task Force Manager, so that the application can be run.

```
static int execute_taskforce(RmNetwork processors,
                             RmTaskforce taskforce)
{ RmTaskforce running;

  running = RmExecuteTaskforce(processors, taskforce, Null(char *));
  if (running == (RmTaskforce) NULL)
```

```
   fprintf(stderr, "mappipe: failed to run taskforce, fault %s\n",
             RmMapErrorToString(RmErrno));
  else
   RmWaitforTaskforce(running);

}
```

There are many areas where this program could be improved. At present it performs an exhaustive search of the network, even if there is no way for the task force to be executed. A useful initial test would be to see whether or not there are enough free processors in the network matching the application's requirements, simply by checking every processor once and removing processors that do not match the requirements. Removing these processors might result in various disconnected networks. Hence it could be worthwhile to calculate one or more minimum spanning trees to find out the largest collection of processors that are connected. Currently if a search fails to obtain a processor once it may try to obtain the same processor again later. Marking such processors using the private fields may produce a significant increase in speed. However, such improvements would result in a program too complex to use as an example here.

Chapter 8

The I/O server

8.1 Introduction

At the time of writing, nearly all Helios networks consist of one or more processors which are plugged into some existing host computer, such as a Unix workstation, an IBM PC or compatible equipment. The processors are usually equipped with little or no input/output hardware of their own. Instead most I/O must go through the host computer. To boot up the network, a program is run on this host computer which resets the network, sends a Nucleus down a link, and performs I/O for all processors in the network. Under Helios this program is known as the **I/O server**.

The Helios I/O server runs on many different computers, ranging from 8086 based PC clones to Unix workstations equipped with fast hard discs, ethernet connections, high resolution displays running the X window system, and so on. Approximately 80 to 90 percent of the I/O server source code is the same for all computers, the remainder being machine specific. In particular every attempt has been made to keep the user interface similar on all machines, while still making full use of the available hardware facilities. This chapter describes in detail the PC and Sun versions of the I/O server, but most of it is also applicable to other versions.

It begins with a general description of I/O in computers, ranging from mainframes to personal computers. This is followed by details of typical Transputer hardware, and an outline of the Helios client server model.

There is a general description of the I/O server, including details of the command line options, the **host.con** configuration file, and the debugging facilities. The I/O server is a very flexible piece of software, and this is to give users some idea of what can be achieved.

The I/O server has an inbuilt Transputer debugger, which allows experienced users to perform post-mortem analysis when a Transputer crashes. This debugger is aimed primarily at experienced users and in particular at Helios system programmers, but its facilities are available to all users who want them. There is a description of the debugger, its various commands and their uses.

There are detailed descriptions of the PC and Sun I/O servers respectively. Other versions of the I/O server should be supplied with their own documentation, giving any machine specific details. Readers should be aware that some parts of this documentation are quite complicated, usually because the underlying hardware itself can

be complicated, but most users will not need to understand how the hardware works in great detail.

Appendix A gives a short summary of the various debugging options of the I/O server, and a list of the options of the configuration file which can be produced by the I/O server.

8.2 The role of the I/O server

This section describes the role of the I/O server in a Helios network. Firstly, how input/output is handled in more traditional machines, and what hardware is actually available. Then, a typical Helios network is described. There is also an explanation of the Helios client-server model, which is fundamental to your understanding of the I/O server.

8.3 I/O in more conventional machines

Before the 1980s, the dominant types of computers were mainframes and minis. Typically these were powerful single-processor machines, which were equipped with large, fast, hard discs, tape systems, and perhaps a machine for punched cards.

Many mainframes were used primarily for batch jobs, with little or no user interaction. Others were equipped with dumb terminals. Most of the input and output was actually handled by front end processors, to avoid overloading the central processor with interrupts for every key. These would buffer whole lines from the terminals before reactivating the user's session in the central processor. Ordinary users had very little control over the hardware. Figure 8.1 illustrates this sort of machine.

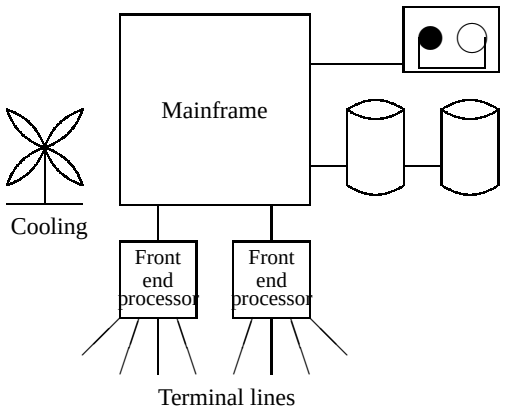


Figure **8.1** A mainframe computer

The first personal computer was launched in 1975. There was no screen, no keyboard, no disc for permanent storage, and certainly no programming language or operating system. However, it could sit comfortably on a desk and did not require a special computer room. It was of little practical use, unless you were a computer enthusiast.

During the next few years more powerful processors were developed. In particular, Intel[1] released the 8086 in 1978 and Motorola [2] released the 68000 in 1979. These 16-bit processors were significantly more powerful than the previous 8-bit processors, and they are used as the basis for present day personal computers.

Some of the personal computers now available are the IBM PC, AT and compatibles, the Commodore Amiga [3], the Apple Macintosh [4] and the Atari [5] ST. Although they each have different operating systems, all the machines share certain characteristics. These characteristics must be provided for in the design of the Helios I/O server.

1. There is an initial ROM bootstrap to initialise the main processor and various pieces of hardware.

2. The machine is equipped with at least one floppy disc drive and there is the option of a hard disc.

3. Following the ROM bootstrap an operating system is read from the disc (either floppy disc or hard disc).

4. The machine is equipped with its own display, capable of using at least 25 rows by 80 columns, and in some cases providing a graphic mouse driver user interface.

5. Also included in the package is a keyboard, one or more serial ports, one or more parallel printer ports, and often a mouse.

6. There are usually a number of expansion slots, to allow users to plug in extra hardware.

All the hardware is managed either by the operating system or by additional programs such as device drivers, which can be loaded on top of the operating system. In particular, when extra I/O hardware is plugged into an expansion slot, another device driver is added to the system configuration. However, the operating system does not always provide access to all the facilities of the I/O hardware or allow maximum throughput. On most machines, users' applications can take over control of the I/O hardware from the operating system and drive the hardware directly, not necessarily by 'legal' means. A typical personal computer is illustrated in Figure 8.2.

Present day personal computers are now more powerful than early mainframes. There is also a third type of computer, the workstation, and this has a processing capability less than a mainframe. Typical workstations run some dialect of the Unix operating system. The standard hardware of a workstation includes a hard disc, keyboard, ethernet connection, two serial ports, a mouse, and often a high resolution display. The display usually runs the X window system or another windowing system. A workstation still boots from ROM and then reads operating system code from disc or from the ethernet. All I/O is handled by the operating system, and it is very difficult to access hardware directly in a user application. A typical workstation is shown in Figure 8.3.

[1] Registered trademark of Intel Corporation
[2] Trademark of Motorola, Inc.
[3] Registered trademark of Commodore-Amiga, Inc.
[4] Registered trademarks of Apple Computers, Inc.
[5] Trademark of Atari Corporation

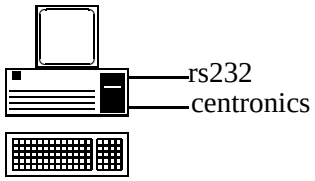


Figure **8.2** A personal computer

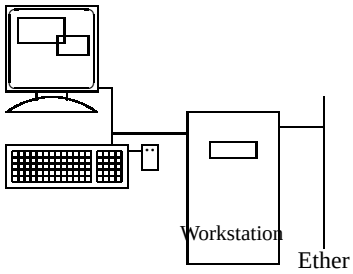


Figure **8.3** A workstation

Helios is not machine dependent. The I/O server must therefore be able to deal with the operating systems and I/O hardware of many different types of computer. For this reason, it must be extremely adaptable. The I/O server can be configured in many different ways, depending on the user's particular configuration.

8.3.1 Transputer hardware

The Inmos T414 Transputer was launched in September 1985, followed by the T800 in 1987. These could have been used as the basis for a new personal computer or workstation. Instead, Inmos produced 'evaluation boards' like the B004, designed to plug into existing computers. These existing computers are known as **host computers** or **I/O processors**.

The B004 is equipped with a single Transputer, a megabyte of memory, and a link adapter which allows the PC host to interact with the Transputer over a Transputer link. The board has no I/O facilities of its own: it relies on the host. This affects the I/O performance, since the link adapter is slow. If the link I/O code is tuned, code transfer rates of 200 Kbytes per second can be achieved, but this is slower than communication between Transputers. Designing Transputer boards like the B004 is easy, because of the small number of components involved. Many manufacturers have produced such boards for IBM PCs and compatibles, Sun workstations, other Unix workstations, the Commodore Amiga, and other popular machines. Some boards have more memory than others, some have special hardware instead of a standard link adapter to achieve greater transfer rates, and some cost more than others. Essentially there is very little difference. A Transputer network based on such a board is shown in Figure 8.4. The I/O processor shown in Figure 8.4 is in a PC, but it could be in any of the machines

mentioned earlier.

The I/O processor is connected by a link adapter to the **root Transputer** identified as `00`. Some boards contain more than one Transputer. Others, such as the Inmos B008 board allow Transputer modules (trams) to be plugged in. Larger networks have separate external racks of processors, such as the Parsytec SuperCluster or Telmat T.Node.

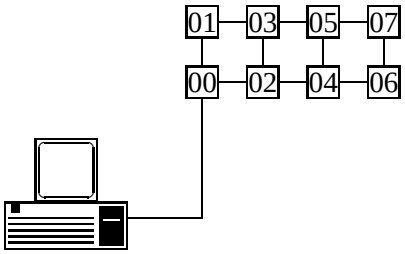


Figure **8.4** Simple network

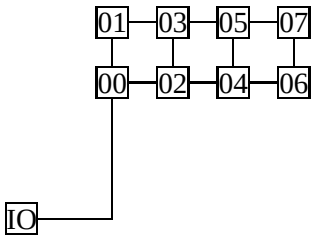


Figure **8.5** A simple network

The I/O server runs on the I/O processor and provides I/O services for the network processors in the system. Comparing input/output through an I/O server with input/output on a mainframe or personal computer is useful. To some extent the I/O processor may be thought of as a front end processor for a more powerful machine, which in this case is a processor network rather than a mainframe. The hardware facilities provided by the I/O processor are the same as those of a personal computer or workstation, including serial lines and mice. However, user applications run on a completely different processor, so they have absolutely no way to access the host computer's I/O hardware directly. Instead the I/O server must supply sufficient flexibility to meet all possible application requirements. With a device like a serial line this means supporting modem ring signals and various baud rates, not just simple reading and writing of data.

To avoid the bottleneck of the single link adapter there is a trend towards moving I/O facilities into the processor network. Transputer graphics modules can provide high resolution displays and the T800 and T425 'block move' instructions allow fast graphics operations. Disc modules with SCSI interfaces allow hard discs and tape units to be plugged into the network. Ethernet modules enable the processor network to become part of a local area network (see Figure 8.6).

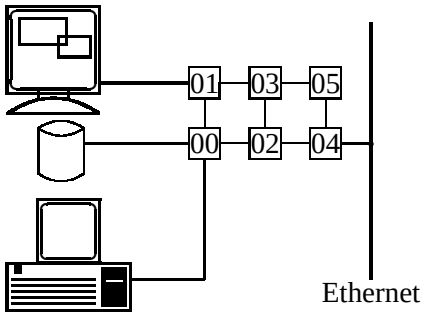


Figure **8.6** A more complicated network

It is important to remember that Helios is a general purpose operating system, and is not Transputer specific. At the time of writing Helios already runs on the Acorn ARM and on the Motorola 68020, as well as Transputers. From now on the term 'processor' will be used to refer to any processor in a Helios network running Helios, not necessarily a Transputer. The I/O processor, or processors, are also part of the Helios network, but run the I/O server rather than Helios.

8.3.2 The role of the I/O server

The exact role of the I/O server can now be explained. The I/O server runs on the host computer or I/O processor, and makes it behave just like any other processor in the network, although it only has one link. The I/O processor has a name like any other processor, which is usually **/Cluster/IO**. At the lowest level, it accepts and generates the protocols on the link (a task carried out by link guardians within the Helios Nucleus). At a higher level, it accepts and responds to search requests from other processors in the system. On the other processors, this task is carried out by the link IOC controllers of the Processor Manager. In addition to the functions of the link IOC controller, the I/O server accepts a number of special messages, such as **IOdebug()** requests. [6]

At a higher level still, the I/O processor (**/Cluster/IO**) contains a number of different servers: file servers, a window server, an error logging server, and so on. These servers behave like any other server in the Helios network, accepting messages according to the General Server Protocol and sending back suitable replies. Apart from the

[6] **IOdebug()** is a low-level debugging routine which works by sending messages directly to the I/O server. It works at a much lower level than C library **printf()** routines, and hence is more likely to work in spite of such problems as memory corruption.

response time, clients cannot distinguish between a service provided by the I/O server and a service inside the rest of the network.

The I/O server has one other useful function: it performs the initial bootstrap of the root processor. Once the root processor has been booted the I/O server becomes a passive object, responding only to requests from the rest of the network rather than generating messages. All 'intelligent' programs, shells, utilities and user applications run inside the network and not on the I/O processor.

The exact services provided by the I/O server vary from machine to machine. The following list is for the PC version of the I/O server, but it is quite typical.

1. **/helios**. This is a file server containing the main Helios system files. This server usually contains the subdirectories **bin** for the default set of utilities, **lib** for the system's internal files, **etc** for configuration files, and **include** for C header files. All Helios networks contain at least one of these servers.

2. **/logger**. This is an error logging service. Depending on the configuration of the I/O server, data sent to the **/logger** device will be displayed on the PC screen or written to a file for later inspection (or both).

3. **/window**. This is a windowing server supporting multiple text windows on a standard 25 by 80 character display, with a hot key switching mechanism to move from one window to the next. These windows are usually referred to as pseudo-windows or server windows.

4. **/clock**. The I/O processor is usually equipped with a battery-backed real time clock (unlike the other processors in the network). This server allows Helios clients to access the real time clock.

5. **/pc**. This provides a limited communication facility between a program running on the PC and programs running in the Helios network.

6. **/centronics**. This allows access to the PC's parallel ports.

7. **/rs232**. This allows access to the PC's serial ports.

8. **/printers**. This allows access to printers generally (parallel and serial printers).

9. **/mouse**. This is an interface to a mouse device driver on the PC side, allowing most PC mice to be used.

10. **/keyboard**. This is a raw keyboard device. Reading data from a window server gives standard ASCII input, with the window server dealing with input processing, such as auto repeat. The raw keyboard device generates events whenever a key is pressed or released. For example, when reading from a window server, pressing and releasing the space bar generates a single character `0x20`. Holding the space bar a long time would generate a number of `0x20` characters, as the auto repeat starts up. With the raw keyboard device, pressing the space bar generates a single event, **scancode 0x39 down**. Holding the space bar has no effect, but releasing it again generates the event **scancode 0x39 up**.

11. **/a**. A file server to provide access to the floppy disc **a:**.

12. **/b**. A file server to provide access to the floppy disc **b:**.
13. **/c**. A file server for the **c:** partition of the hard disc.
14. **/d**. A file server for the **d:** partition.
15. **/rawdisk**. A low-level file server which allows Helios applications (usually the Helios file server) to read and write disc blocks directly.

Comprehensive details of the servers provided by specific implementations of the I/O server can be found in the PC I/O server section, in the Sun I/O server section, or in the release notes for any other version of the I/O server.

8.4 The I/O server options

With most implementations of the Helios I/O server the I/O server is run by executing the command **server** on the host computer. The **server** command may be the I/O server itself, or it may be a shell script or other program that causes the I/O server to be run. The purpose of this section is to describe the common features of most versions of the I/O server.

1. The **server** command line options.
2. The **host.con** configuration file.
3. The debugging facilities.

There are reference summaries of these features in appendix A.

8.4.1 The command line

The command line options for the **server** command are:

```
 server [-a] [-<debugflags>] [-d] [-e] [-C config file]
[-c config file] [+<options>] [nucleus]
```

The -a option

The **-a** option enables all debugging options in the I/O server. This is not necessarily a good idea, since there are rather a lot of them.

8.4.2 Debug options

Individual debugging options can be enabled by giving a simple list, using one letter per debugging option. For example, the command

```
server -biq
```

enables the bootstrap, the initialisation, and termination options. Three of the debugging options: **delete**, **communications**, and **errors**, which use the letters **d**, **c**, and **e** respectively, cannot be enabled by themselves because the letters clash with the other command line arguments. For example, the command:

```
server -d
```

runs the I/O server in its built-in debugger mode, rather than in normal mode with the **delete** option enabled.

The -d option

The **-d** option is used to run the built-in debugger rather than the normal I/O server.

The -e option

An **-e** argument can be used to enable a link connected to a network already running Helios, rather than boot up a processor. It is similar to the **enable_link** option in the **host.con** configuration file.

The -c or -C option

The **-c** or **-C** options can be used to specify a particular configuration file. The configuration file gives Helios details about the network and host computer. By default the I/O server reads the configuration file **host.con** from the current directory. The **-c** option is particularly useful when combined with shell aliases. The following are typical examples:

```
alias helios1 "/usr/helios/server -c/users/root/helios/host.1"
alias helios2 "/usr/helios/server -C /users/root/helios/host.2"
```

These lines define two ways of starting up Helios from anywhere in the host filing system, using two different configuration files **host.1** and **host.2**. One line might start up Helios with the X window system server running in the processor network and the other might use pseudo windows running in the I/O processor.

Configuration options

The **host.con** file is one of the most important configuration files in Helios, and tends to be edited frequently when a user is setting up a working environment. To ease this process slightly the command line can be used to add **host.con** entries. For example, the command line:

```
server +Server_windows
```

runs the I/O server as if the flag **Server_windows** has been added to the **host.con** file. If the option was in the configuration file anyway then the command line addition has no effect.

The command line can also be used to override options in the configuration file. If the **host.con** file had the following line:

```
Helios_directory = c:\helios
```

and the command line was

```
server +Helios_directory=c:\helios.new
```

then the command line option takes precedence and the directory `c:\helios.new` will be used.

The Nucleus option

The final option specifies the Nucleus to be used when booting up the root processor, overriding the default entry in the **host.con** file. Hence the following two commands are equivalent.

```
server /usr/helios/lib/nucleus.dbg
server +system_image=/usr/helios/lib/nucleus.dbg
```

This option is used mainly by system programmers who can produce their own versions of the Helios Nucleus, and is of little or no interest to ordinary users.

8.4.3 The host.con file

After processing the command line arguments the I/O server reads in a configuration file. Most of the I/O server's flexibility comes from this configuration file. By default the file **host.con** in the current directory is used, but an alternative can be specified on the command line using the **-c** or **-C** option. The configuration file is an ordinary text file containing flags and variables, one per line. Typical lines might look like this:

```
# This is a comment
Server_windows
Helios_directory = /usr/helios
```

As with most Helios configuration files, comment lines have a hash character in the first column followed by arbitrary text. The second line enables the flag **Server_windows**, and the third line gives the variable **Helios_directory** the value **/usr/helios**. The exact meanings of most of the various flags and variables are given below. A summary is given in Appendix A.

The understood format

The I/O server reads the configuration file according to the following rules:

1. Each option must be on a single line.
2. All white space on the line is ignored.
3. The names of flags and variables are not case sensitive, so all the following lines are equivalent.

   ```
   root_processor = /tom
   RoOt_ProcessoR               =/tom
   root_PROCESSOR= /tom
   ```

4. The values of variables may be case sensitive, depending on the host computer's operating system.

   For example, consider the following:

   ```
   helios_directory = /usr/helios
   helios_directory = /usr/Helios
   ```

These lines would be typical of a configuration file for a Unix version of the I/O server. Since the Unix filing system is case sensitive, the two values are very different. On the other hand, consider the equivalent under MS-DOS.

```
helios_directory = c:\helios
helios_directory = c:\Helios
```

Since the MS-DOS filing system is not case sensitive, the two lines are equivalent.

5. Numerical variables may be given in decimal, hexadecimal or octal. The following three lines are equivalent.

```
transputer_memory = 1048576
transputer_memory = 0x10000
transputer_memory = 04000000
```

Some of the variables in the configuration file refer to system files that tend to be relative to the Helios directory. For example, the **system_image** option gives the file containing the Nucleus to be booted into the root processor. Usually this is the file **lib/nucleus** inside the Helios directory. The tilde character (~) can be used to give a filename that is relative to the Helios directory.

```
helios_directory = /usr/helios
system_image     = ~/lib/nucleus
```

or under MS-DOS,

```
helios_directory = c:\helios
system_image     = ~\lib\nucleus
```

Use of the tilde character is not compulsory, so the last line could also read

```
system_image     = c:\helios\lib\nucleus
```

Machine details

The configuration file must have two entries: **host** and **box**. The first identifies the particular host computer or I/O processor, for example:

```
host = PC
host = Sun4
```

Since the I/O server was compiled for a particular machine, the entry is redundant in most, but not all versions. It must always be present, to provide a partial guarantee that the configuration file is sensible.

The **box** option identifies the particular board or box plugged into the I/O processor. Some versions of the I/O server only support a single board. For example, the Sun386 version of the I/O server only supports a B008 or compatible board accessed through the KPar device driver. Other versions of the I/O server can support various different boards in one binary file. For example, the PC server can support the B004 board and compatibles, the B008 board with DMA, the Meiko MK026 interface, and the Cesius board. Typical **host.con** entries would look like this.

```
box = B008
box = MCP10000
```

The Helios directory

In nearly all installations the I/O server must support a **/helios** file server. This has the subdirectories **bin**, **lib**, **etc**, **include**, and so on. This is very useful. Consider the C compiler, which must locate the standard header files such as **stdio.h** A search path for these header files can be provided on the command line, but the compiler can default to **/helios/include**. Every Helios installation is guaranteed to have this directory somewhere in the network, irrespective of whether the host computer is a PC, a Sun, or an alternative machine, so **/helios/include** is a sensible default.

The value of the **helios_directory** should be a directory name for the local machine. For example, a typical entry in a Unix version would be:

```
helios_directory = /usr/helios
```

whereas under MS-DOS the entry would be

```
helios_directory = c:\helios
```

Certain entries in the configuration file may refer to files relative to the helios directory and the tilde character being used to indicate this. In such a case, the correct directory separator character must be used. The following two lines show Unix and MS-DOS versions.

```
system_image = ~/lib/nucleus
system_image = ~\lib\nucleus
```

The message limit

Clients and servers interact by passing messages. Although this is transparent to the normal programmer who can use conventional library calls such as **open()** and **fopen()**, the I/O server must accept and reply to messages. Helios defines a particular message format: a message header consisting of 16 bytes; a control vector of up to 255 words; and a data vector of up to 65535 bytes. Most messages are quite small, no more than eight words in the control vector and 512 bytes in the data vector. However, handling large numbers of small messages is much less efficient than handling a few large messages. Hence for file I/O and other activities which require high bandwidth, messages should be as large as possible.

In the I/O server this causes a problem. Consider the PC version, which has to run on a 640K host machine. Of this 640K typically 150K will be used by MS-DOS, extra device drivers, local area network support, ‘terminate and stay resident’ programs, etc. The I/O server itself is about 140K of binary code, leaving at most 350K for data. Usually at least 100K of this is used for the I/O server’s static data and stacks, leaving 250K for message buffers and handling actual requests from the Helios network. Given these limits it does not make sense to have more than one 64K message buffer in the system, leaving 180K for the I/O server’s internal workings. This suffices for most operations. The I/O server has been written carefully to ensure that only one message buffer is needed. However, if the user installs more device drivers than usual (perhaps 300K instead of 150K), the I/O server becomes short of memory. The code size, static data requirements, and so on, are fixed. Hence the only easy way to gain extra memory is to reduce the message buffer size from its maximum of 64K to a smaller value, such as 5K. This will slow down file I/O because that must take place in smaller segments,

but it should give the user a working system. Of course if the user installs 400K of device drivers then the I/O server cannot run at all. The size of the message buffer is controlled by an option in the configuration file.

```
Message_limit = 60000
```

The lower limit is set to 1000, the default value is 2000, and the upper limit on most machines is 65535. The PC version has a different upper limit of 64000, to cope with the segmented architecture of the 8086 chip.

Multiple windows

To create a user session it is necessary to have a window server. This window server provides multiple text windows rather than a window, icon, mouse and pointer interface. In X window terminology, a Helios window server is a terminal emulator, not to be confused with the X concept of a Window Manager. A Helios window server always provides multiple windows, one way or another.

There are several different types of window server under Helios. One implementation runs under the X window system, where every window is a separate entity on the screen. Another version can be provided by the I/O server, where only one window is visible at any one time but the user can switch between windows using a hot key mechanism. These windows are known as **server windows**, because the first implementation was part of the I/O server. A similar window server is used for remote logins over an ethernet.

On most machines the I/O processor is equipped with its own screen and keyboard. If the processor network does not have any other displays, particularly high resolution graphics displays, then this screen must be used as the main output and hence the window server should be run inside the I/O processor, as part of the I/O server. If there is a high resolution display in the network, it makes sense to run a window server on this rather than in the I/O processor. This option is controlled by a **host.con** flag.

```
Server_windows
```

If this flag is given in the configuration file or on the command line then the I/O server will provide a window server. Otherwise it will provide only a **/console** device, which is essentially a simple window. The main purpose of this device is to ring the bell: the high resolution display in the Helios network is unlikely to be equipped with any sound facilities, so programs like the X server can write bell characters to the **/console** device.

Exactly how the I/O server implements multiple windows depends on the implementation. The standard PC version only supports server windows of 25 by 80 characters. The Sun version is more flexible. When running SunView, multiple SunView windows will be used. Otherwise the I/O server assumes that the user is logged in through a dumb terminal or over a network, and will use server windows. As far as Helios is concerned all **/window** servers behave in the same way, and it does not matter how the windows actually appear to the user.

When server windows are used, the I/O server creates its own window, called **Helios Server**, which is not accessible to applications running under Helios. This window

is used to display the I/O server's copyright messages and for most debugging messages. By default whenever data is written to this window it is popped to the front, on the assumption that the user will want to see error and debugging messages as soon as possible. The user then has to use the hot key mechanism to get back to the window in which he or she was working. There are times when this is a nuisance, so the configuration file has a flag to disable it.

```
Server_windows_nopop
```

In addition one of the debugging facilities can be used to enable or disable the 'nopop' mode, as described below.

Processor names

In a Helios network every processor needs its own name. Usually the I/O processor is given the name **/IO**, and the root processor is given the name **/00**. The names are used by all the processors in maintaining a logical map of the system in an internal name table. A processor is given its name when it is booted up and this name cannot be changed without rebooting the processor and cleaning out name tables in every other processor, which is a difficult operation. The I/O server is usually responsible for booting up the root processor and maintaining a name for the I/O processor, and will default to the names **/00** and **/IO**. If desired, the user can override these defaults by modifying the configuration file.

```
root_processor = /tom
io_processor   = /harry_pc
```

When booting up a network of processors the user has to provide a network resource map giving details of all processors including names. It is important that the names used in the resource map for the root and I/O processors match the names in the **host.con** file.

The error logger

The I/O server provides an error logging server, **/logger**. When Helios starts up this device is used as the initial destination for error messages and progress reports. The I/O server can do various different things with data written to the logger device. By default the data is written into the I/O server's own window, which may pop to the front as described above. Alternatively the data may be written into a file, or it may be written both to the window and to a file. One of the debugging facilities described below can be used to change the destination of the logger device. In addition there are two useful configuration file options.

```
logging_destination = file
logfile = /users/root/helios/logbook
```

The first line indicates the default logging destination, and the value should be one of the keywords **screen**, **file**, or **both**. The default is **screen**. The second line names the file to be used when sending data to a file. By default the I/O server will use the file **logfile** in the current directory.

A typical use of the error logger is as follows.

1. Switch the logging destination to file only, or to both file and screen.
2. Perform some operation that is being debugged, which sends data to the error logger directly or indirectly. Possible ways include using the **IOdebug()** library routine or enabling some debugging option in the I/O server.
3. Switch the logging destination back to screen only.
4. Examine the log at leisure, for example by using the command **cat /logger**
5. Clean out the log using the command **rm /logger**, and repeat until the bug has been found.

Alternatively when the I/O server exits, any data written to the **/logger** device will end up in the log file, and can be examined using whatever tools are available in the host environment. The log file will be cleared when the I/O server is run again.

Disabling servers

Most of the time users will want to have all of the servers that the I/O server can provide. In some cases the I/O server already provides options for disabling servers or choosing between servers. For example, the **Server_windows** option gives users the choice between having a **/window** server and a **/console** server. The I/O server contains a more general mechanism to suppress any server: prefix the server name with the string **no_** and add the resulting flag to the configuration file.

```
no_clock
no_helios
```

There are various reasons for suppressing servers in this way. One is to save memory, eliminating a server that is unused reduces the I/O server's initial memory requirements. Another is when initialising a server such as the **/rs232** device causes a crash, because of hardware incompatibility. This tends to happen on PC systems. Some servers should not be suppressed on the I/O processor unless there is an alternative elsewhere in the network (particularly the **/helios** server).

8.4.4 Root Transputer bootstrap

Booting up the root Transputer in a network is a complex process and requires the following stages:

1. Reset the root Transputer, and possibly the rest of the network if the hardware is set up that way.
2. Check that there appears to be a Transputer at the other end. This involves poking one location in the Transputer's memory, then peeking it to check that the value is correct.
3. Perform the initial bootstrap, sending in the file `nboot.i`. This is a very small program, less than 256 bytes because of limits imposed by the Transputer bootstrap mechanism. It does some hardware initialisation such as clearing the 'halt on error' flag, and then accepts further requests.

4. Send in a request for accepting the **system image** or **Nucleus**, followed by the Nucleus itself. This Nucleus is the basic part of Helios, containing the Kernel, System library, Server library, Utility library, Processor Manager, and Loader.

5. The Kernel starts up and needs some additional configuration information such as the current time and the name of the processor. This information is known as the configuration vector and is generated and sent by the I/O server.

6. There is now a short delay while the rest of the Nucleus initialises itself. As part of this initialisation the root processor will send a byte 0xF0 followed by another eleven bytes, and the I/O server has to send back a twelve byte package.

7. Helios has now booted into the root processor and will start sending requests to the I/O server and to individual servers. The I/O server becomes passive, accepting requests and replying to them, but not generating its own requests.

There are various configuration file options that can be used to define elements of this process.

```
bootfile         = ~/lib/nboot.i
system_image     = ~/lib/nucleus
processor_memory = 0x180000
bootlink         = 1
```

The first option defines the bootstrap utility to be used, the little program (less than 256 bytes) that performs the hardware initialisation and accepts the Nucleus. Usually this is held in the file **/helios/lib/nboot.i**. The tilde character can be used to indicate that the file is relative to the helios directory. In an MS-DOS environment the backslash character should be used instead of the forward slash.

```
bootfile         = ~\lib\nboot.i
```

This option is of interest only to very advanced system programmers. The next option defines the system image or Nucleus to be sent into the root processor. Usually this is the file **/helios/lib/nucleus**, but there are a number of different nuclei serving different purposes. For example, there is a Nucleus which contains all of the Helios file server as well, so that Helios can immediately access a hard disc attached to the root processor.

In most cases the Helios Kernel has no difficulty determining how much memory there is on a processor. However there are some processors where there is strange hardware at the end of normal memory, such as video memory, reset hardware, or special I/O hardware. In these cases the Kernel may become confused and attempt to use this extra hardware as normal memory, generally with unfortunate results. Hence the configuration file variable **processor_memory** allows the user to specify the amount of memory on the root processor. This size is sent with the configuration vector to the Kernel, and stops the Kernel from exploring memory to find out how much there is. No attempt is made to validate the variable, so if the **host.con** file specifies more memory than is actually present the system will crash.

On the majority of Transputer boards the link adapter connecting the processor network to the I/O processor is attached to link zero of the root processor. There are

exceptions, for example the Transtech [7] MCP1000 board uses link one instead. The configuration vector sent by the the I/O server to the Kernel indicates the initial state of the various links on the root processor. To build this information correctly the I/O server must know which root processor link connects to the I/O processor, and the **bootlink** variable allows the user to specify this.

Note: if the **bootlink** variable is not set correctly and the I/O processor is not connected to link 0 of the root processor, Helios is unlikely to boot up.

Booting other root processors

Helios is a general purpose operating system, not specific to the Transputer. The I/O server can be used to boot up a number of processors other than Transputers, although Transputers are the default. The first **host.con** option allows users to specify a processor type other than a Transputer.

```
target_processor = ARM
```

Recognised processor types are T414, T800, T425, T400, ARM, i860, 68000 and 320C40. These define the default bootstrap options. In addition it is possible to suppress various parts of the bootstrap process described above by setting the following flags in the **host.con** file.

no_reset_target Disables Transputer reset.

no_check_processor Disables the check that the root processor is a Transputer.

no_bootstrap Disables initial bootstrap with code file **nboot.i** and hardware initialisation.

no_image Disables the download of the system Nucleus.

no_config Disables sending configuration information.

no_sync Disables the synchronisation sequence.

If the flags listed above are enabled, they will suppress the following options (in the order of the list above): resetting the processor; poking and peeking the processor to check that it exists; sending the bootstrap utility **nboot.i**; sending the system image; sending the configuration vector; and waiting for the byte 0xF0, the initial synchronisation. These options suffice for most systems where the root processor and I/O processor communicate through a link adapter or similar mechanism, but if your hardware is very different, it will require a completely different bootstrap mechanism.

Booting without a root processor

In some multi-user networks the main processor network is booted up from a separate system console. Individual users have their own I/O processors but these are equipped only with a link adapter, not with a separate root processor. In this case the I/O server should connect into the network, rather than perform any bootstrap. This can be achieved by the following **host.con** flag.

[7] Trademark of Transtech Devices Limited

```
enable_link
```

This option is similar to the command line **-e** option. However **-e** only applies when the I/O server first starts up, and a subsequent I/O server restart will cause a normal bootstrap. The **enable_link** option is permanent, and the I/O server will never perform a bootstrap.

The **enable_link** option allows the I/O server to be connected to a processor which is already booted, although the link must be reset before it can be used with certain boards. Unfortunately there is no standard procedure on a PC to simply reset the link. The only way to reset the link adapter is to use the **reset_target** option, which resets the whole board, so the user must ensure that this does not also reset the processor.

8.4.5 Special actions

Depending on the implementation the I/O server responds to either three or four special actions.

1. **Reboot** causes the I/O server to shut down all existing streams that have been opened by Helios clients, closing files. It then reboots the root processor. A reboot is not permitted if the **enable_link** option is defined in the configuration file because that option specifies that the I/O server should never attempt to boot up a processor. Rebooting will not cause the configuration file to be re-read. If that is desired the I/O server should be made to exit and then restarted.

2. **Exit** causes the I/O server to shut down all existing streams as before, and then terminate.

3. **Status** should cause the I/O server to display a message

   ```
   Server alive and well. Logging goes to screen.
   ```

   If no message is displayed within about five seconds then the I/O server has hung up for some reason, probably when reading from or writing to the link and because a program in the processor network has overwritten some important piece of memory. Usually it will be necessary to abort the I/O server program or even reboot the I/O processor.

4. **Debugger** causes the I/O server to shut down and then enter the built-in debugger, as described in the next subsection. It is not available in all versions of the I/O server.

Different versions of the I/O server have different ways of triggering these actions. In the PC Server combinations of the control and shift keys with a function key are used. For example, to cause a reboot the user can hold down the control and shift keys and then press function key F10. This key combination is most unlikely to happen by chance, but can be achieved fairly easily. When running on a Sun, using either SunView or the X window system, the I/O server will display a panel with debug buttons for all these activities. When using a dumb terminal user defined key sequences can be used to trigger the special actions. Two of the special actions, exit and reboot, can

be triggered from the Transputer network. There are commands **stopio** and **rebootio** to achieve this. If server windows are in use then there will be another two or three special action keys or key sequences.

1. Switch to next window.

2. Switch to previous window.

3. Redraw current window (only supported if it is likely that other programs may send data to the screen, corrupting the display).

8.4.6 Debugging facilities

In simple Helios networks most if not all input/output goes through the I/O server. Potentially this allows for some very powerful tracing. Using a fairly small amount of code in the I/O server it is possible to generate a debug message whenever a file is read, whenever a timeout occurs, and so on. The debugging message is sent to the error logging device described above, so it can be sent to screen, to a file, or to both.

When the I/O processor has a graphical interface, such as SunView, the main panel will have a pop-up menu allowing users to enable or disable debugging options. Otherwise key combinations or key sequences are used. For example, in the PC version of the I/O server, holding down the control and shift keys and then pressing the **m** key toggles the message debugging option. Associated with every debugging option is a single letter. This letter may be given on the command line to enable various debugging options as the I/O server starts up. For example,

```
server -mn
```

starts up the I/O server with message and name debugging enabled. Most debugging options are vaguely ‘mnemonic’. For example, message debugging is associated with the letter **m**.

Some of the debugging facilities are aimed at system programmers, and in particular the Helios developers, rather than ordinary users. These can still be useful to ordinary users, but it may require more effort to understand the output produced. Where relevant the descriptions below refer to system header files such as **/helios/include/codes.h**, which should contain all the required information. The following debugging options are defined at present. There are unlikely to be any additions to the list since there are no unused letters left in the alphabet.

A All. On the command line this enables all debugging. Since there is a lot of debugging (a typical startup and login sequence generates 75K or more), this option should only be used when the system is known not to work. Toggling the option at run time can have two different effects: if no debugging options are currently enabled then all of them are enabled; otherwise all debugging options are disabled.

B Boot. This gives details of the bootstrap sequence. The output that should be produced is something like:

```
Loading system image c:\helios\lib\nucleus.dbg
Resetting link and processor.
Sending bootstrap, size is 212
Sending system image (77028 bytes)...
Sending 4096 bytes at 7bd90024
Sending 4096 bytes at 8bd90024
   ...
Sending 3300 bytes at 8bd96d70
Sending configuration ...
Waiting for sync byte from Kernel
Received a byte F0
Booted.
```

This output matches the bootstrap sequence described earlier, although exact details may differ depending on factors such as the size of the Nucleus.

C Communication line debugging. When transmitting data down serial lines this debugging option may, depending on the version, generate progress reports. Typical progress reports would look like this:

```
RS232 write : written 60 of 1024 bytes
RS232 read : read 12 of 80 bytes
```

Different versions of the I/O server may generalise this debugging option to other devices which behave in similar ways, such as Centronics and similar printer ports or Midi ports.

D Delete. This option gives debugging information whenever a file or directory is being deleted. It is particularly useful when performing recursive deletes that seem to be taking rather longer than expected, because it allows the user to verify quickly that the files being deleted are the ones that are supposed to be deleted. Typical messages produced by this debugging option are:

```
Delete request for unknown object /users/root/helios/tmp/junk
Deleting file /users/root/helios/tmp/junk
Deleting directory /users/root/helios/tmp
```

Note that the messages give the local file name, not the Helios file name.

E Errors. Clients interact with the servers inside the I/O server by sending requests, and the servers reply with messages indicating either success or an error . The error is encoded into a 32-bit integer, as follows:

```
Bit 31                                          Bit 0
       1 CC SSSSS GGGGGGGG EEEEEEEEEEEEEEE
```

All error codes have their top bit set, to indicate that they are errors. The next two bits indicate the severity of the error, from recoverable to fatal. Then there are five bits giving the subsystem code, the part of Helios that generated the error. For the I/O server this is always the subsystem **SS_IOProc**. The next eight bits give details of the error group or type of error, such as insufficient memory to

perform the operation or object in use. The final 16 bits give some information on the type of object affected, such as a file or a program. Enabling the **errors** debugging option causes the I/O server to give a message whenever some server sends back an error code. A typical message would be:

```
Error : fn ca06800c
```

This error code means: error from I/O server, unknown file. In other words Helios tried to access a file that did not exist. The Helios shell has a built-in fault command that interprets such messages, so typing in:

```
fault ca06800c
```

would give a description of the fault. Programs can use the Helios Fault library to turn error codes into strings. The header file

```
/helios/include/codes.h
```

defines the current error codes and related information.

The **error** debugging option is a restricted form of the message option described below. It only gives information about the error code, not about the request which caused it.

F File I/O. This debugging option generates messages for miscellaneous file I/O activities (such as renaming files) which are not sufficiently important to require a specific debugging option. Typical debugging messages include:

```
Creating/truncating file c:\junkfile
Creating directory c:\tmp
ObjectInfo request for c:\cshrc
Rename request received
Renaming c:\junk1 to c:\junk2
SetDate request for c:\work\lock.c
ServerInfo request received for c:\
```

Most of these messages are fairly obvious. The **ObjectInfo** request is used for getting specific information about an object, such as its size, and is used by programs like **ls**. The **SetDate** request is used by the **touch** command to change the time stamp of a file, typically in association with make utilities. **ServerInfo** is used by the **df** command to determine how big a disc is and how much of its available space has been used.

G Graphics. The output produced by this option depends entirely on the implementation of the I/O server and the type of graphics supported. Usually no output is produced at all.

H Hard disc. This option is used in conjunction with the ‘rawdisk’ device. A ‘rawdisk’ behaves like a single file of perhaps twenty megabytes, and applications can write chunks of 5K or so at 512 byte boundaries. In other words, a rawdisk device allows the Helios file server to read and write sectors of the disc, without going through the host filing system. Typical messages produced are:

```
Rawdisk : open
Rawdisk : close
Rawdisk : seek to 10200
Rawdisk : GetSize request, size is 1400000
R : not on sector boundary
R : not whole number of sectors
R : past end of disk
R : @ 10200, 4 sectors
R : hardware error
W : @ 10200, 6 sectors
```

The size of the hard disc and the offsets are given in hexadecimal numbers.

I Initialisation. This debugging option gives information as the I/O server initialises all its servers. Typical output for a PC version of the I/O server would be:

```
Initialising device IO.
Initialising device helios.
Initialising device logger.
Initialising device window.
Initialising device clock.
Initialising device pc.
Initialising device rs232.
Initialising device centronics.
Initialising device printers.
Initialising device a.
Initialising device b.
Initialising device c.
Server successfully initialised.
```

If, for example, the **no_logger** flag is set in the configuration file, the message 'Initialising device logger' will still appear, but the device will terminate during its initialisation sequence. The information produced by this debugging option can be very useful when the system fails to boot, and the host computer has crashed completely and needs to be rebooted. For example, if the I/O server gets as far as the message 'Initialising device rs232' but no further, then it is fairly certain that the configuration file entries for the rs232 device are wrong.

J Directory I/O. This debugging option gives information when a client opens and reads directories in the host filing system. Typical messages produced are:

```
Reading directory c:\helios\bin
There are 68 entries in the directory
Directory read for /helios/bin : 440 bytes at 0
Directory read for /helios/bin : 440 bytes at 440
Directory read for /helios/bin : 440 bytes at 880
...
Closing directory /helios/bin
```

Every entry in a directory takes up 44 bytes, using the structure **DirEntry** defined in the header file **/helios/include/syslib.h**. Hence in the example the client is reading ten directory entries at a time, and after sufficient practice the user will recognise this behaviour as typical of the Helios shell.

K Keyboard. This option generates information for all keyboard input, whether reading conventional keys through the **/console** or **/window** devices or obtaining scancode events through the **/keyboard** device. The former generates message like:

```
Read char 0x20 from Shell.1
```

This gives the hex code for the keyboard character, which in this case is a space, and the name of the window. The message merely says that the character is now known to the I/O server, not that it has been passed on to Helios. There may not be a client currently reading data from that window, in which case the I/O server will buffer keyboard data. Alternatively the window may be in a cooked input mode (processing and buffering input), in which case no data can be sent until the user has pressed the return key. The messages produced by the raw keyboard device are rather different:

```
Sending raw keyboard event : 39 down
Sending raw keyboard event : 39 up
```

These messages are generated when the data is actually sent to the appropriate client, not when the I/O server detects that a key has been pressed or released.

L Logger. This option allows users to switch the current logging destination. It is a very useful option, so when the I/O server provides a graphical interface there is usually a special button to activate this option. Triggering this option should generate one of three messages:

```
Switching logging to file only.
Switching logging to file and screen.
Switching logging to screen only.
```

A fairly typical way of using the error logger is to have the default destination as the screen. To investigate a problem the user changes the destination to file only, enables suitable debugging options, reproduces the problem, and switches the destination back to screen only. The log file can now be examined at leisure without leaving Helios, for example by using the command **cat /logger**, or after leaving Helios if desired. Under Helios it is possible to empty the error logger by using the command **rm /logger**.

M Message debugging. In a system like Helios where clients and servers interact by passing messages it is useful to report every single message that passes between the Helios network and the I/O server. However, considerable expertise is required to understand the output. Consider the following simple transaction.

```
Request: fn 30, for 2 from c0010234, csize 5, dsize 17
Reply  : fn 0, port c0010234, csize 6, dsize 20
```

There is an incoming message, function code 30, for message port 2 from the message port c0010234, with a control vector of five words and a data vector of 17 bytes. The I/O server replies with a function code 0, to the reply port of the incoming request, with a control vector of 6 words and a data vector of 20 bytes. This may appear to be meaningless. However, the following information can be obtained:

1. The destination port of the reply corresponds to the source port of the request. This allows requests and replies to be matched up, which is rather useful because the I/O server can have several requests outstanding at any one time.
2. The destination port of the request is 2. This information is useful only when the I/O server first starts up. The I/O server starts allocating message ports at 1, and simply increments this number every time a new message port is required. Every server and every stream needs its own message port. The information obtained from the initialisation debugging (shown in the example on the **I** option) now reveals that device **/IO** must have message port 1, and device **/helios** has message port 2. Hence the request is being sent to the **/helios** server.
3. The request had a function code of 30. Looking this up in the header file **codes.h** would reveal that this function code is **FG_Locate**, so the client is looking for something, probably a file.
4. The header file **gsp.h** defines the message format for the locate request, and this does indeed involve a control vector of 5 words. The size of the data vector depends on the name of the object being accessed.
5. The reply had a function code of 0, which according to **codes.h** is the constant called **Err_Null** indicating success. Hence the object that the client is trying to access really does exist.
6. The header file **gsp.h** defines the format for the reply message to a locate request, and this does involve a control vector of 6 words.

Hence both the request and the reply appear to make sense, and with a little effort the user should have a good idea of what is going on. Hopefully the other debugging options would allow the user to get exactly the same information in a simpler format, but message debugging is available when needed.

N Names. This option causes the I/O server's name handling routines to produce debugging messages. Typical examples would be:

```
Context /IO/helios/lib, name Clib, rest helios/lib
IOname is helios/lib/Clib
```

When Helios clients access an object they tend to do so within some context. For example, when a C program calls **fopen("myfile", "r")** the context is the program's current directory. This context is very important in a protected environment because it is used to validate access to the target object. In the example given the context directory is **/IO/helios/lib**.

The second field, the **name**, indicates what object is being accessed relative to the context. In the example this is the file **Clib** within the context **/IO/helios/lib**. The third field indicates how much of the name has been handled before reaching the I/O server. In this case the string **/IO** had been interpreted already before the message arrived at the I/O server. The final **IOname** value gives the full name of the object within the I/O server. This name is then translated into a machine specific name such as **c:\helios\lib\clib** before the actual file access takes place.

There are times when the context is NULL. For example, when a program attempts to open the file **/helios/include/stdio.h** the context is empty, because that file is absolute rather than relative to the current directory. Similarly the context can refer to the whole object rather than a directory containing the object, in which case the name is NULL. Usually these details are of little interest, and all that matters is the name of the actual object being accessed.

O Open. This option gives messages whenever a file is opened. Typical messages are:

```
Supposed to open /users/root/helios/tmp/junkfile
File not found
Failed to open
```

The first message indicates that the I/O server is about to open a file. Since doing this might conceivably crash or hang the I/O server, the message is given before the attempt is made. The second message indicates that the file did not exist and the message did not involve automatically creating a non-existent file, (the Posix **O_CREAT** bit was not set). The third message indicates that the underlying host filing system refused to open the file, for an unspecified reason.

P Non-mnemonically, the letter **P** is used for **Close** debugging. This option gives a message whenever a file is closed.

```
Supposed to close helios/lib/Clib
```

Q Quit debugging. When the I/O server terminates this may produce some or all of the following messages.

```
Shutting down streams and servers.
Streams and servers shut down.
Restoring devices.
Freeing configuration information.
Tidying debugger.
Tidying bootstrap.
Terminating logger.
Server exiting.
```

This option is not particularly useful, but is supplied for completeness. Note that the error logger is shut down as part of the quitting process, so some of the messages are sent directly to the I/O processor's screen rather than to the error logger and hence cannot be redirected to a logfile.

R Read debugging. This gives information whenever a file is read.

```
Supposed to read 1024 bytes at 4096
in helios/include/stdio.h
Having to seek to 4096 from 2048
Returning 712 bytes
```

The first message gives the amount of data to read, the position within the file to read it from, and the file itself. The second message is unlikely to be seen, but indicates that the application and the client have lost synchronisation. The Helios protocols are designed to be fault tolerant and they will recover automatically from such errors. The final message indicates the amount of data actually read, which may be less than the amount requested when the end of the file has been reached.

S Search debugging gives information about distributed search messages that reach the I/O server. When a client tries to access a particular server, for example **/helios**, and that server has not yet been accessed from that processor, then Helios will generate a distributed search throughout the network for the server. Some of these messages may reach the I/O server, which may have a server of the specified name. Typical messages produced by this debugging option are:

```
Distributed search for /01 : unknown server
Distributed search for /helios : found, server is 2
Distributed search for /01/helios : wrong network address
```

The first and last distributed searches fail as far as the I/O server is concerned, although hopefully some other processor in the Helios network will be able to satisfy the search request. The second distributed search succeeds. The number `2` is the message port for the specified server, which may be matched up with the ports listed with the message debugging option.

T Timeouts. Some devices such as windows and serial lines support timed I/O. A typical request might be to read 80 bytes in at most 20 seconds. Timeout debugging can be used to find out exactly when a timeout occurs.

```
Timeout in stream window/console
```

U **Nopop**. This option is used in conjunction with the **Server windows** windowing system and its hot key switching mechanism. The I/O server has its own window, and by default this window pops up to the front whenever data is written to it. A **host.con** option **Server_windows_nopop** can be used to suppress this popping by default. In addition the current mode can be toggled, enabling or disabling **nopop**, using this debugging option.

V OpenReply. **FG_Locate**, **FG_Create**, and **FG_Open**, which are three of the most important requests in Helios, expect the same format for the reply message. This is the **IOCReply1** structure defined in the header file **gsp.h**. Most of the contents of this reply message can be displayed using the **OpenReply** debugging option.

```
FormOpenReply : name /IO/c/doc/ioserver/chap2.tex
              : type 12, flags 20020000, access c7
```

For the meanings of the various fields, please see the header file.

W Write debugging. This is very similar to Read debugging.

```
Supposed to write 1024 bytes at 4096 in helios/tmp/xx
Having to seek to 4096 from 2048
```

X Resources. This option gives a snapshot of what is happening inside the I/O server. Typical output might be:

```
Open streams are : window/console, window/console
There are 13 devices, 2 open streams, and 0 other coroutines
```

The first line lists all the streams currently open to the I/O server. The next line counts the number of servers, the number of open streams and the number of coroutines. Coroutines are similar to Transputer processes but are uninterruptable. The number of other coroutines is likely to be zero, but may be non-zero when a large write to a window or serial line has been XOFF-ed.

Some implementations of the I/O server perform their own memory management rather than rely on system calls, because the I/O server can do it faster or more reliably. The PC implementation is one. If so then the resource debugging option also gives details of memory allocation inside the I/O server.

```
In the small memory pool there are 3 nodes giving 13480 bytes
In the big memory pool there are 6 nodes giving 162576 bytes
```

The memory allocation code maintains two separate heaps, one for small bits of memory of less than 250 bytes, another for larger chunks. These are known as the small and big pools respectively. The number of nodes is an indication of memory fragmentation inside the I/O server, the more nodes the greater the fragmentation. The final value indicates the amount of memory left inside the pool. If either the small or big pool runs low on memory, requests sent to the I/O server may fail because of insufficient memory.

Y List. In a non-graphical environment such as a PC this is potentially the most useful debugging option. It gives a list of all the other options.

```
Available debugging options are : a = all
b : boot         c : serial    d : delete     e : errors
f : file I/O     g : graphics  h : raw disk   i : init
j : directory    k : keyboard  l : logger     m : messages
n : names        o : open      p : close      q : quit
r : read         s : search    t : timeouts   u : nopop
v : open reply   w : write     x : resources  y : list
z : reconfigure
```

Z Reconfigure. This debugging option causes the I/O server to re-read the **host.con** configuration file. This is not as useful as might appear. Many of the options are tested only when the I/O server first starts up, so reconfiguring has no effect on these. Others are tested only when the I/O server activates all its servers, so a reboot is required for the new configuration to have an effect. Only a few options are checked while the I/O server is up and running. Nevertheless, the option is available when required.

8.4.7 The built-in debugger

Debugging tools and facilities are essential when producing software. The range of tools and facilities is considerable.

1. Source-level debuggers are useful for applications software. These allow the programmer to examine the source as it is being executed, and generally provide powerful facilities such as breakpoints, single stepping, watch statements, and so on. Such a source-level debugger is available under Helios. These debuggers require support in the compiler, in particular a special option to compile for debugging. Hardware support for debugging is often desirable but not essential. The debugger will only work on top of an existing system, which must be reliable. Hence source-level debuggers cannot usually be used for debugging system software.

2. If no source-level debugger is available then the approach usually taken is to embed extra statements in the code. Typically these are **fprintf()** statements, sending text messages to the standard error stream. Using C library routines like **fprintf()** is not always sensible, because the C library itself may be corrupted by the bug and hence the expected text will not appear. To overcome this problem Helios provides a lower-level routine **IOdebug()**, which has a similar syntax to **printf()** but is more likely to make the data visible to the user.

3. Particularly when programming in a language like C, one of the more common causes of problems is using invalid pointers. A pointer may be used before it has been initialised, so that it points to some random location in memory. On some machines, but not on Transputers, invalid pointers can be detected and the machine can give a core dump. This core dump contains full details of the program, but not of the rest of the system.

4. In the absence of suitable hardware, some limited debugging can still take place. In particular, if the processor can be reset then effectively the user can get a snapshot of what the processor was doing at the time. The whole processor has to be examined, not just the one program being debugged. Performing this operation is the purpose of the I/O server's built-in debugger.

The I/O server's built-in debugger is specific to the Transputer, and cannot be used with other processors. To use it sensibly requires considerable understanding of the inner workings of the Transputer and of Helios. Its main use is for system programmers.

Where to look

The main facility provided by the built-in debugger is examining memory. Since the whole processor has to be examined, the user needs to know where to look. This subsection describes the memory layout of the Transputer and the location of some of the main Helios data structures. It assumes that the processor has been booted from a link, a reasonable assumption for most Helios systems.

In a network of processors it is usually possible to examine the root processor only, because this is the only one that can be reset by the I/O server. The debugger does contain some support for examining remote processors if these can be reset.

Reserved locations

Transputer memory begins at location 0x80000000, at least for the 32-bit processors that are likely to run Helios. At the bottom of memory are some reserved locations, used by the Transputer hardware (see the table below).

Location	Purpose
80000000	Link 0 Output
80000004	Link 1 Output
80000008	Link 2 Output
8000000C	Link 3 Output
80000010	Link 0 Input
80000014	Link 1 Input
80000018	Link 2 Input
8000001C	Link 3 Input
80000020	Event line
80000024	Timer (high priority)
80000028	Timer (low priority)
8000002C	Saved workspace pointer
80000030	Saved instruction pointer
80000034	Saved A register
80000038	Saved B register
8000003C	Saved C register
80000040	Saved status register
80000044	Saved E register
80000048	2-d block move
...	support
8000006F	

The first eight words give information about what is happening on the specified link. If the word contains 0x80000000 then the link is idle. Under Helios the output links are usually idle, unless the processor was reset in the middle of a link transfer. For those links which are connected to other processors running Helios, the input link is usually set to some value other than 0x80000000 because Helios has link guardian processes continually monitoring the links for traffic. The actual value in these words is a pointer to a location on the stack of the process doing the link I/O, and examining memory in the vicinity of this location may provide useful information.

The next location is for the event line. The Helios Kernel continually monitors the event line, although very little hardware makes use of it. The Kernel provides routines **SetEvent()** and **RemEvent()** so that applications can make use of the event line.

There are two timer pointers, one for high priority processes and one for low priority ones. These will point to the next process (or rather to a location on the stack of the next process) to be woken up by a timeout.

When a Transputer switches from running a low priority process to a high priority one, details of the low priority process are saved at the bottom of memory. Typically this would happen when a timer goes off or when some data arrives on a link, reactivating a high priority Kernel process. If the user's program is the only application running on that processor then there is a good chance that the saved low priority process describes the user's application.

On-chip memory

Between location 0x80000000 and location 0x80001000 is the Transputer's 'on-chip' static memory (the T414 only has 2K of this memory, not 4K). When a processor is booted up the bootstrap program **/helios/lib/nboot.i** is placed at location 0x80000070. This program may be examined if desired.

nboot.i reads in a Nucleus starting at off-chip memory, location 0x80001000. This consists of a single word, the size of the Nucleus, followed by a number of self relative pointers to the components of the Nucleus: Kernel, System library, Server library, Utility library, **nboot.i**, Processor Manager, and Loader. The Nucleus typically takes up between 84 and 100K of memory. During its initialisation phase the Kernel reads a configuration vector from the link, into location 0x80000200. The structure of this configuration vector is defined in the header file

```
/helios/include/config.h.
```

By default Helios does not make use of the 'on-chip' memory. There are routines called **AllocFast()** and **Accelerate()** to allow applications to make use of this memory, if desired. If this happens then the memory containing **nboot.i** and the configuration vector may be overwritten, but Helios no longer needs these.

The root data structure

Every Helios processor maintains a root data structure which can be accessed by all the components of the I/O server. This is the **RootStruct** data structure, defined in the header file

```
/helios/include/root.h
```

The system normally places this structure immediately after the Nucleus itself. This means that the data structure can be found by indirecting through the self relative pointer at location 0x80001000, which holds the size of the Nucleus.

To understand the root data structure it is necessary to examine several other header files as well. For example, the root structure contains various **Pools**, which are defined in the header file **memory.h**. The **LinkInfo** structure is defined in **link.h**. With a little effort, most of the processor's workings can be determined in this way.

The bulk of memory

After the end of the root data structure comes the main memory, which is used by Helios for such tasks as loading programs and allocating stacks and heaps. Memory allocation starts at the top of memory going downwards. While Helios is still up and running the **mem** command can be used to determine which program owns which bit of memory, and to some extent what it is used for.

The trace vector

At the very end of memory the Kernel maintains a private data area known as the trace vector. The Kernel contains a routine **_Trace()** which can be used to put some data into this trace vector, three words, to be exact. The data in the trace vector can then be examined while the processor is being debugged. Typically the first word is a magic number indicating the source of the trace message, and the other two words give actual debugging information. This approach is the lowest level of debugging, and generally it is used only when debugging the Nucleus.

Available commands

This subsection describes the various commands available inside the debugger, and the next subsection illustrates how these commands are used in a typical session. The debugger can be activated in various ways. First the I/O server can be run with the **-d** option, which means that it will enter debugging mode immediately rather than boot up Helios. Second, once Helios is up and running there will be some machine specific way of switching to the debugger. On a PC this can be done by pressing the control key, then the shift key, and finally the F7 key. On a Sun it can be done by clicking on the Debugger button.

The main facility provided by the I/O server is examining memory. Hexadecimal is used for this, because it is usually the most appropriate notation. Occasionally the user will have to type in a memory address, and memory starts at 0x80000000 so in theory the user would have to type in eight digits every time. In practice the debugger maintains a base value, and all addresses are relative to this base. For example, to examine location 0x80001000 the user simply types `1000`, and the debugger automatically adds the memory base. To move to an absolute address the hash symbol can be used. For example, the user could also type `#80001000` to produce the same effect.

Memory may be displayed either as bytes or as words. Words are usually used when examining pointers and integers, but bytes are more useful when examining text strings and so on. There are commands to switch between the two modes.

Most of the commands simply move around in memory. These can be activated by pressing a single key, for example pressing the '.'(full stop) key will move to the next frame of memory. Other commands involve typing in a whole string and pressing return.

Examining memory

By default the debugger will display one frame of memory at a time, either as bytes or as words. Initially a frame is sixteen bytes, but this can be changed with the **:**

command. The following single key commands are available.

RETURN	Redisplay the current frame
.	Move to the next frame
,	Move to the previous frame
;	Disassemble the current frame
'	Move to the next frame and disassemble it
>	Move to the next word
<	Move to the previous word
[	Indirect through an absolute pointer
]	Pop a previous address off the stack
{	Indirect through a self relative pointer

Most of the commands should be straightforward. Disassembling may produce strange output if the current frame falls on an unfortunate boundary in the code, but this is unlikely. The debugger understands two types of pointers. The normal type is an 'absolute' pointer, where a variable holds a pointer to another variable. For example, the root data structure contains the field:

```
word     *Tracevec;
```

This means that there is a variable inside the data structure containing another value such as
0x801ff000, a fixed address in the machine. Self relative pointers contain a value which is an offset from the address of the self relative pointer. They are much less common than absolute pointers. The most important example is at the start of the Nucleus, location 0x80001000. This value contains the size of the Nucleus (60K, for example) and the root data structure immediately follows the Nucleus. Hence the root is at location 0x80001000 + 60K, and can be reached by a self relative indirection when the current frame is at 0x80001000.

The debugger maintains a stack of all indirections, allowing the user to go back to a previous pointer. For example, the frame is currently at the root data structure and the user types [to indirect to the trace vector. The old address is pushed onto the stack, and the frame moves to the trace vector. To get back to the root structure the user can simply type], without having the remember the actual address of this structure. There are a number of related commands that take a single number as argument.

+ **n**	Move forward **n** bytes
- **n**	Move back **n** bytes
: **n**	Set the frame size to **n** bytes
= **n**	Poke the value **n**

For example, typing in : 32 at the debugger prompt will change the current frame size from 16 bytes to 32.

Note: there is one other single character command: pressing the escape key is equivalent to issuing the command **explore**. The next thing to be done is to type in fully the more powerful commands.

analyse

This command will analyse the root processor. This is a special form of reset. When the analyse signal is asserted the root processor will continue operation for a number of cycles, and then halt when it reaches a suitable descheduling point. Some of the processor state information can then be extracted. The **analyse** command is rarely used directly, because it is invoked automatically during the **explore** command.

base

This command may be used to change the base address from 0x80000000 (its default value). It is useful occasionally when debugging a piece of hardware which has been memory-mapped. For example, let us suppose that there is some graphics hardware at the location 0xC0000000 which the user is trying to access. One way, though not necessarily the best way, is to run the debugger and change the base address. The hardware can then be examined and poked. The following commands might be used for this.

```
base #C0000000 ; change the base address
= 0 ; poke this address with value 0
+ 12 ; move 12 bytes forwards
= 1 ; poke this address with value 1
```

Please note that the debugger only supports poking whole integers at integer aligned addresses, because that is all that is supported by the Transputer hardware.

bytes

The debugger can work in two modes when displaying memory. In words mode it displays primarily 32-bit words of information. In bytes mode it displays primarily single bytes. This command can be used to switch from words mode back to bytes mode.

clear

This command sends a little program into the root processor to clear memory. In particular, all memory from the base location to the current position is cleared. This command may be useful if memory has become too cluttered with old information and the user wants to clear it all before rebooting Helios. A typical way of using the command is:

```
1ff000 ; move to near the end of memory
clear ; and clear it all
```

cmp

This command compares the Nucleus currently held in the root processor's memory with the Nucleus as held on disc. If any differences are found then a program has caused some serious memory corruption and has overwritten part of the Nucleus code. Typical output that might be produced is:

```
80001500: 00000000 != 60bf7332
```

This means that the value at location 0x80001500 should be 0x60bf7332, but is actually 0. All this command can tell you is that memory has been corrupted, not which bit of which program did so.

dump

When a Transputer has been analysed a little worm program can be sent into it to find out what it was up to at the time. This is the purpose of the **dump** command. The information is stored locally, and can be displayed with the **info** command. **dump** is rarely used explicitly, since the **explore** command does it automatically.

explore

The **explore** command performs three separate actions. First, it analyses the root processor as per the **analyse** command. Second it sends a little program into the processor to explore its state, as per the **dump** command. Finally it gives details of the results as per the **info** command. This command is used mainly when Helios has already run and the user has switched to the debugger. Since speed is important (the faster the processor can be explored, the more likely it is that the information is useful) the escape key can be used as a short-hand for typing the whole command. The output produced typically looks something like this:

```
Iptr          : 80002838
Wptr          : 800fe028
BootLink      : 80000010
Output links  : 800e8cec 80000000 80000000 80000000
Input  links  : 800fdd40 80000000 800fd110 800fcd00
Event channel: 800fc11c
Timer Queues  : hi 800fc510 lo 800fc91c
Save Area     : W 80000001 I 800042f6 A 800c20c4 B 8000162a
                C 800c19c4 S 00000000 E fff14bc0
Hi Pri Queue  : head 80000000 tail 800fe324
Lo Pri Queue  : head 80000000 tail 800fc91c
```

The first three words give status information about what the processor was doing just before it was analysed. In this case it was executing code somewhere inside the Nucleus, so it will be difficult to find out anything useful from that. The remaining information was described in the subsection on reserved locations earlier. Important things to note are that the root processor was attempting to send data to the I/O processor at the time, and there were Helios processors at the end of links 0, 2 and 3. The timer and queue information may be investigated further.

go

The **go** command boots up Helios into the root processor but does not respond to messages which that processor is attempting to send. This may be useful when investigating a Nucleus that does not boot up correctly. The command may be aborted by hitting the escape key, which causes another **explore**.

info

The **info** command displays information that was produced by the **dump** or **explore** command earlier. It is invoked automatically at the end of **explore**, but the information may be redisplayed by issuing this command. The output produced is the same as for **explore**.

load

This command loads the system image from the disc into memory. This happens automatically when the **cmp** or **go** commands are used.

quit

The **quit** command is used to exit the I/O server and return to the host operating system.

reset

The **reset** command is used to reset the root processor. This is necessary if the processor's memory is to be examined. Please note that a processor does not necessarily stay reset. In particular, if a neighbouring Transputer sends data down the link while the processor is reset then this data will be interpreted as a bootstrap. The user will notice that the I/O server takes a long time to respond to key presses while it still attempts to access the processor, and the processor should be reset again using this command.

server

This command may be used to leave debugging mode, reboot Helios, and enter normal I/O server mode again.

settrace

The Helios Kernel maintains a small area of memory near the top of memory, known as the trace vector. This area is usually accessed through the root data structure, so the debugger has no problem finding it. However, if the root data structure has been corrupted then the normal **trace** command will fail. In that case it is necessary to explicitly set the address of the trace vector:

```
settrace ff000
```

The trace vector takes up four kilobytes from the end of memory, so for a one megabyte processor it would start at 0x800ff000, for a two megabyte processor it would be at 0x801ff000, and so on.

trace

If there is any information in the trace vector then this command will display it. If some application has called the Kernel **_Trace()** routine with arguments 0x55, 0x66, and 0x77, then the output will be something like:

```
Regs: T = 00012345 W = 80fee820 I = 80002324
      A = 00000055 B = 00000066 C = 00000077
```

The command gives a time stamp, details of the stack pointer when_**Trace()** was called as well as the piece of code that called it, and the three arguments to the routine. Note that the trace vector is only 4K, so if there are too many trace statements then the vector will wrap around and somewhere in the data displayed there may be a discontinuity. The timestamps can be used to work out the order in which the data was added to the trace vector.

words

This routine switches the debugger from bytes mode to words mode, as described in the subsection on the **bytes** command.

xp

This routine provides limited support for debugging remote processors. It works by running a small program in the root processor which forwards all debugging messages to the specified link. For example, suppose the network consists of three processors `00`, `01`, and `02`, with link 2 of processor `00` going into link 1 of processor `01`, and with link 2 of processor `01` going into link 1 of processor `02`, giving the usual processor pipeline. The following commands might be used to debug processor `02`.

```
xp 2 ; now debug 01, at the end of link 2 of 00
xp 2 ; now debug 02, at the end of link 2 of 01
```

This technique will work only if all the processors are reset, because it is not possible to use the debugger unless the target processor is reset. Also, it works by running little programs in all the intermediate processors. Hence if the user types the **reset** command then the root processor will be reset, it will stop running the little program, and the user is back to debugging the root processor.

An example session

This subsection describes a short debugging session. Assume that Helios was booted up normally, suspended at a shell prompt waiting for keyboard input, and the user switched to debugger mode. On a PC this could be done by pressing the control key, then the shift key, and finally the F7 key. On a Sun it can be done by clicking on the debugger button. The display might look something like this:

```
Helios PC Transputer Debugger V3.83 21.3.90
Copyright (C) Perihelion Software Ltd. 1987-1990
All rights reserved.
```

The debugger is now waiting for some keyboard input, and the most useful thing to do is to explore what the processor was up to. This can be done by typing the **explore** command or by pressing the escape key. For example:

```
Explore
Saving low memory...
Sending bootstrap...
Restoring low memory...
Iptr         : 800028bb
```

```
Wptr          : 800fe028
BootLink      : 80000010
Output links  : 80000000 80000000 80000000 80000000
Input  links  : 800fdd40 80000000 800fd110 800fcd00
Event channel: 800fc11c Timer Queues : hi 800fc510 lo 800fc91c
Save Area     : W 80000001 I 800042f6 A 800c20c4 B 8000162a
                C 800c19c4 S 00000000 E fff14bc0
Hi Pri Queue : head 80000000 tail 800fe324
Lo Pri Queue : head 80000000 tail 800fc91c
```

Exploring involves running a small program inside the processor, and it is desirable that this does not corrupt the current state. Hence a small amount of memory has to be saved before running the program and restored afterwards. The information shown is not particularly useful. The processor was currently executing a process in the Nucleus. Since all applications were idle this is to be expected, as there are no other active processes. There was no link output traffic, but there were link guardian processes on three of the links implying that there were other Helios processors connected to these. The next stage is to start looking at some memory. The current frame can be displayed by pressing the return key, which produces information such as the following example:

```
80000000: 00 00 00 80  00 00 00 80 ^@^@^@ .^@^@^@ .
80000008: 00 00 00 80  00 00 00 80 ^@^@^@ .^@^@^@ .
```

This shows the current memory address on the left, the contents of this memory in the middle, and the equivalent in ASCII characters (if defined) on the right. The current display mode is bytes. It can be changed to words simply by typing the **words** command.

```
words
80000000: 80000000 80000000  ^@^@^@ .^@^@^@ .
80000008: 80000000 80000000  ^@^@^@ .^@^@^@ .
```

To move to the next frame, type a '.'

```
.
80000010: 800fdd40 80000000  @ .^O .^@^@^@ .
80000018: 800fd110 800fcd00 ^P .^O .^@ .^O .
```

These addresses correspond to the input links, and give the same values as were listed earlier by the **explore** command. Various commands can be used to alter the display.

```
:8
80000010: 800fdd40 80000000  @ .^O .^@^@^@ .
,
80000008: 80000000 80000000 ^@^@^@ .^@^@^@ .
>
8000000c: 80000000 800fdd40 ^@^@^@ . @ .^O .
+4
80000010: 800fdd40 80000000  @ .^O .^@^@^@ .
```

According to the information produced by the **explore** command, the old instruction pointer was at 0x800028bb. This information could be shown again if necessary by typing the **info** command. To examine the code at this location it is necessary to move to that location and disassemble it.

```
28bb
800028bb: 75d5f222 76d3f474  " . . u t . . v
;
800028bb: 22 f2              ldtimer
800028bd: d5                 stl   00000005
800028be: 75                 ldl   00000005
800028bf: 74                 ldl   00000004
800028c0: f4                 diff
800028c1: d3                 stl   00000003
800028c2: 76                 ldl   00000006
```

If the user knows Transputer assembler language it may be possible to work out roughly what piece of code is being executed. This is made easier by the fact that the Helios C compiler normally puts function names into the binary program just before the code of the function, unless this has been suppressed with the **-g0** pragma. Hence by moving backwards through the code the user should eventually come back to a plain string, visible in the ASCII display on the right, identifying the function. This does not work with the Helios Nucleus or any Helios libraries, because these have been compiled with the special pragma to save memory.

It may be useful to see if there is anything in the trace vector. This is unlikely unless something has gone very seriously wrong and the Kernel could not recover, or unless some application contains its own calls to **_Trace()**. If there is nothing in the trace vector then the command will simply redisplay the current frame. Another useful check is **cmp** to see if the Nucleus has been corrupted. This may take a while, since it has to read a file from the disc and compare it with the processor's memory one integer at a time.

```
trace
800028bb: 75d5f222 76d3f474  " . . u t . . v
cmp
comparison finished
800028bb: 75d5f222 76d3f474  " . . u t . . v
```

Just to prove that **cmp** does work, it is possible to corrupt the system image in memory by poking it.

```
1500
80001500: 3273bf60 c03070d0  ' . s 2 . p 0 .
= 0
80001500: 00000000 c03070d0 ^@^@^@^@ . p 0 .
cmp
80001500: 00000000 != 60bf7332
comparison finished.
```

Finally, the root data structure can be examined. First it has to be found. This is done by moving to location 80001000 and indirecting through a self relative pointer (see below).

```
#80001000
80001000: 0000d764 00000020  d .^@^@  ^@^@^@
{
8000e764: 00000000 800fefb0 ^@^@^@^@ . .^O .
```

According to the **root.h** header file the data structure contains a pointer to a set of pointers to the **LinkInfo** structure, as defined in **link.h**.

```
typedef struct LinkInfo{
    byte       Flags;         /* flag byte             */
    byte       Mode;          /* link mode/type        */
    byte       State;         /* link state            */
    byte       Id;            /* link id used in ports etc */
    Channel    *TxChan;       /* address of tx channel  */
    Channel    *RxChan;       /* address of reception channel  */
    struct Id  *TxUser;       /* pointer to user of tx channel  */
    struct Id  *RxUser;       /* pointer to user of rx channel  */
    word       MsgsIn;        /* number of input messages  */
    word       MsgsOut;       /* number of output messages  */
    struct Id  *TxQueue;      /* queue of waiting transmitters  */
    struct Id  *RxId;         /* current message receiver  */
    word       spare1[2];     /* unused space  */
    struct Id  *Sync;         /* synchronisation point  */
    Port       LocalIOCPort;  /* port to be used by our LinkIOC  */
    Port       RemoteIOCPort; /* flag byte             */
    word       incarnation;   /* remote processor's incarnation no. */
    word       MsgsLost;      /* messages lost/destroyed  */
    word       spare2[2];     /* unused space  */
} LinkInfo;
```

It is possible to move to this structure element and indirect through it, remembering that all numbers are in hexadecimal.

```
+10
8000e774: 800fddc0 ffffffff  . .^O . . . . .
[
800fddc0: 800fddd4 800fde1c  . .^O .^\ .^O .
[
800fddd4: 00030270 80000000  p^B^C^@^@^@^@ .
```

Comparing this information with the **LinkInfo** structure indicates that this is link 0, state running, mode intelligent, and the link has the parent, ioproc, and debug flags set (remember that the debugger is currently showing words rather than bytes). The output channel is at 0x80000000 as expected. All indirections are put onto the stack so it is possible to move back to the previous table and go to the next link.

```
]
800fddc0: 800fddd4 800fde1c  . .^O .^\ .^O .
>
800fddc4: 800fde1c 800fde64 ^\ .^O . d .^O .
[
800fde1c: 01060000 80000004 ^@^@^F^A^D^@^@ .
```

This shows the **LinkInfo** structure for link 1, state dead, mode not connected, and no flags set. The output channel for this link is at location 0x80000004. The above example session has illustrated most of the features of the debugger.

8.5 The PC I/O server

PC Transputer boards such as the Inmos B008 board are among the most popular platforms for running Helios. Hence the PC version is one of the most important implementations of the I/O server. Unfortunately the PC I/O server may have to work in a fairly hostile environment. First, it has to work under MS-DOS and in the segmented architecture of the Intel 80x86 processor, yet provide access to all of the standard I/O facilities of a PC. Second, it has to work alongside hardware extensions to the standard PC such as local area network cards, and software such as device drivers that interact with the extra hardware. Third, there may be a number of other programs, particularly 'terminate and stay resident' utilities, in the PC at the same time that may get activated at unfortunate moments and play havoc with the I/O server's normal operation. Fourth, the I/O server has to work within the memory limit of a standard PC, 640K minus space for MS-DOS and other software that the user has installed.

Note: for these reasons, the PC I/O server is not guaranteed to run on all PCs, ATs, and their clones. It is not guaranteed to run on machines that have had extra hardware facilities installed. It is not guaranteed to coexist with any software other than MS-DOS, including hardware specific device drivers. However, all reasonable care has been taken to ensure that the I/O server runs on as many different machines as possible.

The PC version of the I/O server is shipped as a single program, **server.exe**, which is the I/O server itself. There is a separate utility **makedisk.exe**, described in the subsection on the **/rawdisk** device below. This section describes various facilities and options that are specific to the PC I/O server.

8.5.1 Hardware

The configuration file option specifying the host computer can be set to either **PC** or **AT**.

```
host = AT
```

In the original release of the PC I/O server, choosing between these affected certain delay loops. A **PC** host was assumed to run at less than 12MHz, and an **AT** host faster than 12MHz. In the current version the I/O server ignores this variable. The processor box definition can take one of four values: **B004**, **B008**, **MK026**, and **Cesius**.

```
box = B008
```

The B004 interface is based on the Inmos C012 link adapter, but does not support DMA. The I/O server interacts with the link adapter by polling the C012 chip in a very tight loop. The B008 interface is similar but does have DMA. However, on most machines DMA is actually slower than the polling loop. Whether or not DMA will be used depends on a separate flag **dma_channel**, described below. Most Transputer boards produced by other companies are compatible with either the B004 or the B008. An exception is the the Meiko MK026 card, a link adapter card that allowed users to connect a Computing Surface to a PC. The Cesius card is produced by Meiko. It has an MK026 style link adapter and one Transputer. In most cases specifying one of these four boards will suffice. Unfortunately there are always exceptions. The first problem

arises if the Transputer board is set up at a non-standard address within the PC's I/O address space. The default values are as follows.

Board	Address
B004	0x150
B008	0x150
MK026	0x100
Cesius	0x180

If the Transputer board is set to a different address, the configuration file variable **link_base** should be used:

```
link_base = 0x150
```

If the board specified is a B008 it is possible to enable DMA. This requires specifying the dma channel, usually a number between zero and three. The hardware documentation should be consulted for further details.

```
dma_channel = 1
```

The I/O server does not make use of any interrupts generated by the link adapter hardware. There are two additional variables in the configuration file that affect the link hardware, **reset_timeout** and **analyse_timeout**. It is very rare for users to have to change these, and some understanding of the exact workings of a Transputer is required. When booting up or debugging processors it is necessary to reset and analyse them by asserting a chip signal, waiting a while, and then releasing the signal. The I/O server contains some very simple code to do this. The following is used for resetting.

```
      assert signal
      mov cx, <reset_timeout>
 wait:
      loop wait
      release signal
      loop <reset_timeout> times
```

Analyse is similar:

```
assert analyse
loop <analyse_timeout> times
assert reset
loop <reset_timeout> times
release reset
loop <reset_timeout> times
release analyse
loop <reset_timeout> times
```

A problem arises because different machines may need different values for the delay count. In theory a 25MHz machine needs four times the delay of a 6MHz machine, yet the I/O server should run on both. On the other hand, spending an excessive amount of time looping simply causes unnecessary delay. In practice the I/O server uses values

of 8000 for **reset_timeout** and 4000 for **analyse_ timeout**, and these appear to work on all machines without causing an unacceptable delay. Should problems arise in this area then the delay counts can be modified by the user. However, it is unlikely that problems will be encountered in this area.

8.5.2 Special keys

The PC version of the I/O server has two types of special keys. First, when the **server windows** system is running the keys Alt-F1 and Alt-F2 are used to switch windows. To switch to the next window, hold down the Alt key and press function key 1. To switch to the previous window hold down the Alt key and press function key 2. If the server windows system in not running then these keys can be read by a program under Helios.

The PC I/O server does not have a special key for refreshing the current window because in the single tasking environment of MS-DOS it is difficult for any program other than the I/O server to write to the screen and thus corrupt the current window. When the raw keyboard device is enabled the Alt-F1 and Alt-F2 combinations stop working. Hence it is not possible to have multiple windows on the PC side and simultaneously run the Helios X window system, because the X Server requires raw keyboard events. The other type of special key is used for the debugging options and for the four special actions: reboot, exit, status, and debugger. This requires holding down a control and a shift key at the same time, and then pressing another (specified) key. For example: to enable or disable message debugging the user should press CTRL-SHIFT-M, since the letter `M` is associated with message debugging. For the four special actions, function keys are used (see the table below).

Action	Key
Reboot	control-shift-F10
Exit	control-shift-F9
Status	control-shift-F8
Debugger	control-shift-F7

On some PCs not all combinations of shift and control keys are detected correctly by the underlying system software. For example, pressing the left control key and the right shift key may not work as expected. To date, every machine has had at least one combination of control and shift keys that worked reliably, but some experimenting may be necessary.

The control-shift combinations will continue to work even when the raw keyboard is enabled. Furthermore, some of the key presses are actually stripped out. For example, the user may be running the X window system and want to enable the **Open** debugging option. This means pressing down the shift key, the control key, and the `O` key, then releasing these keys in the reverse order. Pressing the shift key will generate a key-down event, as will pressing the control key. Pressing the `O` key will enable the debugging option, but will not generate a key event. Releasing the `O` key has no effect. Releasing the control and shift keys will generate two key-up events.

8.5.3 File I/O

The PC I/O server provides the standard **/helios** file server, mapped onto a directory in the PC's filing system. In addition there are servers **/a**, **/b**, **/c**, **/d** and so on for the various disc drives and partitions. Hence the Helios command

```
ls -l /c/users
```

gives similar information to the MS-DOS command

```
dir c:\users
```

There is a problem with floppy discs on PCs. All PCs appear to come with two floppy drives **a:** and **b:**, even if there is only one real floppy. This is achieved by mapping two logical disc drives onto one physical disc drive. Accessing the wrong floppy produces the following message.

```
Insert diskette for drive B: and press any key when ready.
```

Now consider what may happen under Helios, when the user tries to access the server **/b**. MS-DOS would give the same error message and suspend the I/O server until it detects a key press. Hence the I/O server stops responding to requests from the Helios network, and after a while Helios will decide that the I/O processor has crashed and will stop using it. Since the user interacts with Helios through the I/O processor, the Helios network is no longer accessible for the user.

A fast user could recover from this error by putting in the right floppy disc and hitting a key. Unfortunately if the user is running the X window system in the Transputer network and the X server has taken over the PC's keyboard, through the **/keyboard** device, then the I/O server will intercept all key presses through its interrupt handling routines and MS-DOS will never detect a key. Hence there is no way for the system to recover.

Unfortunately MS-DOS does not provide a simple and reliable way of determining whether there are two separate floppy disc drives or only one. There is a system call:

```
get peripheral equipment list
```

at interrupt 0x11, but the values returned do not appear to be correct on all machines.

Defining available floppy disc drives

It is the user's responsibility to specify which floppy disc drives should be accessible from Helios, with a configuration file option.

```
floppies = a
floppies = ab
```

If no **floppies** variable is defined in the configuration file then it will not be possible to access drives **a:** and **b:** from Helios. If only drive **a:** is specified in the configuration file then there will be a server **/a** but no server **/b**. If both drives are specified with the **floppies** variable then both servers will exist.

Certain local area network systems, notably PC-NFS, cause a different problem. When installed these create pseudo-drives **e:**, **f:**, and so on all the way to **v:**. These

drives are not actually used unless the user explicitly mounts them as network drives. Unfortunately the I/O server has no simple way of distinguishing these pseudo-drives from real drives, so normally the I/O server would create servers **/e**, **/f**, and so on. Every server requires a certain amount of memory, so clearly this is wasteful. To avoid this waste, the PC I/O server only creates servers for drives that are not empty. Hence if the PC is equipped with a real disc partition **e:**, but no file has been into this partition, then the I/O server will not create a server **/e** and the partition will not be accessible from Helios. To avoid this problem, simply copy one file into the empty partition before running the I/O server.

There is a problem with text files under MS-DOS. In a Unix or similar system, linefeed characters are used to split lines of text. A single character is perfectly adequate for this job. Unfortunately under MS-DOS two characters are used, a carriage return character and a linefeed. Programs running under MS-DOS, including PC editors, will use the extra carriage return character. Also, such programs may put CTRL-Z characters at the end of the file to indicate 'end-of-file'.

Helios prefers to use the Unix system of a single linefeed character to split two lines of text. However, it is still very desirable to be able to read text files produced under MS-DOS from Helios, and vice versa. To cope with this Helios distinguishes between text file I/O and binary file I/O, and between MS-DOS style filing systems and normal filing systems. At the Helios System library level and at the Posix library level, all file I/O is in binary mode. At the C library level, applications have the choice of operating in text or binary mode:

```
FILE *text_stream   = fopen("myfile", "r");
FILE *binary_stream = fopen("myfile", "rb");
```

Some Helios commands work in binary mode. For example, the command:

```
cp /helios/include/stdio.h /ram/stdio.h
```

does a binary copy of an MS-DOS format text file into a normal filing system, leaving the spurious carriage returns in place. Examining the file will show up the carriage characters. Binary mode must be used when transferring true binary files such as programs. There is a separate command: **tcp**, to copy text files, stripping out the unwanted characters.

Although the Helios C library and other Language libraries are capable of stripping out the spurious characters in MS-DOS format text files, inherently this involves a small overhead. If the PC is used primarily or entirely in order to run Helios, it may make sense to switch all files to a Unix mode and inform Helios to treat the PC drives as normal Unix drives rather than MS-DOS style drives. This is achieved with the **host.con** flag:

```
unix_fileio
```

Before enabling this option it is necessary to convert all text files on the PC from MS-DOS format to normal format, usually with the **xlatecr** command. If the option is enabled before all text files are converted then the remainder may cause problems. This is particularly true of system configuration files such as **/helios/etc/initrc**. To interact with the MS-DOS filing system the PC I/O server uses normal MS-DOS system calls, interrupt 0x21.

8.5.4 Multiple windows

The PC I/O server implements the usual server windows interface. The I/O server assumes it runs in a 25 rows by 80 columns text mode, and allows any number of text windows of this size subject to memory limitations. The keys used to switch between windows are Alt-F1 to switch to the next window and Alt-F2 to switch to the previous window. All output to the screen goes through the PC BIOS calls, using interrupt 0x10. These have proved portable across all types of PC and give an acceptable performance. There is no need to install the MS-DOS **ansi.sys** screen driver in order to run the I/O server. Reading the keyboard is done through the keyboard BIOS routines, interrupt 0x16.

8.5.5 The error logger

The PC I/O server implements the standard error logger device. When error logging goes to the screen it is handled by the same code as for the window server described above, and output goes through the BIOS routines at interrupt 0x10. When error logging goes to a file it uses the MS-DOS system routines at interrupt 0x21.

8.5.6 The clock device

The PC I/O server implements the standard clock device. In the PC implementation this clock device can be written, so Helios applications are able to change the time and date maintained by the I/O server. The I/O server uses the MS-DOS system call at interrupt 0x21, function codes 0x2A, 0x2B, 0x2C, and 0x2D, to interact with the PC's clock.

8.5.7 X window system support

When running the X window system in the Helios network the X server will need access to a mouse and keyboard device. The PC I/O server can provide these devices through **/mouse** and
/keyboard servers, if desired. However, the default PC installation of Helios does not include the X window system so by default these servers do not exist. To enable these servers it is necessary to set a flag in the configuration file **Xsupport**.

The raw keyboard device works by taking over the PC's keyboard interrupt vector 0x09 and interacting directly with the keyboard chip. This prevents the PC BIOS and MS-DOS from detecting any keyboard presses, for example the caps-lock key will no longer light up the keyboard 'LED' and the key sequence CTRL-ALT-DEL will no longer cause a reboot. (The reset button should still work, and so should powering down the PC.) It will not be possible to read keyboard data from the **/window** or **/console** devices, and it will not be possible to switch windows. However, the various CTRL-SHIFT key pressing sequences such as CTRL-SHIFT-F9 to exit the I/O server, will still work, and exiting will restore the PC to its normal state. There are no configuration options for the raw keyboard device.

When using a keyboard other than a standard UK QWERTY keyboard another problem may arise. Different keyboards tend to use the same scancodes for shift keys, control keys, and so on. This is not true for letters. For example, when the I/O server

detects scancode 0x11 down, this corresponds to character `w` on a QWERTY keyboard but character `z` on a French AZERTY keyboard. When the raw keyboard device is not enabled this does not cause any problems, since the BIOS routines perform the scancode translation. Otherwise the I/O server assumes that a QWERTY keyboard is in use. If a French AZERTY keyboard is attached then when not running the X window system, CTRL-SHIFT-Z will cause the configuration file to be reread. When the X window system is being used CTRL-SHIFT-Z will toggle write debugging, because the I/O server assumes that the key pressed is `w`.

The raw keyboard device relies on a PC with standard hardware, in terms of the chips used and so on. Hence it is possible that the device does not work correctly on some PCs, because of subtle or unsubtle differences in the hardware. The mouse device is more complicated, because there are various different PC mice. For portability the PC I/O server does not interact with the mouse hardware directly. Instead the user has to install the mouse device driver that (usually) comes with the mouse hardware before running the I/O server. The I/O server is designed to interact with the Microsoft mouse device driver, but most other device drivers are compatible with this.

When using serial mouse devices there may be a clash between the mouse driver and the I/O server code to drive the serial lines. This is described in section 8.5.8. There are two configuration file options to control the sensitivity of the PC mouse. Mouse sensitivity causes two problems. First, how much mouse movement should correspond to a single unit movement? If a tiny movement of the mouse corresponds to a single unit then accurate positioning becomes difficult. If a large movement corresponds to a single unit then it takes a long time to move the mouse cursor from one side of the screen to the other. Second, how many such units should be accumulated before they are packed into a message and sent off into the Helios network? If an event message is generated for every unit of movement then the link between the I/O processor and the root processor can be saturated very easily. If an event message is generated only for a large number of movement units, accurate positioning is difficult. The configuration file options are:

```
mouse_divisor    = 5
mouse_resolution = 1
```

The **mouse_divisor** variable controls the amount of mouse movement corresponding to a single unit. The **mouse_resolution** variable corresponds to the number of units that have to be recorded before an event message is generated. The default for both variables is one.

8.5.8 Serial ports

Typical PCs are equipped with one or more serial or RS232 ports, known as **COM1:**, **COM2:**, and so on. These serve several purposes: dumb terminals can be hooked into the PC; the PC can be used as a dumb terminal to some other machine such as a Unix workstation; a serial printer can be attached to the PC; a serial mouse can be plugged into the PC; a dial-up modem may be connected; or some other hardware can be attached. Most of these should also be possible from Helios, so the PC I/O server must provide support for the RS232 hardware and allow Helios applications to access the RS232 ports. The **/rs232** hardware only provides a transport layer. If data

is written to, say, **/rs232/default** then it will be sent out of the appropriate serial port. Higher-level software such as printer spoolers or terminal drivers should be used to perform any higher-level formatting that may be required.

There is a problem with supporting RS232 hardware, because the support that is provided by MS-DOS and the lower-level BIOS facilities is not sufficiently powerful to meet the requirements of non-trivial applications. Hence the I/O server has to take full control of the serial line chips (the 8250 UARTs) and the appropriate interrupt lines. Like the raw keyboard device this relies on having fairly standard hardware in the PC, and hence there may be some PCs where the **/rs232** server does not work as expected or at all. It is even possible that when the I/O server attempts to access the RS232 hardware it will crash, because the hardware does not behave as expected. To avoid this problem the **/rs232** server is disabled by default, and the I/O server will only support this device if the following variable is specified in the configuration file.

```
rs232_ports = 1,2
```

This line specifies that the serial lines for **COM1:** and **COM2:** should both be accessible from Helios. It is possible to restrict access to just one port or to any number of ports.

```
rs232_ports = 1
rs232_ports = 2,3,5,6,7
```

Unless the user actually needs to make use of the serial lines from Helios, it is usually easier to comment out this variable and not have the **/rs232** server.

The assumptions made by the I/O server are as follows.

1. Every serial port is controlled by an 8250 UART or a compatible chip
2. The **COM1:** port is at I/O address 0x03f8
3. The **COM2:** port is at I/O address 0x02f8
4. The user has to supply the addresses of any other ports in the configuration file
5. Either interrupt vector 0x0B (usually **COM2:**) or interrupt vector 0x0C (usually **COM1:**) may be used for any of the ports, so the I/O server has to check all UARTs to see which one(s) interrupted

If the PC is equipped with more than the usual one or two serial ports then the I/O server has no simple way of finding out where these ports are located. Hence the user has to supply the base addresses in the configuration file.

```
rs232_ports = 1,3,4
com3_base   = 0x300
com4_base   = 0x308
```

Usually the **/rs232** server takes over both interrupt vectors 0x0B and 0x0C, no matter which ports are specified. This can cause problems when a serial mouse is attached to one of the ports, because the mouse driver expects to receive interrupts. Consider the case where the mouse is attached to **COM2:** and the user's Helios application needs to access **COM1:**. It is necessary to limit the I/O server to interrupt vector 0x0C only, (the interrupt vector for **COM1:**). The following configuration file entry achieves this.

```
rs232_interrupt = 1
```

Setting the variable to two means that the I/O server will only take over interrupt vector 0x0B, corresponding to **COM2:**. If in doubt the user can experiment. There is a 50 percent chance that the user will guess right first time. It is possible to imagine configurations where the I/O server's configuration cannot cope. For example, ports 1 and 3 could be hooked to interrupt vector 0x0C, ports 2 and 4 to interrupt vector 0x0B, and the user has attached a serial mouse to port 2. There is no easy way by which the user can exploit both the serial mouse and port 4 from Helios.

The PC **/rs232** server behaves like a standard Helios RS232 device. If only one port is accessible from Helios then the **/rs232** directory will contain a single entry, **default**. If two ports are specified in the configuration file, say **COM1** and **COM2**, then the **/rs232** directory will contain three entries: **default**, **com1**, and **com2**. The **default** entry will be equivalent to either the **com1** or the **com2** entry, and it is possible to use the **mv** command or the **Rename()** library routine to change the default. By default the first port which is accessible or named in the configuration file will also be the **default** port (usually **com1**). However, it is possible to specify an alternative in the configuration file.

```
rs232_ports = 1,2
default_rs232 = 2
```

Both **COM1** and **COM2** are accessible from Helios, so normally **default** is equivalent to **COM1**. The second line changes the default to **COM2**.

8.5.9 Parallel ports and printers

In addition to the serial ports, the I/O server also provides access to the parallel or Centronics ports. In this case the BIOS routines provided by the PC are sufficiently powerful to allow the I/O server to drive the hardware, and there is no need to interact directly with the hardware. This makes configuration much easier. A typical PC has one or more Centronics ports, known as **LPT1:**, **LPT2:**, and so on. There is a system call to allow the I/O server to find out how many ports there are, and there is no need to interact with the chip(s) controlling the ports, so there is no equivalent to the **rs232_ports** variable in the configuration file. Instead the I/O server always provides a **/centronics** server, unless explicitly suppressed with the **no_centronics** flag or the PC does not have any parallel ports. A typical **/centronics** server would contain three entries: **default**, **lpt1**, and **lpt2**. Following the form of the serial lines, the **default** entry is equivalent to one of the others, usually **lpt1**, but a configuration file entry can change the default.

```
default_centronics = lpt2
```

The **/printers** device is a combination of all serial and parallel ports. On a typical PC the **/printers** device would contain entries **default**, **com1**, **lpt1**, and **lpt2**, meaning that the PC is equipped with one serial printer and two parallel printers. The **/printers** server contains the same entries as the **/rs232** and **/centronics** servers put together, so if a serial port is not accessible through the **/rs232** server it will not be accessible through the **/printers** server either. As with the **/rs232** and **/centronics** server it is possible to specify the default printer in the configuration file.

```
default_printer = com2
default_centronics = lpt1
```

Both the **/centronics** and **/printers** devices only provide a transport layer. If data is written to the device **/centronics/lpt2** it should come out of the corresponding parallel port unchanged. Higher-level software such as printer spoolers are responsible for performing any formatting that may be required.

8.5.10 The rawdisk device

Most of the time, any file I/O performed using the PC hard disc or floppies makes use of the underlying MS-DOS filing system. MS-DOS takes care of allocating disc blocks when required, maintaining the directory structure, and so on. In any PC Helios installation there must be at least one disc drive or partition that is maintained by MS-DOS, so that the I/O server can be run from that partition. There is an alternative to using MS-DOS as the underlying filing system for at least some of the user's file I/O requirements, and that is using the **/rawdisk** device. Suppose the user has a 40 megabyte hard disc, formatted as a 32 megabyte **c:** partition and an 8 megabyte **d:** partition. In this case it is possible to turn partition **d:** into a Helios rawdisk, and read and write disc sectors directly bypassing the MS-DOS filing system. This avoids the overheads of MS-DOS, but leaves all the work of organising the hard disc to the user's application. Usually this would be the Helios filing system, but it does not have to be.

For example, the user's application generates a table of 6 megabytes during its initialisation phase and then needs to access this table as quickly as possible. This table does not fit into any processor's memory, so it has to be stored on disc. Once the application has finished the table can be deleted. For maximum performance the user could turn the **d:** partition into a rawdisk of 8 megabytes. The application opens a stream to **/rawdisk/0** and writes the table of 6 megabytes to it, using up 12288 sectors of 512 bytes each. The application can then seek to a particular sector on the disc by seeking within the rawdisk device, and read that sector or a number of sectors off disc. The MS-DOS filing system is not involved in any of this, so the whole table is guaranteed to be continuous on disc and there are no overheads in writing or reading the data. A given disc drive or partition cannot be used by MS-DOS and as a rawdisk device at the same time. To turn an MS-DOS drive or partition into a rawdisk device it is necessary to use the **makedisk** command. This causes any data on the drive to be lost or, rather, to be very difficult to access. After use of **makedisk** MS-DOS will no longer be able to write data to that drive. As far as the MS-DOS filing system is concerned the drive does not contain any files, but all available disc sectors have been used up. To make the drive usable again by MS-DOS it will have to be reformatted, using either the **format** or the **fdisk** commands. Again, this will cause any data on the disc to be lost. The **makedisk** command is shipped with the standard Helios product. It should be invoked with a single argument, the letter of the drive or partition to be modified, for example

```
makedisk a
```

When invoked it will produce output such as:

```
Disc statistics :
The following information must be put in the devinfo file.
    sectorsize        512
    sectors           17
    tracks            9
    cylinders         272

About to convert disk to raw format.
This will erase all data on the disk.

Press return to continue, or ctrl-C to abort.
```

The disc statistics information is used for configuring the file server to use the rawdisk device. The user is given one last chance to abort the process, if desired. For safety reasons the **makedisk** command refuses to modify the **c:** partition of the hard disc, because in most installations this is the boot disc for the whole PC. Please note that some of the sectors near the start of the partition or disc will not be accessible from Helios, since these sectors must still be understood by the MS-DOS filing system to stop it writing to the disc. Once one or more discs or drives have been converted to raw format the I/O server should be informed about them. This is done with a configuration file variable:

```
rawdisk_drive = da
```

This means that the device **/rawdisk/0** corresponds to drive **d:**, and **/rawdisk/1** to drive **a:**.

8.5.11 The /pc device

The PC I/O server only provides access to the devices which are available on most PCs, and this suffices for nearly all users. To access other hardware it is usually necessary to purchase and modify the I/O server sources. However, there is a very limited facility by which users can install a 'terminate and stay resident' program before running the I/O server, and then activate this program from a Helios application. That application should open a stream to a device **/IO/pc**, and then perform message passing on the resulting stream. When the I/O server receives such a message it will invoke software interrupt 0x60, which should be the user's program. When the interrupt returns the I/O server will send back a reply message. Consider an example program to do this. First, the Transputer side:

```
/**
*** This program illustrates how to activate trap 0x60
*** on an IBM PC or compatible. This allows you to
*** install your own Resident modules and access them
*** from Helios, (to control devices not supported
*** by the standard Helios server).
**/
#include <stdio.h>
#include <stdlib.h>
#include <syslib.h>
```

```
#include <message.h>

int main(void)
{ MCB           mcb;            /* for message passing */
  Object        *IO;
  Stream        *stream;
  Port          reply_port;
  WORD          result;
  int i, j;

            /* First, find the server */
  IO = Locate(NULL, "/pc");
  if (IO == Null(Object))
    { printf("Unable to locate /pc - exiting.\n");
      exit(1);
    }
            /* Second, open a Stream to that server */
  stream = Open(IO, "/pc", O_ReadOnly);
  if (stream == Null(Stream))
   { printf("Unable to open /pc - exiting\n");
     Close(IO);
     exit(1);
   }
            /* Get a reply port for message passing */
  reply_port = NewPort();
  if (reply_port == NullPort)
    { printf("Unable to get reply port - exiting\n");
      Close(IO);
      exit(1);
    }
            /* Fill in a message structure */
  mcb.MsgHdr.DataSize = 0;
  mcb.MsgHdr.ContSize = 0;
  mcb.MsgHdr.Flags    = MsgHdr_Flags_preserve;
  mcb.MsgHdr.Dest     = stream->Server;
  mcb.MsgHdr.Reply    = reply_port;
  mcb.MsgHdr.FnRc     = 0x20765432;
  mcb.Data            = Null(BYTE);
  mcb.Control         = Null(WORD);
  mcb.Timeout         = 10 * OneSec;

            /* Try to send the message */
  result = PutMsg(&mcb);
  if (result != 0)

   { fprintf(stderr, "PutMsg returned %lx\n", result);
     Close(IO);
     Close(stream);
     FreePort(reply_port);
     return(result);
   }

            /* and wait for the reply */
```

```
  mcb.MsgHdr.Dest = reply_port;
  (void) GetMsg(&mcb);

            /* Tidy up and exit */
  Close(IO);
  Close(stream);
  FreePort(reply_port);
  return(0);
}
```

The PC side is detailed below.

```
;
; Resident program to test the Helios call-trap facility.
;
; Install a simple routine at interrupt 0x60, to be called
; by the PC I/O server when it receives a private protocol
; message for the /pc device. The routine gets a pointer
; to the Server MCB in registers ds:dx, allowing it to
; manipulate the message before it is sent back to the
; client - make sure that the data size and control size
; entries in the MCB are set correctly. The routine should
; return in under two seconds, with a 32-bit reply code in
; dx:ax - this is sent back as the message FnRc to the client.
;

cseg    segment para  public  'CODE'

        org     100H

        assume  cs:cseg, ds:cseg, es:cseg, ss:cseg

Init    proc    near

        mov     dx,cs           ; force all segment registers
        mov     ds,dx           ; to a sensible value
        mov     es,dx
        mov     dx,offset trap  ; install routine "trap" at
        mov     ax,2560H        ; interrupt vector 0x60
        int     21H

        mov     dx,offset signon ; display a message to show that
        mov     ah,9            ; the routine is installed
        int     21H

                                ; terminate but stay resident
        mov     dx,((offset Pgm_Len+15)/16)+20H
        mov     ax,3100H
        int     21H

Init    endp

;
```

```
; This is the routine that will be activated by the Server
;
trap    proc    far
        sti                     ; make sure that interrupts are
                                ; enabled

        mov     dx,offset warn  ; display a message to show that
        mov     ah,9            ; the trap has been activated
        int     21h
        mov     dx,8765H        ; return code 0x87654321
        mov     ax,4321H

        iret

trap    endp

cr      equ     0dH
lf      equ     0AH

signon  db      cr,lf,'Trap 60 handler installed',cr,lf,'$'
warn    db      cr,lf,'Trap 60 activated',cr,lf,'$'

Pgm_Len equ     $-Init          ; size of this program,
                                ; needed to terminate
                                ; but stay resident
cseg    ends

        end     init
```

The example PC program simply displays a message when it is run, and another message whenever a Helios application causes the program to be invoked. On the Helios side the following points should be remembered. The example code does not take into account any of these points, to keep it simple.

1. There may be more than one **/pc** server in the network. In fact there may be a processor called **/pc**, containing a server **/pc/pc**.

2. Message passing is unreliable. It is possible for either the request or the reply message to get lost, for example because an intermediate processor crashed or ran out of memory. The protocol used between the application and the PC program should be able to cope with errors of this sort.

3. Streams tend to 'time out' under Helios. If a stream has not been used for half an hour or so then the corresponding server is allowed to assume that the client has gone away or crashed, without closing the stream. The behaviour observed by the application is similar to that for the previous case, because message passing fails.

The PC side is rather more complicated.

1. The trap routine will be invoked with a pointer to a message control block or MCB in registers **ds:dx**. The MCB structure is defined in the header file **message.h**

2. The message will be in Transputer format, so an integer in the structure is a 32-bit long on the PC side

3. The trap routine should return a function code in the **dx:ax** register pair. This function code will be installed in the message. In addition the message ports will be adjusted as required.

4. The trap routine can send back data if required. The MCB will contain a suitable control and data vector, which can be filled in by the trap. The trap routine should also fill in the control vector size and data vector size in the message header, or the I/O server will be unable to determine how much data to send back. The other fields in the MCB must not be changed by the trap routine.

5. The I/O server will preserve all registers on the stack. The trap routine must not corrupt the stack, and should not use more than about 500 bytes of stack.

6. If the PC program consists of a mixture of C and assembler then the programmer is responsible for ensuring that these work together correctly. In particular the various segment registers may have to be set up correctly before calling C routines, to allow access to static data.

7. If the trap routine is going to perform file I/O it must set up the Program Segment Prefix or PSP correctly, as well as some other MS-DOS specific variables.

8. The trap routine must not suspend the I/O server indefinitely. In particular, some other Helios application may want to access a server inside the I/O processor and will try to send a request down the link. Unless the I/O server accepts this request within a few seconds things will start to go seriously wrong. As a general rule the trap routine should return within about two or three seconds.

A more complicated example of such a 'terminate and stay resident' utility, and a Helios application that interacts with it, is shipped with the PC graphics library. For details of writing such programs generally, and in particular how to perform file I/O, the *MS-DOS Encyclopedia*[8] may prove useful.

8.6 The Sun I/O server

After PC Transputer boards, the various types of Sun plug-in boards are probably the most popular type of Transputer expansion system. Amongst the manufacturers producing Sun boards are Inmos, Parsytec, and Transtech. This section describes the Sun implementation of the I/O server, which can drive all of these boards.

[8] Published by Microsoft Press, a division of Microsoft Corporation.

8.6.1 Introduction

There are two quite fundamental differences between the PC I/O server and the Sun implementation. A typical PC plug-in board has a single link adapter and one or a small number of processors. It is unusual to have more than one such board in a PC, because in a single-tasking operating system like MS-DOS it is not possible to run several different copies of the I/O server. A typical Sun board might come with four link adapters, allowing four users at the same time. Each link adapter is known as a **site**. The I/O server must be able to cope not only with different hardware, but also with the different sites on a given piece of hardware. Every site allows one I/O server, so if there are four sites then it is possible to have four copies of the I/O server running simultaneously, each supporting one user.

The second difference involves local area networking. Most Suns tend to be attached to an ethernet, and software should be able to exploit this. In particular it should be possible to access some Transputer hardware plugged into a Sun somewhere else in the network, yet still be able to access all the resources in your own machine. This will be slower than accessing the Transputer hardware directly, because most of the communication now has to go over the ethernet as well as across the link. This gives us the picture of the world shown in Figure 8.7.

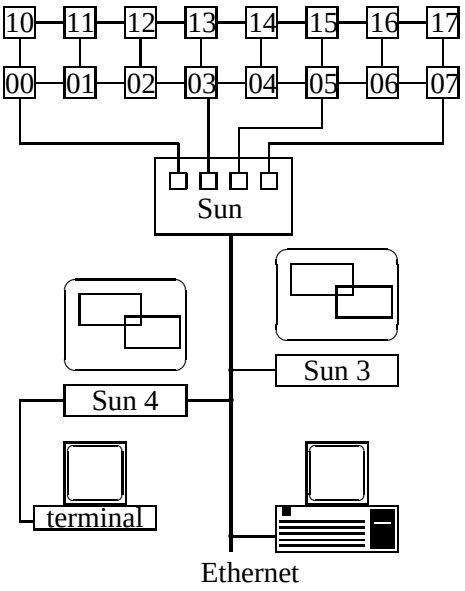


Figure **8.7** A Sun network

At the top is a Helios network, with 16 processors but it could be any number. There is a Sun with four sites, possibly but not necessarily just a file server. This Sun is attached to an ethernet network. Also on the network are a Sun3, a Sun4 with a dumb terminal attached through a serial port, and a PC running PC-NFS and Telnet. On the two Suns the users may be running SunView or the X window system. The dumb terminal may be a ‘standard’ VT100 terminal, or something very obscure. It should be possible to

run Helios from any of these machines and still gain access to the resources of the user's own machine.

To cope with these requirements Helios comes with four separate programs to run on Sun hosts. All of these are shipped in Sun3, Sun4, and Sun386 versions.

1. **server**: the I/O server itself. This should always run on the user's Sun. It can access a plug-in board inside that Sun, or some remote board accessible through the ethernet.

2. **hydra**: a link daemon. This program can only run on a Sun equipped with the plug-in boards. It allows an I/O server running somewhere else to access the board, over the ethernet.

3. **hydramon**: a little administration program to control the behaviour of the link daemon.

4. **serverwindow**: a separate program which is started by the I/O server to interact with SunView[9].

In the diagram all four sites are shown connected to a single Helios network. This is one way of configuring the system, but not the only way. Another configuration would have all four sites going into four separate, non-overlapping networks. Alternatively there could be two networks, with two sites going to each network, or some other combination.

In addition to the usual I/O server configuration file **host.con**, the Sun system uses a separate configuration file **hydra.con** for the link daemon **hydra** and the utility **hydramon**. This new configuration file uses the same syntax as the **host.con** file.

8.6.2 Hydra

Hydra is the Helios link daemon. It is run on the Sun containing the processor network, and allows access to this hardware from any other Sun connected to the same ethernet network. The I/O server is run on these other Suns, and gives access to all the resources of these other Suns including the graphics display. This is shown in Figure 8.8.

In most installations the hydra program is controlled by the system administrator and started up automatically when the Sun boots up, typically as a local service in the Sun's **/etc/rc** file. Alternatively the program can be started up manually at any time by any person who has access to the binary. The command line syntax for hydra is:

$$hydra[-C < hydra - configuration - file >]$$

By default the program reads the configuration file **hydra.con** from the current directory. A different configuration file can be specified on the command line. It must contain the following information.

host The type of I/O machine.

box The type of external hardware.

hydra_host The ethernet name of the Sun host.

[9] Trademark of Sun Microsystems

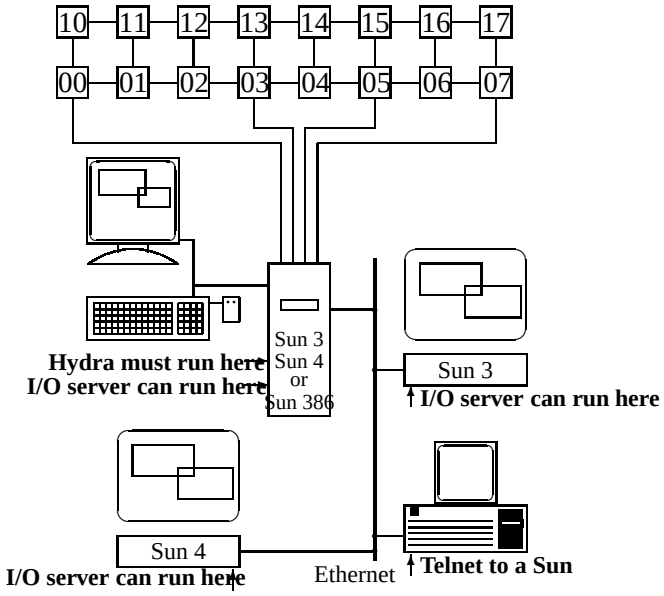


Figure **8.8** Hydra and the I/O server

address The Unix or Internet socket name.

site description The accessible sites.

In normal operation Hydra behaves as a TCP/IP service. If it is to work correctly then this service must be registered with the Sun's operating system. In particular there is a system resource file, **/etc/services**, which should be changed on the machine running Hydra and every machine which may run the I/O server. This file can be changed only by the system administrator, and in many installations the Network Information Service (NIS) can take care of most of the work. The following line should be added to the **/etc/services** file.

```
hydra           1234/tcp        # Helios link daemon
```

This gives the name of the service, `hydra`, a unique socket number which must not clash with any other service socket number in the network, and a specification that the service is a **tcp** service rather than **udp** or some other protocol. A typical **hydra.con** file looks like this.

```
host             = SUN
box              = IMB
hydra_host       = Molly
#family_name     = AF_UNIX
#socket_name     = /tmp/hydra
family_name      = AF_INET
connection_delay = 5
all_sites
#imb0
#imb1
```

The various fields in the configuration file have the following meanings:

1. **host** serves to identify the host computer. It is somewhat redundant since the program has been compiled for a Sun anyway, but it provides a useful validation check for the configuration file.

2. **box** identifies the Transputer hardware to be used. The various possible boxes are listed in section 8.6.5.

3. **hydra_host** is the ethernet network name of the Sun containing the Transputer hardware. Every machine attached to an ethernet has two identifiers, a network number such as 89.0.0.150, and a text name such as Sun5 or Molly. These identifiers can be found in the Sun configuration file **/etc/hosts**. The **hydra_host** variable should be set to this text name.

4. **family_name**. Under SunOS there are two types of sockets. The Unix sockets of the family **AF_UNIX** are internal to the machine and do not allow remote access. Internet sockets, family **AF_INET**, do allow remote access. The **AF_UNIX** family is supported only for debugging purposes or when there are problems with the installation process, and normally the **AF_INET** family should be used.

5. **socket_name** is used only in conjunction with **AF_UNIX** sockets as described above. The socket used by the daemon and the clients must be created within the Unix filing system, so a file name must be supplied.

6. **connection_delay**. When a client first connects to the link daemon and boots Helios into a processor the initial communications traffic is very heavy. If various people attempt to do this at the same time then there may be timing problems. Hence there is always a minimum delay between successive connections, the default being ten seconds. Some other delay can be specified using this variable.

7. **all_sites** specifies that all the available sites should be accessible over the ethernet. In most installations this is the default. The alternative is to specify only those sites that can be accessed over the ethernet, by listing the names of those sites.

8.6.3 Hydramon

Most of the time there should be no need to interfere with the behaviour of hydra, but just in case a utility is provided to interact directly with the link daemon and perform various operations. This utility is known as hydramon. This utility is usually available only to the system administrator, or to trusted people. The hydramon command line options are:

$$hydramon[-C < hydra - configuration - file >]$$

The hydramon program needs to read the same **hydra.con** configuration file as the link daemon. By default it will attempt to read this file from the current directory, but an alternative file name can be specified on the command line. When hydramon starts up it connects to the link daemon and should give the menu:

```
Site 0 (IMB0) : running, owned by bart @ Folly
(Top, Bottom, Next, Prev, Disconnect, Use, Free, Help, Quit)
?
```

The first line gives the current site number and name, and the state of the site. In the example shown the site is currently running a Helios session for user `bart` on machine `Folly`. An owned site may also be **reset**, usually because the user is debugging that site, or **booting** if the user is in the middle of booting up that site. Sites that are not owned can be **free** or **unused**. A **free** site is accessible through hydra, but nobody is currently accessing this site as far as Hydra is aware. There may be some other application using the site without going through hydra. An **unused** site is not currently accessible from Hydra. Typically this would happen if the site was not specified in the **hydra.con** configuration file.

The various commands perform the following actions.

1. **Top** moves to site 0.

2. **Bottom** moves to the last site known to the link daemon. This depends on the hardware available.

3. **Next** moves to the next site, allowing the user to step through the various sites and find out what is happening on each site.

4. **Previous** moves to the previous site.

5. **Disconnect** can be used only on sites that are currently owned. It forcibly disconnects the session and sets the site back to free. The session is aborted abruptly with no opportunity to save any data or perform any tidying up, so this option should be used with care.

6. **Use** takes an unused site, one that is not currently accessible through the link daemon, and sets it to free mode so that it is now accessible. This can be used to modify which sites are accessible, without modifying the **hydra.con** configuration file and restarting the link daemon.

7. **Free** performs the inverse operation to **Use**. It takes a free site and changes it to unused, making it inaccessible through the link daemon.

8. **Help** gives much the same information as described here.

9. **Quit** terminates the hydramon session.

8.6.4 Supported hardware

The Sun implementation of the I/O server works in terms of sites. Essentially a site is some form of link adapter connecting the Sun host to a processor network. The current release of the I/O server supports the following hardware.

Transtech MCP1000

This VME board, previously known as the Niche NTP1000, is suitable for Sun3 and Sun4 and comes with four sites. The sites are called **nap0**, **nap1**, **nap2** and **nap3**, corresponding to device drivers **/etc/nap0** and so on. It is possible to plug several of these boards into one Sun, giving additional sites **nap4**, **nap5** and so on.

Inmos B011

This board is supported by the I/O server. It is a VME board suitable for Sun3 and Sun4, but only has one site. The board is accessed by direct access to the VME bus, and it is not possible to have more than one board in one Sun.

Inmos B014

This is another VME board with just one site. Helios does support more than one of these boards in a Sun, and the sites are called **bxiv0**, **bxiv1** up to **bxiv9**. These correspond to device drivers **/dev/bxiv0** onwards.

K-Par

In a Sun386 installation it is possible to use any Inmos B008 or compatible PC board (if it supports DMA), and access this board through the K-Par device drivers that are shipped with the Sun386 version of Helios. It is possible to plug two of these boards into one Sun giving two sites, called **imb0** and **imb1** and referring to the device drivers **/dev/imb0** and **/dev/imb1**.

Archipel Volvox-1/S

The Volvox board is a link adapter for the SBus based Sun SPARCstation workstations. The site is named **vxv0**, corresponding to the device **/dev/vxv0**.

8.6.5 Which configuration do I need ?

One of the main problems with producing a flexible system is that the user must spend a considerable amount of time deciding how to configure it all. This subsection describes some typical systems, and the configuration that is required for each. In the I/O server configuration file **host.con**, the relevant entries are **box** and **site**. Possible entries for box are:

```
box     = NTP1000
box     = MCP1000
box     = B011
box     = B014
box     = IMB
box     = VOLVOX
box     = remote
```

One of the first six entries should be used when the I/O server runs on the same Sun as the Transputer boards. In that case the I/O server needs to interact with the device driver or the actual hardware, so it needs to know what the hardware is. The last entry is used when the I/O server goes through the link daemon hydra. In that case the I/O server does not need to know what the actual hardware is, since those details are taken care off by the link daemon. If there is more than one site then the **host.con** file may specify the particular site to use, irrespective of whether the I/O server goes through the link daemon or interacts with the hardware directly.

```
site = 2
```

If a particular site is specified then the I/O server will try to access only that site. If the site is already in use then a suitable error message will be generated and the I/O server will exit. If no particular site is specified then the I/O server will search all possible sites to see if any of them is free, and it will use that site. In the **host.con** file sites are numbered from zero onwards. For example, with the Transtech MCP1000 board site number 0 is **nap0**, site 1 is **nap1**, and so on. If the installation needs to use the link daemon hydra (not all installations do) then there should be a configuration file **hydra.con** containing an entry defining the Transputer hardware. See the description of **hydra.con** below. For example:

```
box = imb
```

It is not legal to specify `box = remote` in the **hydra.con** file because the link daemon always interacts with the hardware, not with another remote daemon. The **hydra.con** file contains a number of other options. First it may be desirable to restrict the link daemon to only certain sites, leaving the other sites inaccessible over the ethernet. The system administrator could use this option to ensure that he or she can always use one site, by accessing that site directly. Alternatively all sites may be made accessible. The **hydra.con** options controlling this are:

```
all_sites
#imb0
imb1
```

If the **all_sites** option is given then all the processor sites can be accessed through the link daemon. However, if the option is disabled then only those sites listed in the configuration file are accessible. In the example this would mean that site **imb0** could not be accessed through the link daemon, but site **imb1** could. Note that a site accessible through the link daemon but not currently in use is still available for other software. For example, the link daemon may be set up to allow remote access to site 3 amongst others, but nobody is currently using this site through the daemon. Then any user can access the site directly, for example to run the I/O server without going through the link daemon. Attempts to access a site through the link daemon when another piece of software is using it will fail, and a suitable error message will be produced. As a general rule, there will be only one copy of the **hydra.con** configuration file and this will be controlled by the system administrator. Every user may have her or his own copy of the **host.con** configuration file, although the system administrator should keep a master copy. Using the options above, various different configurations can now be considered.

One Sun with one site

If the installation involves just one Sun not attached to an ethernet network, and the network processor board has just one site, then only the I/O server needs to be configured. This is the most basic installation as shown in Figure 8.9.

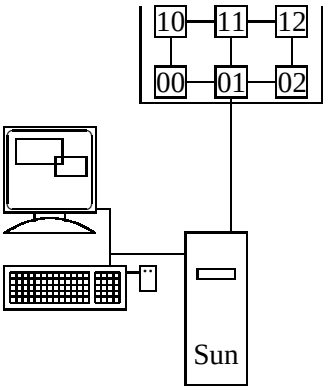


Figure **8.9** One Sun with one site

The **host.con** file should specify the actual hardware, for example:

```
box = b011
```

There is no reason for specifying a site because there is only one possible site. There should not be a **hydra.con** file since there is no point in running the link daemon if no remote I/O servers can access it over the ethernet.

One Sun with identical multiple sites

Again assume that the Sun is not attached to an ethernet. However, this time the processor network board contains multiple sites and each site has exactly one processor. Since there is no ethernet there is no point in running the link daemon, so the **hydra.con** file can be discarded. The **host.con** file should specify the actual hardware. There is no point in specifying a particular site because all sites are equivalent, so this option is usually commented out.

```
box = mcp1000
# site = 2
```

In a network like this typically one user would have the main Sun display and a full windowing system, and the other users would have dumb terminals plugged into the Sun's serial ports. This is illustrated in Figure 8.10.

Much the same effect can be achieved by having the Sun and several PCs attached to the ethernet. The PCs are used for telnet sessions to the Sun, and the users can then run the I/O server on the Sun. The various resources of the PC are not available. Another similar system would involve a number of different Unix workstations, with the users gaining access to the Sun through **rlogin** and running the I/O server on the Sun. Again only the Sun's resources will be available, not the resources of the other Unix workstations.

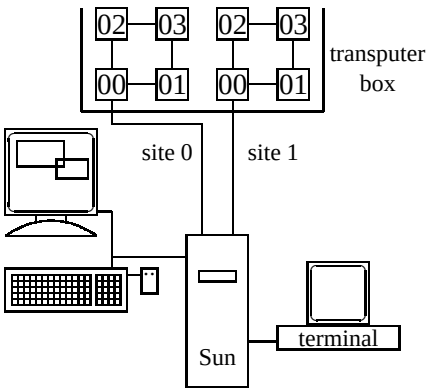


Figure **8.10** Sun and terminals

One Sun with varied multiple sites

A small variation on the above configuration would have different processor networks attached to the different sites. Site zero might have 32 processors, site one only eight processors, and sites two and three only four processors each.

In this case the user will want to specify a particular site when the I/O server is started up. For example, the **host.con** file might contain the entries:

```
box   = b014
site  = 2
```

Alternatively the site could be specified on the command line, for example:

```
server "+site=2"
```

using the I/O server's + command line option to add a line to the configuration file. This technique is a fairly quick way of finding a free site.

Many Suns with many sites

A very different configuration would involve the processor network board plugged into a file server, with all I/O servers running remotely over the ethernet. In this case all I/O servers would need to access a remote link, using the **host.con** entry set out below:

```
box = remote
```

In addition it is now necessary to start up the link daemon hydra. This needs to know what hardware is plugged into the Sun, so this information has to be put into the **hydra.con** file. In the configuration specified all sites must be accessible remotely. Hence the **hydra.con** file should contain the following entries.

```
box = mcp1000
all_sites
```

If all the sites are equivalent then there is no point in specifying a particular site in the file **host.con**. If the sites differ in various ways then it may be desirable to specify one site, either in the **host.con** file or on the command line, as described above.

A complicated configuration

The configuration starts to get complicated when a mixture of the option systems is desired. For example, there are a considerable number of Suns on the network and any one of these may be used to run the I/O server. One of the Suns is equipped with processor network hardware providing a number of sites, and users may wish to run the I/O server on this Sun for maximum performance. There are also various PCs running Telnet sessions connected to the Sun with the processor network, or to other Suns, and the I/O server must cope with these as well. Such a setup is shown in Figure 8.11.

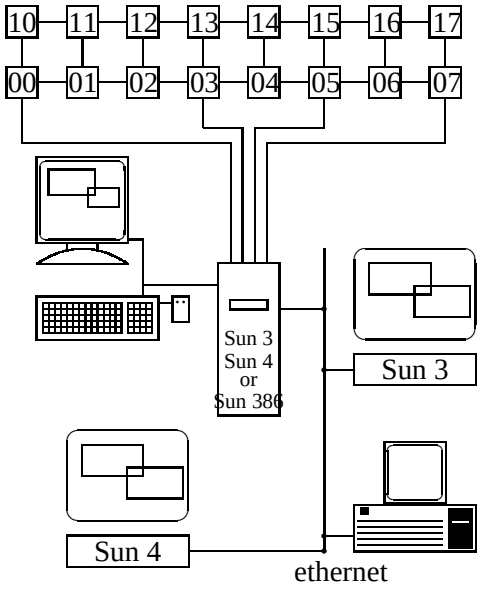


Figure **8.11** A complicated configuration

First, given such a configuration it will always be necessary to run the link daemon hydra. The **hydra.con** file will need to specify the hardware, and which sites can be accessed remotely. Unless there is a very good reason otherwise, it is usual to allow access to all sites. Hence the **hydra.con** file would typically contain the following entries.

```
box = imb
all_sites
```

If a user needs to access a site over the ethernet, through the link daemon, then something like the following **host.con** entries should be used.

```
box  = remote
site = 1
```

This assumes that the various sites are attached to different hardware, which is quite likely in this setup, and that the user prefers to access the hardware on site 1. If a user is currently running on the Sun with the processor network then it is usually desirable to

avoid going through the link daemon because of the extra communication overheads. Hence the **host.con** entries would be:

```
box  = imb
site = 0
```

8.6.6 Other host.con link I/O options

There are a number of **host.con** I/O server options for interacting with the link. These are shown below:

```
box                = IMB
#site              = 0
#box = remote
#family_name       = AF_UNIX
#socket_name       = /tmp/hydra
family_name        = AF_INET
hydra_host         = Molly
connection_retries = 10
```

1. The **box** field should specify either one of the types of supported hardware as described earlier, or **remote**. The hardware can be specified if the I/O server runs on the Sun with the processor network attached. Otherwise **remote** should be used, indicating that the I/O server should go through the link daemon hydra.

2. The **site** variable can be used to specify the particular site to access, irrespective of whether the I/O server is accessing the hardware directly or going through the link daemon. If the variable is not defined then the I/O server will search for any free site.

3. The **family_name** option is only used when going through the link daemon, and should be either **AF_UNIX** or **AF_INET**. The Unix family is used only for debugging purposes. Please see the section 8.6.2 on hydra for more details.

4. The **socket_name** variable is only used when accessing the link daemon through Unix sockets, for debugging purposes. It should match the entry in the **hydra.con** configuration file.

5. The **Hydra_host** option is used when accessing the link daemon through Internet sockets. It should match the entry in the **hydra.con** configuration file.

6. With the option **connection_retries**, the link daemon may reject new connections if it is very busy, because several people are trying to connect and boot at the same time. In that case the I/O server will try to connect several times, at five second intervals. The number of retries is controlled by this entry in the **host.con** file.

8.6.7 The windowing interface

The Sun I/O server provides two different windowing interfaces. The choice of windowing interface depends on the current **TERM** environment variable, which must be

defined before the I/O server is run. The Sun's **printenv** command can be used to find out the current value.

If the **TERM** environment variable is set to **sun**, **sun-cmd**, or any other string with **sun** as the first three characters, then the I/O server assumes it is running under SunView and it will exploit the facilities of SunView including multiple real windows and pop-up menus. Full details are given below.

If the **TERM** environment variable is set to anything else then the I/O server assumes it is running on a dumb terminal of some sort. This could be a traditional terminal plugged into a serial port on the Sun, or a PC running telnet. The I/O server will use the **Server_windows** pseudo windowing system, with a hot key switching mechanism.

The I/O server consults the Sun's **termcap** database for some of its operations. The user may need to understand some of the workings of this database before reading the subsections below, by consulting the appropriate Sun documentation.

Window output

When using SunView or a dumb terminal the I/O server will consult the termcap database called **/etc/termcap** to work out how to drive the screen, and in the case of dumb terminals to find out how big the screen is. This means that the **/window** server can accept standard Helios output escape sequences and convert these to the machine-specific ones to drive the actual output. If the termcap database is incorrect the results are likely to be very confusing.

The following termcap database entries are used:

1. **bl** to ring the bell. If this entry does not exist the I/O server will send the ASCII bell character 0x07.

2. **cl** to clear the screen.

3. **ce** to clear to the end of the current line. If the termcap database does not have this entry then the I/O server will emulate it by writing the required number of space characters.

4. **cm** to move the cursor to a particular location on the screen.

5. **am** to determine whether or not the terminal wraps. The Helios standard for screen output specifies that screens do not wrap, so if the actual terminal does wrap the I/O server has to take care when writing data in the last column.

6. **mr** and **me** to switch output from normal to inverse video and vice versa. If these two options are not defined then the I/O server will substitute the **so** and **se** options instead.

7. **ro** and **co** are used with dumb terminals only, to determine the screen size. If the I/O server is to work correctly then these two entries must be defined correctly.

In general if the Helios screen output is confused one way or the other, this is probably because the termcap entry is not correct. The user will have to work out a correct termcap entry, to be installed by the system administrator.

Window input

In addition to screen output, the termcap database is also consulted to find out what data the terminal generates when particular keys are pressed. This data is then converted to the proper Helios sequence. For example, a typical termcap entry might specify that the terminal generates the sequence **E**[**215z** when the up cursor key is pressed. Whenever the I/O server detects this sequence in a window it will translate it to the correct Helios sequence for that key, byte 0x9B followed by the letter 'A'. The I/O server will test for the following entries in the termcap database, to perform the necessary conversions.

Termcap	Helios
`k1-k9`	Function keys 1 to 9
`k;`	Function key 10
`&8`	Undo key
`@7`	End key
`kI`	Insert key
`kN`	PageDown key
`kP`	PageUp key
`kh`	Home key
`kd`	Down-arrow key
`ku`	Up-arrow key
`kr`	Right-arrow key
`kl`	Left-arrow key
`%1`	Help key

As with the output sequences, if any of the termcap entries are not defined correctly then the user has to correct them.

SunView

When the I/O server is run under SunView it makes use of the facilities provided by that program. In particular, every window produced under Helios will result in a separate window on the Sun's display. The I/O server has its own window which should look something like Figure 8.12.

At the top of a window is a label line giving the name of the window. Below this is a control box for the I/O server, with four buttons, a cycle to control the error logger, and a pop-up menu to control the debugging options. These can be activated as follows.

1. Clicking on **Reboot** (positioning the mouse cursor over the button and pressing the left mouse button) will cause the I/O server to reboot.

2. Clicking on **Debugger** will cause the I/O server to reset the root processor and enter the built-in debugger. The debug session will happen inside the I/O server's window.

3. Clicking on **Status** will cause the I/O server to display the message **Server alive and well.** in the I/O server's own window.

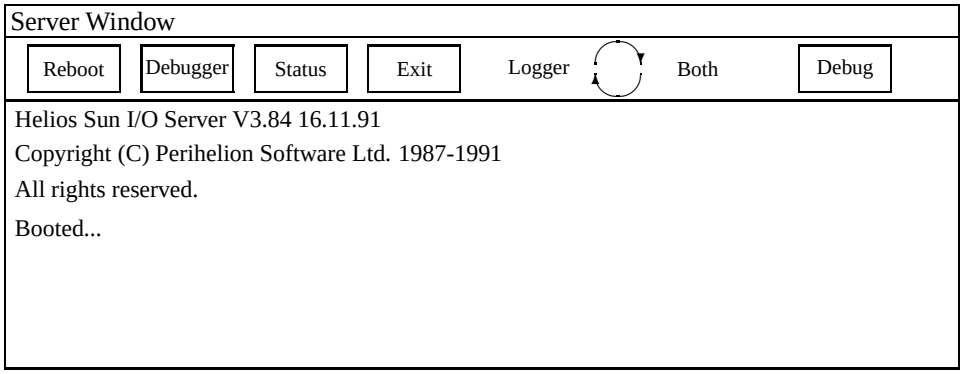


Figure **8.12** The I/O server's window

4. Clicking on **Exit** will cause the I/O server to exit, closing all open files in the process.

5. Clicking anywhere in the vicinity of the logger cycle should switch the error logging destination between screen-only, file-only, and both screen and file.

6. Clicking the left mouse button on **Debug** will activate all debugging options if none are currently enabled, or it will disable any debugging options that are currently enabled. This is equivalent to the **ALL** debugging option.

7. Clicking the right mouse button on **Debug** will produce a pop-up menu listing all debugging options. If any options are currently enabled then these will be highlighted. If the right mouse button is released when it is on top of one of the menu options, then that option is toggled. If the right mouse button is released when outside the menu then this has no effect.

Below the control box is an ordinary text window. When the I/O server writes to the window the text will appear here. Any data typed into this window will be read by the I/O server. Ordinary windows produced by Helios are similar, but do not have the control box.

In addition the windows displayed by the I/O server obey the conventions of SunView. Pressing the middle mouse button when the mouse cursor is positioned over the window's title bar allows the window to be moved. Pressing the right mouse button will pop up a standard SunView menu allowing the window to be moved, resized, iconified, and so on. Care has to be taken when resizing a window, because most applications do not check regularly to see if the window has been resized. For example, when using the MicroEmacs editor it is necessary to exit the editor, resize the window, and start the editor again.

The current implementation of SunView uses up a considerable number of Unix file descriptors for every window, and since a single Sun program like the I/O server can have only 64 file descriptors these are considered a scare resource. Hence the I/O server does not produce the SunView windowing itself. Instead it forks off another program to do the windowing, and it interacts with that program using pipes. By default the I/O server will pick up the correct window program from the Helios directory, either **serverwindow.sun4**, **serverwindow.sun3** or **serverwindow.sun386**. This can be overwritten by an option in the **host.con** configuration file.

```
serverwindow = /usr/local/bin/serverwindow.sun386
```

This option is provided mainly for debugging purposes. When running under SunView there is no way of performing graphical operations in the Helios network that produce output on the Sun's display. The only facility that is provided is text windows.

Dumb terminals

When the I/O server runs on a dumb terminal it implements a pseudo windowing system with a hot key mechanism to switch between windows. This gives an environment similar to that of the PC I/O server. However, there are some difficulties. On a PC there is no problem specifying an obscure key combination such as control-shift-F10 to perform an action such as reboot. It is most unlikely that any Helios applications will be interested in such a combination. Dumb terminals do not offer the same flexibility. It is not generally possible to detect when both shift and control keys are down, and the terminal may not even be equipped with function keys. Hence the I/O server needs to be told which keys or key sequences to use for operations such as rebooting, switching windows, or toggling debugging options, and the information is provided by the **host.con** configuration file.

First, the user has to specify one particular key as the hot key or main escape sequence. All operations can be performed by pressing this hot key, and then one other key. For example, the user may specify that function key one is the hot key: pressing function key 1, then the key 'o', would toggle the **open** debugging option; pressing function key 1, then the key '1', would switch to the next window. The following sequences are defined:

Key Sequence	Operation
hot key '1'	Switch to next window
hot key '2'	Switch to previous window
hot key '3'	Refresh current window
hot key '7'	Enter built-in debugger
hot key '8'	I/O server status request
hot key '9'	I/O server exit
hot key '0'	Reboot I/O server
hot key 'a'	Toggle debugging option ALL
hot key 'b'	Toggle debugging option BOOT
...	
hot key 'z'	Debugging option Z

There are two ways to specify the hot key in the **host.con** file. First, it is possible to specify the termcap name for the key to be used. For example, the termcap name for function key 1 is **k1**, so the following **host.con** entry would make function key 1 the hot key.

```
escape_sequence = k1
```

Sometimes a particular key is not defined in the termcap database. In this case it is still possible to define that key as the hot key, by specifying exactly what bytes are produced by that key. For example,

```
escape_sequence = #\E[S
```

The hash character is used to indicate absolute data `\E[S`, rather than a termcap name. The example defines a key that produces three bytes, an escape character, open-square-bracket, and the letter S. The syntax used is the same as that for the termcap database.

1. `\E` is the escape character, 0x1B.
2. `^Q` specifies control-Q or 0x11, (the caret character should be followed by a single upper-case letter and the corresponding control character is produced).
3. `\0123` is the octal number 0123, or hex 0x53. Any three octal digits starting with a backslash produces the appropriate octal number, provided the number is less than 256.
4. `\n`, `\r`, `\t`, `\b`, `\f`, generate linefeed, carriage return, tab, backspace, and formfeed respectively, as per the C syntax.
5. `\\` generates the backslash character, 0x5C.
6. `\^` generates the caret character, 0x5E.
7. Any other character produces itself.

For example, consider the following possible **host.con** entry:

```
escape_sequence = #\E^Q\012p\n
```

This specifies that the hot key (or the sequence of hot keys) produces the following bytes: 0x1B, escape; 0x11, control Q; 0x0A, octal 012; 0x70, the letter p; and 0x0A, linefeed. Most terminals should allow for a rather simpler escape sequence.

Some of the special operations such as switching windows are particularly useful. These can be invoked by a two-key sequence, the hot key followed by another (specified) key, but often it is desirable to allocate some additional special keys for these operations. The following **host.con** entries could be used to map the main operations onto the function keys.

```
switch_forwards_key  = k2
switch_backwards_key = k3
refresh_key          = k4
debugger_key         = k5
status_key           = k6
exit_key             = k7
reboot_key           = k8
```

The same syntax is used as for **escape_sequence**. In the example the termcap names for the various function keys are used, but absolute character sequences can be specified by using the hash character.

If a key or key sequence is used by the I/O server then it cannot be read from Helios. For example, if the user has specified the cursor keys for various operations then Helios has no way of reading the cursor keys.

8.6.8 Background operation

The Sun I/O server can be run in the background when using SunView to give multiple windows. If a dumb terminal is in use then the I/O server needs full control over the screen and keyboard while it is running, so it cannot be run in the background.

8.6.9 File I/O

The Sun I/O server provides two file servers: **/helios** and **/files**. **/helios** is the standard Helios top-level directory, containing the main Helios binaries, include files and so on. It is mapped onto some directory in the real filing system by using the **helios_directory** variable in the **host.con** configuration file. In most installations **/helios** will contain the following subdirectories:

1. **bin**: binaries for the main Helios commands.
2. **lib**: binary files not accessed directly by users, but accessed indirectly by other programs such as the linker.
3. **include**: the C header files.
4. **tmp**: temporary storage space for applications.
5. **etc**: the system text resource files.
6. **local**: a subdirectory for installation-specific files.

Problems arise in a multi-user environment with perhaps four sites and four simultaneous users. Suppose that the four sites are attached to four separate Helios networks with 32, 16, 8 and 4 processors respectively. Each site needs a different configuration, in particular a different resource map, when booting up. Essentially this means that every site needs a different copy of the **/etc** subdirectory. The four sites can share the **bin**, **lib**, **include**, and **local** directories because in general these do not depend in any way on the site being used.

One solution to this problem is simply to give every site its own copy of the **/helios** directory, but this is inefficient in disc usage because most of the files can be shared safely. An alternative approach has been taken. Suppose that the copy of Helios supports four sites. In that case the **/helios** server will contain the following subdirectories: **bin**, **lib**, **include**, and **local** as before. There will also be subdirectories **etc0**, **etc1**, **etc2** and **etc3**, four separate copies of the **/etc** subdirectory for the four different sites.

When an application under Helios tries to access the directory **/helios/etc**, the I/O server automatically modifies this to correspond to the site being used. For example, if the user is connected to site 2 then the actual directory being accessed is **/helios/etc2**.

The result is that there is a single Helios directory which can be shared by all users simultaneously, without any worries about where site specific information should be held.

The other file server, **/files**, provides access to the root directory of the host filing system. For example, the Helios file **/IO/files/usr/games/rain** corresponds to the file called **/usr/games/rain** on the Sun. This file server allows access to any object within the host filing system including the devices in the **/dev** directory.

8.6.10 The error logger

The Sun I/O server implements the standard error logger device.

8.6.11 The clock

The Sun I/O server implements the standard clock device. It is not possible to write this clock device in the Sun implementation, so Helios applications cannot change the time and date maintained by the I/O processor.

Chapter 9

The Kernel

The Kernel is the foundation upon which the rest of the system is based. To all client programs it appears to be a Shared library like any other.

The major subsystems in the Kernel are described below. In addition to these the Kernel provides a number of additional facilities which will not be described here.

9.1 Kernel data structures

The current state of the Kernel is stored in, or referenced by, two global data structures. These are the **Root** structure and the **Config** structure.

9.1.1 The root structure

The **Root** structure is where the Kernel stores all its static data. Fields of the root structure reference all the major data structures of the system. A pointer to the root structure may be obtained using **GetRoot()**. The data structures to which the Root Structure prints are described later in this chapter. The Root Structure fields are:

PortTable Pointer to port base table.

PTSize Number of slots in port base table.

PTFreeq Descriptor of first Port Table Entry (PTE) in free queue.

Links Pointer to a NULL terminated table of pointers to **LinkInfo** structures.

SysPool Memory pool to which all memory allocated by the Kernel is transferred.

FreePool Pointer to main free memory pool.

Incarnation Incarnation number (unused).

BufferPool Cache of free message buffers.

BuffPoolSize Number of **BufferPool** buffers currently in use.

LoadAverage Low priority process load average. This is approximately the average number of microseconds processing time that each running process has consumed, averaged over the last 3 seconds. On a busy Transputer this will be between 1500 and 2000.

Latency The number of microseconds it would take a high priority process to start execution after being scheduled.

TraceVec Pointer to a 4K trace vector beyond the end of the main free pool.

EventList List of **Event** structures which have been submitted through **SetEvent**.

EventCount Count of number of events seen since bootup.

Time Current system time in seconds since 00:00:00 on 1 Jan 1970.

FastPool Pool for carriers of free fast RAM areas.

MaxLatency The Maximum value of **Latency** seen so far.

IODebugLock Serialisation lock for **IOdebug** operations.

MachineType Processor type code.

BufferCount Number of bytes of memory used by the Kernel in message buffers.

MaxBuffers Maximum value of **BufferCount** seen so far.

Timer The value of the system timer the last time that **Time** was updated. Hence this can be as much as one second out.

Errors Number of processor errors seen so far (Transputers only).

LocalMsgs Total size of all messages passed between local message ports. This includes messages which have been buffered and delivered later.

BufferedMsgs Total size of messages which have had to be buffered pending later delivery.

Flags System flags:

- **rootnode** This was the first processor booted in the system.
- **special** This is a special version of the Nucleus (it has a built-in file server).
- **ROM** This Nucleus is ROM-resident.
- **xoffed** The processor has sent an Xoff link protocol byte through all its active links.

LoaderPool Pointer to the pool into which the Loader allocates all code and libraries.

Configuration Pointer to **Config** structure.

ErrorCodes Array of Kernel error codes. This avoids having these codes in the code as constants. Only ten codes are used, but they are used in several places each. This gives as sizable saving in Kernel size.

IODebugPort If this port is not **NullPort** then IOdebug messages are delivered here rather than to the link with the **debug** bit set.

GCControl Controls the Kernel port garbage collector. The bytes of this word are interpreted as follows:

- **0** Enable garbage collector only if this byte is non-zero.
- **1** Increment the **Age** field of inactive ports once every this number of seconds.
- **2** Free any port if its **Age** field reaches this value.

NThreads Current number of executing threads created through the Kernel. In the case of Transputer versions of Helios, this does **not** count threads created directly by Transputer instructions. Threads must also terminate by calling the Kernel **StopProcess** routine for this count to be decremented.

MaxThreads Maximum value of **NThreads** seen.

9.1.2 The configuration structure

The Config structure is passed to the Kernel by the machine-specific bootstrap system. This structure defines the initial configuration of the processor. A pointer to this structure can be obtained using **GetConfig()**.

The fields of this structure are:

PortTabSize Number of slots in base port table (unused).

Incarnation What booter believes our incarnation is (unused).

LoadBase Address at which system was loaded.

ImageSize Size of system image.

Date Initial value for root **Time** field.

FirstProg Offset of initial program in system image. If zero a processor dependent default value is used.

MemSize If non-zero this is the size of the available RAM, which the Kernel accepts unconditionally. If zero, the Kernel will try to work out the size of memory for itself.

Flags Initial value of root **Flags** field. In particular the **rootnode**, **special** and **ROM** flags should be set here.

MyName The offset from this word to the name of this processor, stored after all other fields in this structure.

ParentName	A similar offset to the name of this processor's booter.
NLinks	Number of links this processor has.
LinkConf[NLinks]	A **LinkConf** structure for each link.
Names[...]	Processor and Parent names stored at the end of this structure.

9.2 Message passing

The primary activity of the Kernel is the passing of messages between tasks. This is designed to be efficient and independent of the locations of the source and destination threads in the processor network.

9.2.1 Message ports

Messages are passed to **ports** which are represented in all programs by a **port descriptor**. A port descriptor is a 32-bit value which indexes into a Kernel table of ports. The descriptor also contains an 8-bit cycle field which must match an equivalent field in the port table entry. This allows the Kernel to distinguish between a current descriptor for that table slot, and a descriptor for a previous occupant of that slot.

The port table is a two-level structure consisting of a base table of pointers to arrays of Port Table Entries (or **PTE**s). A port descriptor contains two 8-bit indexes, one selects the pointer in the base table and the second the entry in the referenced PTE array. A port table entry consists of the following fields:

Type	This may be **free**, **local**, **surrogate**, **trail**, or **permanent**; these last three are essentially equivalent and will all be referred to as **surrogate** ports.
Cycle	This must match the Cycle field in the descriptor.
Age	This records the length of time since this port was last used.
TxQueue	For local ports a queue of waiting transmission processes.
RxQueue	For local ports a queue of waiting reception processes.
Descriptor	For surrogate ports (described below), the descriptor for which this is a surrogate (this field overlays **TxQueue**).
Link	For surrogate ports, the link through which **Descriptor** is valid.

Local ports are used as rendezvous points for message passing between threads on the same processor. When a thread attempts to send a message to a local port which has an empty **RxQueue**, or a non-empty **TxQueue** it is added to the **TxQueue** and suspended. Similarly an attempt to get a message from a port which has an empty **TxQueue** or a non-empty **RxQueue** will result in the thread being added to the **RxQueue**.

A surrogate port is a local representative for a port on another processor. Messages may only be sent to a surrogate port, not received.

A user program may only create local ports, surrogate ports are created only by the Kernel in circumstances described later. However a user program may destroy both local and surrogate ports, but only those which it has created, or which the Kernel has created on its behalf.

Ports are garbage collected. If a particular message port has not been used for a long time, it will be destroyed by the Kernel to save port table space. The current default garbage collection time is 4 hours 15 minutes. This feature may be disabled or altered with the **GCControl** field in the Root structure.

9.2.2 Message structure

Helios messages are divided into three parts, a header, an optional control vector and an optional data vector.

The header contains the following fields:

DataSize Size of data vector in bytes in the range 0 to 65535.

ContSize Size of control vector in 32-bit words in the range 0 to 255.

Flags Flag bits:

- **preserve** Preserve current route.
- **exception** Kernel exception message.
- **sacrifice** Message may be destroyed by Kernel.
- **bytesex** Originating processor byte order (0 = lsb first, 1 = msb first).

Dest Descriptor for destination port.

Reply Optional descriptor for the reply port.

FnRc Function or Return code.

Normally the control vector contains fixed-sized, word-aligned data items while the data vector contains variable-sized data items, indexed by the control vector entries.

A message is described and manipulated by means of a **message control block** or **MCB**. An MCB contains the following fields:

MsgHdr A message header as defined above.

Timeout A timeout in microseconds.

Control A pointer to the control vector in memory.

Data A pointer to the data vector in memory.

For transmission all the fields of the header must be initialised to an appropriate value, and the **Control** and **Data** fields must point to the vectors to be sent. The timeout may either be a positive timeout value, or -1 which is interpreted as infinite.

For reception only the **Dest** field of the header, the vector pointers, and the timeout need to be initialised. The header will be overwritten with the header of the received message. Care must be taken that the vectors point to memory areas large enough to accept the incoming message, as this is not checked.

9.2.3 Message passing functions

There are three main message passing functions: **PutMsg**, **GetMsg**, and **XchMsg**. The first two take a single argument of a pointer to an MCB and either deliver a message to, or receive a message from the port whose descriptor is in the message header destination field.

XchMsg takes two MCB pointers as arguments and attempts to perform a request/reply exchange. The first MCB argument describes the request message and the second the reply message. If only one MCB is given it is used for both messages. This function will retry the interaction in the face of recoverable errors, but will return if a more serious error occurs. Thus it constitutes a simple Remote Procedure Call (RPC) mechanism. On return from **XchMsg** the caller will know either that the destination was unreachable, or that the request was delivered and a reply returned at least once.

In addition to PutMsg and GetMsg, the routines **PutReady** and **GetReady** test for whether a port is ready for transmission or reception respectively.

The **MultiWait** function is used to wait for a message from any one of a set of ports. Its arguments are a pointer to an MCB and an array of port descriptors. The function tests each of the ports in the array in turn for readiness. If one is found to be ready the message is received from that port and its index in the port array is returned. Otherwise the calling process is added to the **RxQueue** of each port in the list. The first port to receive a message will restart the process, which will then detach itself from the remaining port queues and return the index of the receiving port. The **Timeout** is the only field used in the MCB. On return the MCB will be filled in with the received message as with **GetMsg**.

The precise semantics of message passing are deliberately undefined to allow the widest range of possible implementations. When calling **PutMsg** a program must be aware that it may be suspended until the message is delivered but should not assume that it has been delivered if **PutMsg** returns immediately. Programs should also attempt to receive messages as quickly as possible, otherwise they may be lost. This style of ‘eager-reader’ programming is a well known technique for ensuring deadlock free message passing.

9.2.4 Inter-processor message passing

A message is sent from a source processor to a port on some destination processor by passing it from processor to processor through their links. The surrogate ports play an important part in achieving this.

Message transmission

When **PutMsg** is called, the destination port is inspected. If it is a local port the message is delivered, or the process queued as described above. Otherwise the port is a surrogate, and contains a link number plus the descriptor of a port which is valid through that link. The original destination descriptor in the message header is replaced with the descriptor stored in the port. **PutMsg** now queues for access to the link. Once control of the link has been obtained, the message is transmitted through the link in the following order: header, control vector, data vector. This is preceded by a link protocol byte indicating that a message is being sent (see section 9.3.2 for a full description of the link protocol). Once the message has been transferred through the link, the **PutMsg** call returns. However, the message may not have yet reached its destination, so a successful return from **PutMsg** must not be interpreted as an indication that the message has been delivered.

Link guardians

Attached to each input link channel is a **Link Guardian** process. In addition to operating the link protocol, it is responsible for delivering messages to their destination, or forwarding them through another link. When an incoming message is detected, its header is received into a local buffer and inspected. The destination port descriptor is mapped to its associated port table entry and validated. If a **Reply** descriptor is present in the message, a new surrogate port is created. The original descriptor and the current link number are stored in the surrogate port, and a descriptor for this new port is put into the message header **Reply** field in place of the original.

The type of the destination port is now examined. If the type is local then the message has reached its destination processor. The state of the port's queues is examined and if it is ready to receive a message immediately (that is, **TxQueue** is empty and **RxQueue** is not empty) then the Link Guardian copies the header into the waiting MCB, and transfers any control and data vectors directly from the link into the waiting buffers.

If the port is a surrogate then the message must be forwarded to another processor through some other link. If the destination link is in use then the message must be buffered, otherwise the message can be transferred directly. If the link is free the destination descriptor is replaced by the descriptor stored in the port and the header is immediately retransmitted. If the remainder of the message (control plus data vectors combined) is less than 65 bytes it is received in one operation and retransmitted through the destination link. If the remainder is less than 257 bytes then it is received and retransmitted sequentially in 64-byte chunks.

If the message is greater than 256 bytes long, the Link Guardian enters double buffer mode. In this case it awakens a double buffer process associated with the destination link. These two processes now cooperate to transfer the message from the source link to the destination link. While one process is receiving a chunk of data from the source link the other is transmitting a previously received chunk through the destination link. Once both operations have completed, the processes **rendezvous** and change places. This continues until the entire message has been received.

If a destination port or link is not ready to receive the message immediately it

must be buffered for later delivery. The Kernel uses any free system memory for this purpose. To buffer a message, a piece of memory is allocated and the message received into it from the link. Once this has been done a Kernel process is created to call **PutMsg** to deliver the message and free the buffer. As an optimisation, for small messages, the message buffer and process stack are allocated in the same piece of memory. Another optimisation is to maintain a small number of pre-allocated buffers of a fixed default size (currently 1K). These are used as buffers for small messages and as the stacks of the delivery processes.

Port trails

It should be clear from this description that inter-processor message passing is achieved by sending messages along trails of surrogate message ports. The only way that such a trail can be made is as a return path left behind a message as it moves through the system. Long trails are made by a message being routed through a sequence of adjacent shorter trails in turn. Part of the link protocol (*q.v.*) causes adjacent processors to exchange a descriptor each, and it is from these initial simple trails that all others are derived.

Most port trails are short lived, existing only between the time that a message is delivered to its destination and a reply is generated. The default behaviour for messages is to destroy the trail as it passes through it. The **preserve** flag in the message header prevents this, allowing the trail to be used again.

The double buffering, and the sequential single buffering for smaller messages, means that if the links are free, then a message will be spread out across several processors and links. This is essentially a form of wormhole routing and gives a much higher data rate than if each message were stored and forwarded.

There are several reasons for adopting this trail-based routing scheme rather than a processor-id based scheme. Firstly, by avoiding introducing globally unique processor identifiers, running systems can be interconnected without worrying about clashing identifiers. Secondly, the system is capable of being expanded and contracted without having to inform every processor of the changes. Thirdly, the number of processors is not limited by such things as the processor id field size in any data structures. Fourthly, at each processor on a trail, the destination of a message is exactly defined by its destination descriptor and no routing table lookup is required. Finally, only active connections consume resources on intermediate processors.

Error handling

It is not always possible to deliver a message at all. The destination port descriptor may be invalid, the link indicated by a surrogate port may have changed mode, or the processor through that link may have crashed. If a failure is detected during the **PutMsg** call then an appropriate return code is generated. If the failure is detected by a Link Guardian it must attempt to do two things: recover from the error, and inform the sender that the message has been lost. In the case of an invalid descriptor, or a bad link, only the header will have been received. The Link Guardian recovers by receiving and destroying the remainder of the message. If a destination link is not detectably bad, but the processor beyond it has crashed or stopped listening, then any attempt to transmit

through that link will result in a timeout. Since the transmitter on the other end of the source link is running exactly the same timeout, it will fail at approximately the same time. Hence the Link Guardian does not need to take any recovery action, and the message failure will ripple back to the message originator.

When a message is lost or cannot be delivered, the Link Guardian attempts to inform the originator. It can only do this if the lost message contained a reply port. In this case the Link Guardian sends an **Exception** message back along the trail already built. This has two effects: first, it destroys the trail built by the lost message, and second, it informs the originator of the reason for the message failure. This means that the result of a **GetMsg** call may yield a result associated with a previous **PutMsg**. Since these two operations are likely to be linked in an RPC style interaction, this is the correct behaviour. Messages without a reply port are simply destroyed silently since there is no way of finding out where they originated.

Message passing reliability

The Kernel does not guarantee to deliver all messages reliably. On Transputer systems, in the absence of link failures and processor crashes, this results in all messages being delivered reliably. However, since the higher-level protocols are designed to handle message loss, when a link does break or a processor crash, the system can recover from this and continue where it left off. Hence, the lack of message reliability is more a philosophical point of view than a reflection of the true nature of the system.

The only time the Kernel will explicitly destroy messages is in order to resolve any deadlocks in the message passing system. The major cause of such deadlocks is a program failing to receive messages sent to it. In this case the destination processor, followed by processors successively closer to the source of he messages, will fill all available memory with buffered messages and eventually stop listening for more. The Helios servers are designed to accept messages as soon as possible. The standard protocols are designed to avoid the transmission of unexpected messages which may cause congestion. This behaviour is known as ‘eager-reader’ behaviour, and is a common requirement of deadlock free routing systems.

9.3 Links

9.3.1 LinkInfo

The processor links are managed through the **LinkInfo** structures, one per link. The fields of the LinkInfo structure are:

Flags Some flags:

parent	The link through which processor was booted.
ioproc	The processor through this link is a non-Transputer I/O processor.
debug	Send low-level debug messages through this link.
report	Changes in the state of this link will be reported to the Processor Manager.

	stopped The processor through this link has temporarily quenched all traffic.
Mode	Link mode: Null, Dumb or Intelligent.
State	Link state (see below).
Id	Link number.
TxChan	Address of transmit hardware control location.
RxChan	Address of receive hardware control location.
TxUser	Pointer to current transmitting process, if any.
RxUser	Pointer to current receiving process, if any.
MsgsIn	Total data received on this link so far.
MsgsOut	Total data transmitted on this link so far.
TxQueue	Queue of processes waiting for access to transmit on link.
RxId	Current process receiving a message from link, used only during link-to-local port transfers.
TxFunction	Optional link data transmission function. This is used to redirect link traffic to another device and it is also used to support links based on hardware other than Transputer style links.
RxFunction	Optional link data reception function.
Sync	Synchronisation point used by Link Guardian and **KillTask**.
LocalIOCPort	Destination port for system messages from this link. This port is sent across the link in the **Info** link protocol message (see below).
RemoteIOCPort	Port for system messages through this link. This is a surrogate port for the remote processor's **LocalIOCPort**.
Incarnation	Currently unused, intended to distinguish between reboots of the remote processor.
MsgsLost	Number of messages lost or destroyed by this Kernel.
DBInfo	Pointer to a data structure associated with this link's double buffer process.

The link **Mode** determines whether the link is **intelligent** and can be used for message passing, or whether it is a **dumb** link. A link is either mode can be in one of several states. The valid states are:

Null	The link is not connected.

Dumb Not used.

Running Indicates an intelligent link is sending and receiving messages successfully. For a dumb link indicates that the link is allocated.

Timedout Indicates that the Link Guardian has seen a reception timeout and that the link is probably dead.

Crashed Indicates that the link has been active, but is no longer.

Dead Indicates that the link is inactive. For dumb links, indicates that the link is unallocated.

The Kernel will move links between states as necessary according to the link protocol, but links may also be forced into a given state by the **Configure** function.

9.3.2 Link protocol

In addition to passing messages through the links, the Link Guardians maintain a low-level link protocol between themselves. This is to enable the system to detect processor failures, and to retain synchronisation of the link traffic.

The link protocol consists of single byte codes, followed by some type-specific data. The protocol bytes are as follows:

Write Followed by a 32-bit address and a 32-bit value. The Link Guardian accepts but ignores this message.

Read Followed by a 32-bit address. If the address is a special probe value, the Link Guardian returns a word of **Alive** protocol bytes. Otherwise the contents of the addressed word are returned.

Msg Followed by a Helios message as described above.

Null Never sent across the link, this is used to initialise the buffer into which the protocol byte is received. It indicates a timeout to the Link Guardian.

Term Sent by a processor just before it resets itself to put all its neighbours links into Dead state.

Reconfigure Sent when a link is being reconfigured from intelligent to dumb state, causes the equivalent state change on the other end of the link.

Reset On a Transputer, the **Reset** byte is followed by the bytes 0x21 0x2f 0xff 0x03 0x21 0x2f 0xff. This is interpreted by the Link Guardian as an instruction to reset the processor. The same sequence sent to a reset Transputer will boot it with a reset instruction.

Xoff Causes the remote processor to stop sending any more messages through this link.

Xon Restarts messages stopped by a previous Xoff message.

Info Followed by three more **Info** bytes plus an 8-byte info message. This message contains the **LocalIOCPort** descriptor for the sending processor plus a flag indicating whether a reply is needed. Whenever an **Info** message is received, the links **RemoteIOCPort** is re-initialised with the port descriptor received.

Alive Followed by three more **Alive** bytes. This is sent as a response to a special probe **Read** message.

Dead Followed by three more **Dead** bytes. This is part of the idle exchange described below.

Most of these messages are used as part of the idle handshake. This is designed enable a Kernel to remain confident about the state of the processor through a link in the absence of link traffic. If no message has been received through a link for some time the Link Guardian goes through the following sequence.

1. Send a **Write** message to write a word full of **Dead** bytes at a special probe address.
2. Send a **Read** message to read back the word just written. If the response is **Dead** bytes then the link is marked dead and the idle exchange terminated. If the remote processor is running, it will respond with a word of **Alive** bytes.
3. Send an **Info** message indicating that a reply is required.
4. Receive an **Info** message in reply.

This exchange is event driven. The response to each step triggers the next action. The Link Guardian always spawns a worker process to transmit the link protocol messages, leaving itself free to receive messages. Thus both ends of the link may initiate the idle exchange simultaneously. Since the protocol is event driven, it can be entered part-way through. For example, when a processor is first booted, it initiates just the **Info** exchange with its parent.

9.3.3 Dumb link access

When Helios is using a link for message passing, it is not available for direct use by application programs. A link can be made available for direct use with the **Configure** function. The argument to configure is a **LinkConf** structure, which defines the new state of the link:

Id The number of the link to be changed.

Flags The new value for the **LinkInfo Flags** field. Only the state of the **report** and **debug** flags may be changed.

Mode The new link mode, either Dumb or Intelligent. If the link is changing from intelligent to dumb mode, a **Reconfigure** protocol byte is sent through the link to force the far end into dumb mode too.

State The new link state. If the mode of the link is changing from dumb to intelligent this defines the state of the link. When changing from intelligent to dumb mode the link state is forced into Dead state. If the link is not changing mode the state will not be affected.

Once a link is in dumb mode, it can be used for direct access. The Kernel provides a number of functions for this, which will only operate on links in dumb mode. The **AllocLink** function reserves the link for use, this enables different programs to use the same link in a controlled manner. The **FreeLink** function reverses the effect of **AllocLink**. An allocated link is signified by changing its state from Dead to Running.

The functions **LinkIn** and **LinkOut** perform data transfers across the chosen link. These functions will only operate on Dumb mode links in Running state. Each function takes a link number, a buffer and size, and a timeout. The timeout, like that on **PutMsg** and **GetMsg** is used to detect transfer failures rather than provide a timing service. A minimum timeout of two seconds is imposed as a result of the implementation of these functions. An example of direct link usage is given in appendix C.

9.4 Tasks and threads

The primary execution unit under Helios is a **task**. This is the object to which resources such as ports and memory are allocated. Within a task there may exist a number of **Threads**. These all share the resources of the task and may communicate or synchronise with each other through shared memory and semaphores.

9.4.1 Tasks

A task is described to the system by a **task** data structure:

Node In theory used by Kernel to list all tasks. In practice not used, but reserved for possible future use.

Program Pointer to the code (or text) of the program the task is executing.

MemPool task's memory pool. All memory allocated by the task is linked into this pool.

Port Task's initial message port, used to receive the program's environment.

Parent Message port back to this task's creator.

IOCPort Message port for communication with the local Processor Manager.

Flags Flag word. In debugging systems this controls the level of debug output provided by the System library.

ExceptCode Function called on delivery of a hardware exception to this task. This is unused in Transputer versions of Helios.

ExceptData Data passed to **ExceptCode**, no longer used.

HeapBase Base of task's initial heap.

ModTab Pointer to base of task's module table.

TaskEntry Pointer to Processor Manager's controlling data structure for this task.

A new task is created by the **TaskInit** call which takes a pointer to the **task** structure for the new task[1]. **TaskInit** expects the **Program**, **Port** and **MemPool** fields to be initialised. In turn the **HeapBase** and **Modtab** will be initialised by the Kernel. Since the **TaskInit** call will start the task running the remaining field should also be initialised before the call, although failure to do so will not affect the functioning of the Kernel, only the resulting program. The memory layout presented when a task is started is described in chapter 16, *Program representation and calling conventions*.

Normally a task is only terminated at its own request. This is done by **KillTask** which is only called by the Processor Manager. **KillTask** is responsible for suspending all threads belonging to a particular task. To do this it scans the processor timer queues, run queues, port tables and LinkInfo structures in order. It identifies threads which belong to the task by comparing their workspace (stack) pointers to the range of addresses defined by the task's memory pool. If a thread is found it is removed from the queue.

9.4.2 Threads

Threads are the entities which actually execute code. On the Transputer they are supported entirely by the hardware and on other processors a lightweight software scheduler performs the same functions.

Threads are created by the Kernel calls **InitProcess** and **StartProcess** and can halt themselves by calling **StopProcess**. There is no way to suspend or stop a thread externally. On the Transputer, threads may be created with the appropriate machine instructions without needing to inform the Kernel.

Since threads can be created easily without Kernel intervention, and because the number of threads can grow large, the Kernel does not maintain any permanent per-thread data structures. Instead a structure, known as an **Id** structure, is allocated whenever the Kernel needs to suspend a thread. A Id structure has the following fields:

rc Return code. Indicates reason for wakeup.

next Next Id in queue.

tail Tail Id in queue. Only valid in the first Id on the queue.

state Pointer to saved thread machine state.

endtime Time at which this thread should be restarted if still waiting.

mcb For message queues, a pointer to the MCB.

This structure is always allocated on the thread's stack, along with any saved state.

[1] Currently the **task** structure is allocated in the memory space of the Processor Manager, and pointers to it are passed to the Kernel and the new task itself. This violation of memory security will be changed in the future and must not be relied upon.

9.5 Timeout handling

All message operations contain a timeout, and the Kernel protects itself against dead links and crashed processors by applying a timeout to all link transfers. All these timeouts are driven by the timeout process. This process wakes up once every second and scans the port tables and LinkInfo structures for Id's whose endtime is less than the current time. If one is found, it is dequeued and resumed with a timeout return code.

The result of this mechanism is that such timeouts may expire anything up to a second later than the time for which they are set. The cost of maintaining a more accurate timeout mechanism is not justified by the use to which they are put. All such timeouts are used to detect exceptional, usually failure, conditions. These happen rarely and it does not matter that the indication is slightly late. More accurate timing can be achieved by using a second timeout process which executes **Delay** for the required time and then raises an event (possibly by aborting a port, or sending a message).

9.6 Semaphores

Between the threads of a single process there is no need to pass messages since they share the same address space and can share memory. However, it will be necessary to protect this shared memory against concurrent access, and there are other situations in which threads may need to synchronise.

In Helios these requirements are met by the implementation of semaphores. A **Semaphore** structure is allocated in the task's address space and is private to that task. Inter-task semaphores should not be used.

A semaphore may be initialised by **InitSemaphore** to any integer value, either positive or negative. The **Wait** function is defined to return as soon as the semaphore counter may be decremented to yield a non-negative result. The **Signal** function simply increments the semaphore. The **TestWait** function will always return immediately, and will return TRUE if it managed to execute a **Wait** operation on the semaphore, and FALSE otherwise.

The effect of this is that if a semaphore is initialised to 3, then, in the absence of any calls to **Signal**, three calls to **Wait** will be allowed before a thread is blocked. If the semaphore is initialised to 1 then it acts as a serialising mutual exclusion lock. If the semaphore is initialised to -3 then it will block any **Wait** operation until *four* **Signal** operations have been performed. This can be used, for example, to wait for the termination of a number of **Fork**ed threads.

In non-Transputer based versions of Helios there is also a **TimedWait** call. This is essentially a **Wait** call with a timeout. If the call to **Wait** suspends the thread for longer than the timeout value, the thread is rescheduled and the **Wait** will return with an error. The semaphore will not be decremented.

9.7 Memory management

The Kernel's memory management system is optimised for the allocation of large blocks for program use, and for speed in allocating message buffers and Kernel worker

process stacks. User programs normally use the System library **Malloc** and **Free** calls (or language specific memory allocation) which implement a Heap Manager within blocks obtained from the Kernel.

Kernel memory blocks are all allocated to a **Pool**, whether it be a free pool, a system memory pool or a task pool. Tasks may also create their own pools, but care must be taken in how they are used.

A Kernel memory block consists of a **Memory** header structure followed by the memory itself.

Node Link in owning pool's memory list.

Size Size of block including this structure. All memory blocks are a multiple of 16 bytes long, so the least significant four bits of this field are not used. These bits contain the following flags:

- **FwdBit** Allocation state of this block: 1 = allocated, 0 = free.
- **BwdBit** Allocation state of block physically before this one in memory.
- **Fast** Indicates that this is a carrier for a fast ram block.
- **Reloc** Indicates that this is a relocatable block, this is used in some systems to implement a recoverable RAM disc.

Pool Pointer to pool which owns this block.

In addition to this structure, any block marked as free will contain a pointer to its **Memory** header in its last word. This allows the Kernel to locate the headers of both a block's physical neighbours quickly.

Memory blocks are collected together into pools. A pool is described by the **Pool** data structure:

Node List node for collecting pools together, currently unused.

Memory List of **Memory** blocks which constitute this pool.

Blocks Current number of memory blocks in this pool.

Size Total size of blocks currently in this pool.

Max Original size of pool (only useful in free memory pools).

There are essentially two types of memory pool: free and allocated. A free pool contains only blocks marked as free. The blocks in the **Memory** list of the pool are kept in strictly **descending** address order. A newly initialised free pool must describe a contiguous region of memory. The first block on the list is a header-only block placed at the top of the memory region and is marked as allocated. The remainder of the memory is initially described by a single free block whose **BwdBit** flag bit is set. These ensure that the boundaries of the pool are protected by allocated blocks. An allocated pool contains only blocks marked as allocated, and these are placed in the **Memory** list in any order.

The external memory routines, **AllocMem** and **FreeMem** are translated by the Kernel into the more general **Allocate** and **Free** routines. **Allocate** takes a size, and pointers to a source free pool and a destination allocated pool. It starts by adjusting the size to the next highest multiple of 16 and adds the size of the **Memory** header (also 16). It then scans the blocks in the free pool's memory list until it finds a block whose size is greater than or equal to the required size. If the block size is not much larger than the required size, the whole block is allocated. Otherwise the block is split into a piece of the required size, plus the remainder. The required piece is always taken from the top of the block to avoid having to alter the memory list links. It is because of this that free pool blocks are listed in descending order, and Helios allocates memory from the top of RAM. Once a block has been found, its **FwdBit** and its successor's **BwdBit** are set and it is linked into the destination pool. The **Size** and **Blocks** counts are adjusted for each pool.

Free returns a memory block to its free pool. The source pool is obtained from the blocks **Pool** field, and the free pool is passed as an argument. The block is first removed from its current pool and the counts of both pools adjusted accordingly. If the block's **BwdBit** is clear then the address of the predecessor's header is picked up from the word immediately before the current block's header. The two blocks are coalesced by adding the current block's size to that of the predecessor; the predecessor now becomes the current block for the rest of the function. By adding the current block's size to its base address, the header of the successor is found, if this is free the blocks are coalesced; the memory list pointers being re-initialised to point to the new block. If neither of the block's neighbours is free, then its place in the memory list is found by searching. Once inserted into the list, the block's **FwdBit** and its successor's **BwdBit** are cleared.

On the Transputer (and some other processors) a small quantity of fast, on-chip, RAM is available. To avoid occupying valuable space in this memory with system data structures, this is managed by a slightly different mechanism. Regions of fast RAM are described by **Carrier** structures:

Addr Address of fast RAM region.

Size Size of region in bytes.

An initial carrier for the entire fast RAM area is allocated from the system free pool and put into a special fast RAM free pool. Calls to **AllocFast** search the carriers in this pool for a match. If one is found it is transferred to the destination pool, otherwise a new carrier is allocated and a block is split, the new carrier being allocated to the destination pool. Since carriers are marked as such, the fast RAM they represent can be freed by passing them to the normal **FreeMem** routine. This detects carriers and passes them to the internal **FreeFast** routine which returns the fast RAM to the free pool.

9.8 Events

The Kernel provides access to the processor's hardware interrupt handling mechanisms through an event handling mechanism. Attached to each interrupt source is a priority

ordered list of event handling procedures. When the interrupt occurs the event procedures are called in decreasing priority order.

On the Transputer, the only available interrupt is the event line. When this is signalled, a process attached to the processor's event channel is awakened. This process calls all the procedures in the list of **Event** structures which have been passed to the Kernel through **SetEvent**. These procedures will be called in the context of their parent task, but at high priority. For this reason they must avoid being blocked, or consuming more than the minimum CPU time. In general these procedures should simply signal a semaphore, or send a message to a low priority process which will continue with the event processing.

On non-Transputer processors, an additional **Vector** field in the **Event** structure selects one of several possible interrupt sources. When an interrupt occurs, the procedures registered for that interrupt name are called in order of priority until one returns a TRUE result.

Chapter 10

The System libraries

The Helios Nucleus contains four Shared, or Resident libraries. The Kernel is described in chapter 9. The Server library is described in chapter 12, *Writing servers*. The remaining libraries are the System library and the Utility library. These are described below.

The reader is referred to the system headers for the precise details of function prototypes and constant values, and to the Encyclopaedia for a technical description of each function. Some knowledge of these is assumed in the following descriptions.

10.1 The System library

The System library is primarily responsible for presenting a procedural interface to the GSP protocol operations. As such it corresponds approximately to the conventional system call interface of most operating systems. It should be remembered, however, that it is just a library, executing in user mode and storing its data structures in user memory.

10.1.1 System library data structures

The primary data structures manipulated by the System library are **Object** and **Stream** structures. An **Object** corresponds exactly to a GSP context object. A **Stream** corresponds to a GSP direct operation port.

An **Object** is usually created as a result of a **Locate** or **Create** operation and is initialised from the reply message. It can also be created under other circumstances described later. An object contains the following fields:

Node All **Object**s known to the System library are chained together through this field.

Type The object's type from the reply.

Flags The object's flags from the reply plus some System library flags.

Result2 The last error on this **Object**.

FnMod The return code passed back from the server in the reply. This will be bitwise ORed with the function code of all indirect operations which use this **Object** as a context.

Timeout Base value for calculating timeouts.

Reply Local port for all replies to indirect operations using this object.

Access The object's capability from the reply.

Name The object's canonical path name from the reply.

A **Stream** is initialised from the reply to an **Open** operation. It contains the following fields.

Node All **Stream**s known to the System library are chained together through this field.

Type The object's type from the reply.

Flags The object's flags from the reply plus some System library flags.

Result2 The last error on this **Stream**.

FnMod The return code passed back from the server in the reply. This will be bitwise ORed with the function code of all direct operations which are sent through this **Stream**.

Timeout Base value for calculating timeouts.

Reply Local port for all replies to direct operations using this object.

Access The object's capability from the reply.

Pos The current object position pointer, initialised to zero.

Server The direct operation port from the reply.

Mutex A semaphore to protect this **Stream** against concurrent access.

Name The object's canonical path name from the reply.

The first eight fields of these structures are identical, and in some cases may be used interchangeably in some System library functions. Where such a polymorphic function needs to distinguish between the structures, it can do so by examining the **Flags** field.

10.1.2 System library flags

The flags controlled by the System library in the **Flags** field are:

Mode These four bits are a copy of the least significant four bits of the mode passed to **Open**. They are used to define the mode if the **Stream** needs to be reopened.

Remote Indicates whether the server being used is on another processor.

Append This is the same bit as **O_Append** and is saved in case a reopen is necessary.

Application These four bits are reserved for use by the application.

Stream Indicates whether this is an **Object** or **Stream**.

10.1.3 Open modes

In addition to the mode bits defined as part of the GSP **Open** operation, the System library **Open** operation also defines, or examines, mode bits such as:

Append All **Write** operations should be performed at the end of the object. If this bit is set the **Open** function issues a **Seek** operation before returning.

10.1.4 Object and stream manipulation

In addition to the results of **Locate**, **Create** and **Open**, **Objects** and **Streams** may also be created through other System library functions.

The functions **CopyObject** and **CopyStream** each produce a duplicate of their argument. While **Objects** may be copied freely, care must be taken in copying **Streams**. This is because a copy does not contain a fresh direct operation port but shares the port of the original. This means that a **Close** operation from either will invalidate the other. To partially avoid such problems, the **Closeable** flag in the copy is cleared. This means that when a copy is closed it will not generate a **Close** operation to the server, but closing the original will.

The functions **NewObject** and **NewStream** create an Object or Stream from their arguments. The new Object or Stream is given type **Pseudo**, and is not validated against the server. The exception to this is if the mode passed to **NewStream** (actually a full **Flags** field) has the **OpenOnGet** bit set. In this case an **Open** operation will be performed.

A pseudo Object is one which has not been validated with the supporting server, hence there is no guarantee that the server, or the object, still exist, or that the capability is valid. A pseudo Stream is one which has not been opened, so the direct operation port is invalid. These Streams and Objects may be used anywhere that a normal one may be used. The System library detects this and calls either **ReLocate** or **ReOpen** to validate and/or open the object before continuing with the desired operation. A failure in the validation or open operation results in the entire operation failing.

The function **PseudoStream** builds a **Stream** from an **Object** and a set of flags. This is actually a jacket which extracts the name and capability from the Object and calls **NewStream**.

The function **Abort** causes any pending operations on a **Stream** or **Object** to be aborted. This is somewhat drastic because it uses **AbortPort** to detach any waiting **GetMsg** calls from the reply port with a hard error code. The result of this will be to terminate an interaction with a server part way through. With respect to pipes, this function issues a GSP **Abort** operation.

10.1.5 The environment

When a task is first started it has no **Object**s or **Stream**s. For some system tasks and servers this is acceptable. However most tasks need an environment defining their arguments, initial I/O stream etc. This is passed to the new task from its parent through the GSP **SendEnv** protocol. The System library provides an implementation of both the sending and receiving parts of this protocol in the **SendEnv** and **GetEnv** functions.

The **SendEnv** function is given a destination port plus a pointer to an **Environ** structure. This contains the following fields:

Argv A pointer to a NULL terminated array of pointers to strings. By convention the first argument is the name of the program.

Envv A pointer to a NULL terminated array of pointers to strings. By convention these are all of the form **"NAME=value"**.

Objv A pointer to a NULL terminated array of pointers to **Object** structures. Empty slots in the array may be filled with the value **MinInt**.

Strv A pointer to a NULL terminated array of pointers to **Stream** structures. Empty slots in the array may be filled with the value **MinInt**.

There are system conventions with regard to the use of the entries in the **Objv** and **Strv** arrays. The entries in the stream array are referenced by their index, and are expected to correspond to the Unix conventions for streams. Thus **Strv[0]** is the task's standard input, **Strv[1]** is its standard output and **Strv[2]** is its standard error stream. In general **Strv[n]** will be referenced by the POSIX library through file descriptor **n**.

The entries in the **Objv** array are assigned specific purposes and have been given specific names:

Cdir Current directory.

Task Entry in the local Processor Manager.

Code Entry in Loader.

Source Original program source file.

Parent Processor Manager entry for parent task.

Home User's home directory.

Console Control console or window.

CServer Directory of control console, allows new windows to be created on the same device.

Session User's Session Manager entry.

TFM User's Task Force Manager.

TForce If this task is part of a task force, this is the TFM entry for the task force.

When **GetEnv** receives an environment, it expands the marshalled entries in the four arrays of the environment message. The **Argv** and **Envv** entries are converted into direct pointers, and the terminating -1 is converted into a NULL. The object and stream array entries are converted, through **NewObject** and **NewStream** into full **Object** and **Stream** structures. These are all given type **Pseudo**. This means that if a program never accesses a stream or object passed in its environment, then the supporting server will never be contacted, saving time and memory. The only exception to this is if a stream has the **OpenOnGet** flag set, in which case it will be opened in **GetEnv** before the acknowledge message is sent. A symmetric **CloseOnSend** flag causes **SendEnv** to close a stream. This is only done once the acknowledge message has been received. Thus either the sender or the receiver (and briefly both) always has an open connection to the server.

10.1.6 Fault tolerance and recovery

The System library implements the standard GSP recovery strategy. Recoverable errors result in the retry of individual messages or message exchanges. Warnings result in a call to **ReOpen** for **Streams** and a retry with an incremented retry field for **Objects**. More serious errors are reported to the application.

10.1.7 Memory management

The System library provides a memory management package for its own use and for the use of application programs. This package obtains large blocks of memory from the Kernel and implements a heap within them. A feature of this mechanism is that if any Kernel block is emptied it is returned to the system.

All memory blocks allocated by the System library are composed of a **Memb** structure followed by the memory itself. The **Memb** structure contains the following fields:

word Size Size of memory, including header, rounded up to next 8-byte boundary. The least significant bit is set for free blocks.

Memb *Next Next memory block in free list.

Each memory block obtained from the Kernel is headed by a **HeapBlock** structure:

Node Node Link in list of heap blocks.

word Size Total size of available memory in block.

word Free Number of bytes free in block.

Memb *FreeQ List of free blocks.

These heap blocks are chained together in a list, and all free memory blocks within a heap block are chained on the **FreeQ**.

The action of **Malloc** is to first adjust the size to allow for the header and align it to the next multiple of 8 bytes. The heap blocks are then scanned in strict order for one

with sufficient free space, and the free queue of that block scanned on a first-fit basis for a suitable memory block to allocate. If no space can be found in any block then a new heap block is allocated. The size for this heap block is taken from the **HeapSize** field of the program's header. As a special case, if the requested size is more than half the standard heap size the memory is allocated directly from the Kernel. A **Memb** header is placed at its beginning, but the size is negative to mark it as a special block. The function **MemSize** takes account of this in calculating the size of the block.

There are two memory freeing functions, **Free** and **FreeStop**. Both release a memory block back into its containing heap, coalescing it with it neighbours if possible. If the operation results in all the memory in a heap block being freed, the heap block itself is returned to the Kernel. The difference between the two routines is that **FreeStop**, once the memory has been freed, halts the calling thread. This function exits solely so that a thread may free its own stack and halt as a single atomic action.

10.1.8 DES encryption support

The System library contains an implementation of the DES standard encryption algorithm. This is used by the system servers to encrypt all capabilities. The code is a straightforward implementation of the algorithm and as such is not particularly fast. On a 20MHz T800 it takes approximately 8 milliseconds to encrypt a single 64-bit block. This can be reduced to between 4 and 5 milliseconds per cycle if **DES_KeySchedule** is used to pre-generate the key schedule, but this is only true if many blocks are to be encrypted with the same key.

10.2 Utility library

The Utility library collects together a number of useful functions which do not properly belong in any other library. These include some string and memory manipulation functions from the C library, Thread creation, debugging functions and fast RAM access.

10.2.1 C library functions

Because they are used by the system servers and libraries, a number of routines from the C library have been moved into the Nucleus. This avoids the need to include the entire C library into the Nucleus.

The functions in this library are:

longjmp **setjmp**
strcat **strncat**
strcpy **strncpy**
strcmp **strncmp**
strlen **memcpy**
memset

10.2.2 2-D block move

In Transputer versions of Helios, the function **bytblt** provides a procedural interface to the Transputer 2-D block move instructions (see *The Helios Encyclopaedia* for further information).

10.2.3 Thread creation

This library provides a low-level thread creation interface, essentially a veneer on top of the Kernel routines.

The **NewProcess** function allocates a stack, using **Malloc**, and calls **InitProcess**. The result is a pointer to the position of the first argument for the new thread. The functions **RunProcess** and **ExecProcess** start a thread created by **NewProcess** running. The main difference between these routines is that **RunProcess** starts the thread running at the same priority as the caller, while **ExecProcess** starts it at a selected priority. If the caller decides not to start a process, it can call **ZapProcess** to deallocate the stack.

The **Fork** function is the most frequently used interface to these routines. It simply calls **NewProcess** to allocate and initialise a stack, copies the supplied parameters into the new thread's stack frame, and calls **RunProcess** to start it.

As described in chapter 9, *The Kernel*, threads are only able to terminate themselves, they cannot be terminated externally except on task exit. A thread created by **NewProcess** (and hence **Fork**) can terminate by returning from its entry procedure. This returns to code which will deallocate the stack and halt the process (by calling **FreeStop**).

10.2.4 Using fast RAM

Helios provides two mechanisms for using fast, on-chip, RAM on the Transputer and on other processors which provide this facility. These allow the programmer to move either the stack or the code of a procedure to fast RAM.

The **Accelerate** function executes a given procedure using fast RAM for its stack. The RAM to use should have been allocated with a call to **AllocFast**. When **Accelerate** is called, a stack frame is built at the top of the fast RAM area supplied, the arguments to the function are copied across, the current stack is set to the new frame and the function is entered. When the function returns, code is entered which reinstates the original stack and returns from **Accelerate**. Obviously only one thread at a time can call **Accelerate** using any given fast RAM area. The execution time of the called procedure must be such that the cost of transferring to fast RAM is rendered negligible.

The **AccelerateCode** function moves the code of a given function into fast RAM. Because most functions access constants, strings and external procedure calling stubs in a PC-relative way, the entire module containing the procedure must be moved into fast RAM. This means that the procedure must be small and be compiled on its own into a separate module. In C this means that the procedure must appear in a source file of its own. From the supplied procedure pointer **AccelerateCode** locates the module header, allocates sufficient fast RAM, and copies the entire module over. If this is successful, it re-calls the module's initialisation routines to re-install it in the module table

(see the chapter entitled *Program representation and calling conventions* for more details of this). Following this, any invocation of the moved procedure will execute the fast RAM copy, no special action is needed on the part of the callers to do this.

10.2.5 Debugging support

Helios as a whole provides a low-level debugging mechanism which allows simple text messages to be routed to an I/O server from any processor in the system. These messages contain a NULL terminated string in their data vector and have the function code **0x22222222**. These messages are originated by the **IOputs** function. This examines the processor's links for one with the **debug** flag set. If found, the debug message is sent through the **RemoteIOCPort** of that link. This will cause it to be delivered to the Processor Manager of the neighbouring processor. On receiving a debug message, the Processor Manager simply re-calls **IOputs** to pass it on. If the root structure **IODebugPort** is not **NullPort**, then **IOputs** will deliver the message to this port. This allows programs to intercept the IOdebug messages for display in some alternative way.

The function **IOdebug** provides a **printf** style interface for generating debug messages. Like **printf** this routine takes a format string followed by a number of parameters. Within the format string escapes of the form `%<`**char**`>` indicate how the next parameter should be displayed. Unlike **printf** this function does not support any of the format modifiers which may appear between the `%` and the format character. The format characters allowed are:

c Print a single character (char).

x Print in base 16 (int).

d Print in base 10 (int).

s Print string (char *).

o Print only last component of pathname (char *).

N Print whole pathname (char *).

A Print access mask (AccMask).

E Print error code (word).

P Print pointer (void *).

T Print object type (word).

X Print access matrix (Matrix).

C Print capability (Capability *).

M Print message header (MsgHdr *).

O Print Object in the format `"<`**Object: Type Nam**`>"` **(Object *)**.

S Print Stream in the format "<**Stream: Type Flags Name**>" **(Stream *)**.

F Print function code (decoded only on debugging systems) (word).

Normally a **newline** is added to the end of each string generated by **IOdebug**. This can be disabled by terminating the format string with a % character.

To avoid problems with mixed output, **IOdebug** waits on the **IODebugLock** semaphore in the root structure before starting, and releases it after having generated its output. This is the only instance where different tasks share a single data structure. If a task is terminated while holding this lock all subsequent **IOdebug** attempts will be blocked. However, it is a simple matter to write a program to re-initialise this lock if this is suspected:

```
#include <syslib.h>
#include <root.h>

int main(void)
{
RootStruct *root = GetRoot();

InitSemaphore(&root->IODebugLock,1);

IOdebug("IOdebug lock cleared");

return 0;
}
```

Chapter 11

The System servers

This chapter describes the Helios Processor Manager and Loader, both of which are part of the Helios Nucleus. The Processor Manager implements most of the operating system functions not covered by the Kernel. The description here covers its management of names and tasks in the processor. The description of the Loader covers the external interface of the Loader, and how it actually works. It also includes the Helios calling conventions and debugging interfaces.

11.1 The Processor Manager

The Processor Manager has two primary functions: to manage the processor's **Name Table**, and to manage the **tasks** running in the processor. These functions are described below.

11.1.1 The Helios naming scheme

It is a feature of Helios that all objects managed by the system are available through a single consistent naming scheme. This gives the appearance of a rooted tree of directories containing other directories and objects. The levels nearest the root of the tree are implemented in a distributed manner by the Processor Managers. Lower levels of the tree are implemented by servers.

The upper [1] part of the tree forms a hierarchy of logical processor, cluster and network names. Each processor is a directory named by an unique pathname in this hierarchy. Server names are placed within the processor directory in which they are running. The directory tree within a server is managed entirely by that server. While the server may alter the naming conventions at this point, it is not recommended.

While each Processor Manager logically contains a complete copy of the name tree, it is in fact built on a need-to-know basis. So, if a particular processor has not accessed any services in a particular processor or network, it will not have any knowledge of it. When a Processor Manager is asked by a task for access to an unknown service, it initiates a network wide search for it.

To allow servers to be named in a position independent way, the Processor Managers allow abbreviated names which omit higher levels of the name tree. In this case

[1] UP is towards the root of the tree, DOWN is towards the leaves.

the closest instance of that server will be located.

11.1.2 The I/O controller

When a task wants to send an indirect message to a server, it sends it initially to its **IOCPort**. Waiting on this is a thread, the I/O Controller or **IOC**, which was created by the Processor Manager when the task was created. It acts as the task's agent in searching the name table. Additionally, a special **LinkIOC** accepts messages from remote processors and acts as the agent for remote tasks for interaction with the local name table.

The IOC is responsible for locating the processor on which the server is resident, and then for forwarding the message to that processor. When passed an indirect GSP message it extracts the element indicated by the **Next** field and looks this up in a hash table of all the names it currently possesses. This implements the abbreviated name mechanism and should result in a pointer into the name tree. From here the IOC follows the pathname through the directory structure as far as possible. If the last name table node found contains a message port, because it is a local server name or a cached remote name, the **Next** field of the request is updated and the message forwarded to the port.

If the hash table lookup yields no match, or following the path does not end in a cached name, the IOC performs a distributed search for the last name item which did not match (see section 11.1.3 for details of the distributed search protocol). If this search fails the request is returned to its originator with an error. If the search succeeds, the message port that it returns is cached in the name table and the request is forwarded to it. Subsequent accesses to the same name will find the cached port and forward messages to it in the normal way.

The IOC is also responsible for part of the GSP fault recovery strategy. Retries of indirect GSP requests all contain a retry counter. Each cached name table entry contains a **Confidence** level which is reset to a default level each time a request with a zero retry counter is forwarded through it. Each time a request with a non-zero retry counter is sent, the confidence level is reduced by the value of the counter. When the confidence level reaches zero, the cached name is removed from the name table, and the next request will force a new distributed search for that name. If this fails then a hard error will be returned to the sender.

11.1.3 Distributed search protocol

The mechanism used by Helios to locate servers is the distributed search protocol. Initially any Processor Manager knows only about its local servers and as a result of the Kernel link idle exchange can send messages to the **LinkIOC** threads in its immediate neighbours. These near-neighbour connections are used by the distributed search protocol. The distributed search protocol consists of the following messages:

Search Request *Message Header:*

Flags	Preserve
Dest	Link.RemoteIOCPort
Reply	Local reply port
FnRc	FG_Search

Control Vector:

String	Name	name being sought
word	Id	Unique identifier
word	Link	Sender's link number

Success Reply *Message Header:*

Flags	NONE
Dest	Request.Reply
Reply	Server port
FnRc	Err_Null

Control Vector:

String	Name	Canonical name of service
word	Flags	Flags field
word	Link	Sender's link number

Failure Reply *Message Header:*

Flags	NONE
Dest	Request.Reply
Reply	NullPort
FnRc	Error Code

Control Vector:

String	Name	Set to -1
word	Flags	Set to 0
word	Link	Sender's link number

The **Name** field contains the service name being sought. If the first part of the name is already known (the cluster or processor name for example) then this should be included. The **Id** field is a random number which identifies this search, it is used to detect cycles in the processor network. The **Link** field contains the sender's number for the link this message is being sent on, it is used to detect link errors.

From the originating processor a search request message is transmitted on each of its active links. It then waits on the reply port for the replies to arrive.

In a neighbouring processor, the search message will be delivered to the **LinkIOC**. This first checks the **Id** field against a table of the most recent **Id**s received. If the identifier is found then a **Failure** message is returned to the reply port. Otherwise the **Id** is entered into the table and the local name table searched for the service name. If the name is found, and it is not a cached name from another processor, a success reply is returned. This contains the full canonical name of the service, a set of flags to OR into the name table entries **Flags** field, and the **Link** field from the original request.

If the name is not found locally, new search requests, containing the original **Name** and **Id** fields are transmitted on all the active links except the one on which the original request was received. A reply is then awaited exactly as in the originator's case.

Searchers wait until either a successful reply has been received, all neighbours have replied with a failure, or a timeout expires. In the last two cases, a **Failure** reply is returned to the originator. If a **Success** reply arrives, the reply port is freed, causing all subsequent responses to be destroyed by the Kernel. The **Success** reply is then returned to the originator.

Name table management

The management of the name tables is primarily the responsibility of the Processor Manager, there is little provision for external management. The only functions that a task can perform are to create a name in its local name table, delete that name, clear the name table of all cached names, and extend the name tree.

Name creation

A new name may be added at any level in the name tables, not necessarily just at the processor level. Such new names, however, are restricted to being in the current processor's path. They may not be added to other processors or clusters.

A name is added with the **Create** indirect operation. The target in the **Common** field indicates the name to be created, which must not already exist on this processor, but may exist on other processors. The type should be **Name**. The information structure passed is a **NameInfo** structure:

Port The server's initial request port, to which all indirect operations will be directed.

Flags Flags for the new name entry. The only flag which may currently be set is **StripName** which tells the Processor Manager to position the **Next** field of any request just past the server's name. If absent, **Next** will point to the server's name in the request.

Matrix Initial access matrix for the name. This will be **AND**ed with **DefNameMatrix** before being installed.

LoadData Originally intended to contain data to enable the automatic loading of the server when first accessed. This field in currently unused.

Because the port descriptor in this structure is not seen by the Kernel when this message is passed, it cannot be translated into a surrogate, and hence this operation is restricted to being sent only to the local Processor Manager.

Name deletion

The result of creating a name will be a pathname and capability for the new name. Only the capability returned from this create encodes the right to delete the name. The default name access matrix is *rwv:rx:ry:rz* which omits both delete and alter rights for all categories. Hence, only the server which installed a particular name is able to remove it.

Clearing the name table

Under certain circumstances it is useful for all the cached names in the Processor Manager's name table to be cleared. This is achieved by the **Reconfigure** operation.

Request *Message Header:*

Flags	Preserve
Dest	IOC port
Reply	Local reply port
FnRc	FG_Reconfigure

Control Vector:

IOCCommon	Common	Common part of GSP request

Reply *Message Header:*

Flags	NONE
Dest	Request.Reply
Reply	NullPort
FnRc	Return Code

The target of this operation is the processor's **/tasks** directory (to avoid problems with cached names in other processors).

Extending the name tree

This is performed by a normal **Rename** operation sent to the processor's own directory node. The **ToName** field should contain the complete new path name of the processor. This operation is restricted to adding extra levels of network and cluster names. Hence, the current processor name must be a terminating substring of the new name. For example, a processor may be renamed from **/Net/03** to **/Cluster/Net/03** but not to **/Cluster/Net1/03** or **/Net/04**. This function is used primarily by the Network Server when booting the processor network.

11.1.4 The Task Manager

The second major function of the Processor Manager is the management of all the tasks which have been created in the processor. The interface to this is presented through the **/tasks** directory in every processor. Some of the control functions are performed by the standard GSP operations while others are performed by special operations. The full description of these protocols is given in in the chapter entitled **GSP**.

Task to IOC messages

In addition to sending all indirect operations to its **IOCPort** a task may send it a number of private messages. These interact with the Task Manager part of the Processor Manager.

MachineName

Request *Message Header:*

Flags	preserve
Dest	IOC port
Reply	local reply port
FnRc	FG_MachineName

Reply *Message Header:*

Flags	NONE
Dest	Request.Reply
Reply	NullPort
FnRc	Return Code

Control Vector:

String	Name	Machine name

This operation requests the full name of the current processor. This string may change over time as new levels are added to the processor name (see section 11.1.3 above).

SetSignalPort

Request *Message Header:*

Flags	preserve
Dest	IOC port
Reply	new signal port
FnRc	FG_SetSignalPort

Reply *Message Header:*

Flags	NONE
Dest	Request.Reply
Reply	old signal port
FnRc	Return Code

Signal Message *Message Header:*

Flags	NONE
Dest	signal port
Reply	NullPort
FnRc	EC_Recover—SS_ProcMan—EG_Exception—EE_Signal—signal

This operation is used to establish a port for the delivery of signals to the task. Whenever a signal is generated for the task either through **SendSignal** or from the Kernel or the Processor Manager, a message will be sent to this port.

Exit

Exit Message *Message Header:*

Flags	preserve
Dest	IOC port
Reply	new signal port
FnRc	EC_Error—EG_Exception—EE_Kill

Control Vector:

word	Code	Program return code

This message is sent by the task to its **IOCPort** when it wishes to exit. It is sent by the System library **Exit** function. This is the only way that a task can exit. However, under certain circumstances this message is generated internally by the Processor Manager to force a task to quit.

The **Code** field contains the program's return code, the argument supplied to **Exit**. This will be returned to the task's parent in the Program info Message.

11.1.5 Debugging system control messages

In debugging and development versions of the Helios Nucleus a set of **IOdebug** messages can be generated. These are controlled by a set of flags in the Processor Manager and in each task structure. These flags are changed by the following messages.

Debug

Request *Message Header:*

Flags	preserve
Dest	IOC port
Reply	local reply port
FnRc	FG_Debug

Control Vector:

IOCCommon	Common	Common part of GSP request
word	Flags	Debug flags to be set

Reply *Message Header:*

Flags	NONE
Dest	Request.Reply
Reply	NullPort
FnRc	Return Code

This operation should be directed to a processor's **/tasks** directory. The set of flags in the **Flags** field are **exclusive-OR**ed with the current set of debug flags in the Processor Manager. The flags are interpreted as follows:

ioc1(0x01)	Report all indirect operations sent by all tasks.
ioc2(0x02)	Report the name table lookup process for each indirect operation.
ioc3(0x04)	Report on message forwarding for each indirect operation.
search(0x08)	Report on each distributed search made from this processor.
searchwork(0x10)	Report on each distributed search received at this processor.
mem(0x20)	Report all memory allocations made by the Processor Manager.
tasks(0x40)	Report all task operations.
info(0x01000000)	Report processor state every 5 seconds.

Some of these options will produce large quantities of output, and will significantly affect the system's performance.

In addition to the above flags, the **Flags** field of all tasks created will be initialised to **(debug_flags>>8)&0xFF**. This allows a set of default task debugging flags to be set for all tasks (see next section).

SetFlags

Request *Message Header:*

Flags	preserve
Dest	IOC port
Reply	local reply port
FnRc	FG_SetFlags

Control Vector:

IOCCommon	Common	Common part of GSP request
word	Flags	Flags to set

Reply *Message Header:*

Flags	NONE
Dest	Request.Reply
Reply	NullPort
FnRc	Return Code

This operation affects the **Flags** field of a **task** structure. It exclusive-ORs the request **Flags** field with the structure's **Flags** field. In the System libraries, certain of these flags are inspected and debugging information is output if they are set. These flags are defined in **task.h**:

ioc Report each indirect operation and its result.

stream Report each direct operation and its result.

memory Report each memory allocation and release operation.

error Report all message passing errors.

process Report all task related operations.

info Produce a single report on the state of the task.

meminfo Produce a single report on the state of the task's heap.

fork Report each **Fork** operation and the stack used by the thread when it exits.

servlib Report incoming requests in Server library dispatcher.

fixmem Disable the acquisition of new heap block from the Kernel.

11.2 The Loader

The **Loader** server is part of the Nucleus and is responsible for loading code from files into memory in preparation for execution. It manages the demand loading of Shared libraries, and the general sharing of code between tasks. Where possible it will obtain the code needed from another processor in the network, rather than fetch it from the file system.

11.2.1 Code management

Code is loaded into a processor through the **Load** request, which is a variant of the **Create** request (see later). The first thing the Loader must do is to locate a copy of the code required. There are five possibilities:

1. It may already be loaded in this processor.
2. The Loader may already contain a symbolic link to the code.
3. It may be resident in a neighbouring processor.
4. If a library, it may be in **/helios/lib**.
5. Its location is supplied in the request.

If the code is already loaded, its **use count** is incremented and a successful reply is returned to the **Create** request. To allow for case insensitive file systems, the name comparison is case insensitive.

If the Loader contains a symbolic link to the code, placed there by the **cache** command, then the name and capability stored in it are used to read the code.

If the Loader has no knowledge of the code, it needs to find it. The first attempt is to ask its immediate neighbours. To do this it sends a message containing the name of the code required, plus its own processor type, to each of its neighbours. If the neighbour is of the same processor type, and it contains a copy of the code then it returns its name and capability. Otherwise an error is returned.

If the code being sought is a library from a Resident library reference **ResRef**, then the directory **/helios/lib** is scanned for the file. The matching process here is complicated by the potential use of host file systems which are case insensitive. The library is first sought using the name from the **ResRef** directly. If this fails the processor type is appended (on the Transputer either **.t4** or **.t8**) and the search retried. If this fails the original name is converted to lower case and the two searches repeated.

If the code is not a library then a source for it will have been provided in the load request in the form of a name and capability.

To load a piece of code, either from disc, or from another Loader, the object is opened and the first twelve bytes read. These are an **ImageHdr** structure which defines the size of the code. A block of memory of this size is allocated and the indicated number of bytes read into it. The code is then scanned for Resident library references. If the library is already present, a pointer to it is placed into the code. Otherwise its whereabouts are sought, using this algorithm recursively.

Each code object managed by the Loader has two variables associated with it: a **use count** and a **retain flag**. Each time a code object is used by a task its use count is incremented. When the task exits, the use counts on all its code objects are decremented. If the use count of any code object falls to zero, and its retain flag is false, its memory is reclaimed, otherwise it is kept. The retain flag is set on all libraries when they are loaded, and on all code loaded through a symbolic link. Additionally, if an attempt is made to **Delete** a piece of code, its retain flag is set to false. If its use count is zero it is deleted immediately, otherwise it will be deleted when it is no longer in use.

11.2.2 Error detection

All code in Helios is read-only, there is no relocation, and no code is self modifying. The Loader uses this fact to implement a simple error check on all code. When a piece of code is loaded, it is checksummed. Whenever a task starts or exits, all the code it uses is checksummed again and the result compared with the original sum. If they differ an error message is generated on the logger and the code's retain flag cleared. This will force it to be unloaded at the earliest opportunity and if needed again, a fresh copy will be fetched.

11.2.3 Loader protocol

In keeping with the Helios conventions, the Loader presents a directory interface through the **/loader** directory on all processors. The entries in this directory each represent a single piece of code loaded into that processor, or a symbolic link to it. The normal operations for directory listing may be applied to this server. In addition the following operations have an extended or modified function.

Create

The **Create** operation is used to load code into the Loader. This is modified by **OR**ing the **FF_LoadOnly** flag into the function code **F** field. The optional **Info** field is a **LoadInfo** structure:

Cap Capability of object to be loaded.

Matrix Initial access matrix of object, this will be ANDed with a default matrix.

Pos Start position of image in object.

Name The canonical pathname of the object to load.

The **Cap** and **Name** fields should have been obtained from the result of an **Open**, **Create** or **Locate** operation. The **Pos** field is present to allow the loading of code which is embedded in some other information in a file.

Open

Access to a loaded code object is acquired through the normal **Open** operation. The action of **Open** depends on whether the mode flags include the **Execute** mode. When **Execute** is not required, a direct operation port is returned and the object may be read just like a normal file. This is used, for example, for transferring the code from one processor to another. If **Execute** mode is requested then the **Object** field of the open reply contains a pointer to the start of the code in memory. No direct operation port is returned, but the **Closeable** flag is set, forcing the client to send a **CloseObj** operation when it has finished with the code.

Link

The **Link** operation is used unmodified. The action of this request is to place a reference in the Loader under a given name to a particular piece of code. Any subsequent use of this code will result in it being loaded and retained.

This facility is used to mark particular commands and libraries as cacheable. The actual code will only be loaded on demand, but will be retained once loaded.

Delete

This operation marks an object in the Loader as a candidate for removal. It does not force its removal unless the code is not currently in use. Otherwise the object will be removed the next time the use count reaches zero.

CloseObj

When an object has been opened for execution the server ensures that a **CloseObj** reply will be generated when the object is closed. Because no direct operation port is returned, a normal **Close** operation cannot be used.

The action of **CloseObj** is to reduce the use count on the code object being closed. If it reaches zero, and the retain flag is false, the code is deallocated, and the use counts on all the libraries it used are also decremented. This may cause further deallocations and use count decrements recursively.

Chapter 12

Writing servers

12.1 Introduction

The purpose of this chapter is to explain how to write servers under Helios. Helios is based on the client-server model of computing. To do any work under Helios other than pure computational work, for example inverting a matrix or sorting some data in memory, programs need to interact with servers. For example, to read data from a file it is necessary to interact with a file server, and to send diagnostics to a screen it is necessary to interact with a window server. As far as application programmers are concerned this usually happens transparently. User applications simply call standard libraries such as the Posix or C library, and the system performs the necessary interaction with the server. This chapter looks at the other side: the server side. It explains how advanced programmers can write their own servers, thus extending the facilities provided by Helios.

Even with the support provided by Helios, writing a server is a non-trivial job, and should be attempted only by experienced programmers who have used Helios for some time. This chapter assumes that the reader has some understanding of the workings of Helios, such as the naming scheme and the structure of the Nucleus. Since the vast majority of servers are written in the C language this chapter also assumes familiarity with that language.

Section 12.2 provides a general introduction to servers, starting with the Unix daemon mechanisms and then describing how and why Helios servers are different. It gives a brief description of the Helios message passing system, the General Server Protocol, and the Server library.

Section 12.3 contains the code for a simple server, a **/lock** server which provides locking between separate programs. The sources of this lock server and other servers described in this chapter are supplied with Helios in the directory **/helios/users/guest/examples/servers**.

Section 12.4 gives a more detailed description of some of the aspects of writing servers: error codes, protection, and other server library routines.

Section 12.5 gives the code for a more complicated server, the **/include** disc which acts as a simple read-only file server. The section continues with a description of how to extend this server so that it becomes a full RAM disc.

Section 12.6 describes Helios device drivers, what they are for, and how to access them from a server. To illustrate the use of device drivers the sources for another

server, a **/keyboard** server, together with an example device driver, are included.

Section 12.7 concludes this chapter with a description of how to write servers without using the Server library. An example is given of an MS-DOS compatible file server.

This chapter has been written mainly as a tutorial rather than as a reference work. Further information on particular aspects can be found in other chapters. The online help system and the Helios encyclopaedia contain full details of the various routines outlined in this chapter. The Helios header files are also a good source of information.

12.2 Helios servers

The basic concept of the client-server model is very simple.

1. Somewhere in a network of processors there is a program **A** that requires a service. The service could be reading some data from a file, running another program or a collection of programs, or showing a picture of the starship Enterprise on a graphics display. Many services involve I/O of some sort, but not all.

2. The program may be physically unable to perform the service. For example, some essential piece of hardware such as the graphics display might be attached to a different processor, making it inaccessible.

3. Alternatively it may be inconvenient for the program to perform the service. For example, application programs do not usually want to control a hard disc, keeping track of all the sectors and maintaining the directory structure. Instead such programs expect to interact with an existing program which implements a filing system.

4. There is another program **B**, possibly on the same processor as **A** or possibly on a different processor. This program has been written to perform a specific service. Occasionally a program may provide more than one service. Alternatively a collection of programs may be required to provide one service.

5. Program **A** can communicate with program **B** and it can request that the required service be performed on its behalf.

6. In this situation, program **A** acts as a client, and **B** acts as a server.

The client-server model is by no means unique to Helios. In fact it is an essential part of many existing operating systems, including most Unix systems. This chapter describes the daemon-based approach used by Unix, as the mechanisms involved are supported by Helios. However, Helios was designed from the beginning around the client-server model and provides much more extensive facilities.

12.2.1 Unix daemons

Originally Unix was designed as a single-processor operating system isolated from the rest of the world. With the development of networking such as ethernet, this situation changed, and client-server facilities had to be added to Unix, for example in the form of **sockets**in BSD UNIX.

1. A program willing to provide a service must become a daemon. Unix systems typically come with a number of these daemons, for example, **telnetd** is the telnet daemon permitting users to log in from remote machines over the network.

2. A daemon must create a socket and **bind** an address to this socket. This registers the daemon with the system, thus allowing clients to contact the server. The daemon can then accept incoming connections. The addresses used are simple numbers such as **69**, rather than text names. For convenience, Unix maintains a **services** file, mapping names such as **nfs** onto the numbers, but system administrators are responsible for maintaining this file.

3. When a client needs to access a daemon it creates a socket. It must then specify the service required, using the service address, and the address of the processor running the service, which is another number. Again there is a file called **hosts** mapping text names onto processor addresses. The client can then connect to the daemon. Note that the client must know the identity of the processor running the daemon.

4. The socket routine returns a Unix file descriptor, and the **read()** and **write()** routines can be used on this descriptor to achieve communication between client and server. The underlying nature of this communication, whether stream-based or datagram-based, can vary.

5. At any one time a daemon may have several clients connected to it, all requesting different services. Typically a daemon's main loop calls the **select()** routine to find out which of the current clients is requesting a service, or to accept connections from new clients. For every such client the daemon will read details of the service required from the corresponding socket, perform the service, and send back a reply to the client.

6. The protocols used between the clients and the daemon are **NOT** defined by the operating system. Hence one daemon might interpret a request code **1** to mean "open file", whereas another daemon might interpret exactly the same request to mean "delete file".

Helios provides the Unix socket calls as part of its Unix compatibility, and many of the usual Unix daemons are available. Hence existing Unix daemons can be ported to Helios with little or no effort. For example, the Helios implementation of the X server uses the standard socket based server code rather than the Helios mechanisms.

12.2.2 Helios servers

Helios has been designed specifically around the client-server model. In particular the Helios naming scheme has been designed around servers.

1. Under Helios a program willing to provide a service, and which does not want to use the Unix mechanisms, must become a server. Occasionally a Helios program may become a daemon as well as or instead of a server, to provide the service over a local area network such as ethernet as well as within a multi-processor Helios machine.

2. A server must register itself with the system by creating a new name inside the processor. Helios always uses textual names for its addressing rather than numbers.

3. Clients can access most servers by standard library calls. For example, exactly the same Posix **open()** routine can be used to access a filing system on the same processor, a ram disc on some other processor in the network, a window inside an I/O processor somewhere in the network, or a serial port attached to a modem. There is no need for any special routines on the client side to access a service.

4. Furthermore, the client does not need to know the location of the service. For example, If a client tries to access **/fs/users/dick** and the system has not yet accessed the server **/fs**, there will be a distributed search throughout the network of processors.

5. As far as the client is concerned, the exact mechanisms used to access servers are irrelevant, because the client simply uses standard I/O calls such as **open()** and **stat()**. In fact there is a System library which turns these calls into Helios messages sent to the appropriate servers. Servers accept these messages, perform the requested service, and send reply messages. These replies are accepted by the System library, and the original library call will return.

6. Helios servers are usually internally multi-threaded. For every client accessing the service there will be a separate thread inside the server program responsible for handling requests from that client. Hence if two clients want to access two different files inside the same file server, these files can be accessed in parallel. However this does involve a cost, because servers must perform some synchronisation operations between the various threads.

7. The basic protocol used between Helios servers and clients, or rather the System library which performs all the work on behalf of the client, is defined by Helios. This protocol is the General Server Protocol. It is recognised by all Helios servers, although not all servers need to support all parts of the protocol. For example, a mouse server does not need to accept incoming data as a result of a **write()** call, because there is nothing useful it could do with such data.

When it comes to writing Helios servers the essential points to note are **message passing** and **General Server Protocol**. Helios servers must accept and respond to incoming messages, so it is necessary to understand how this message passing works. The contents of all the various messages for all types of servers is defined by a protocol, and this protocol must also be understood.

12.2.3 Message passing

Helios servers accept requests from clients in the form of messages. These messages are generated by the System library as a result of the client's library calls. Servers must receive these messages, perform the requested service, and send back replies. This subsection gives an outline description of Helios message passing, which has already been described in detail in previous chapters.

The basic data structure used for message passing is the MCB, as defined in the header file **message.h**.

```
typedef uword         Port;
#define NullPort      ((Port) 0)

typedef struct MsgHdr {
        unsigned short DataSize;
        unsigned char  ContSize;
        unsigned char  Flags;
        Port           Dest;
        Port           reply;
        word           FnRc;
} MsgHdr;

typedef struct MCB {
        MsgHdr        MsgHdr;
        word          Timeout;
        *Control;
        byte          *Data;
} MCB;
```

Messages are sent to message ports. If the message's destination is going to send a reply, it will need a reply port. There is a special value **NullPort** which is recognised by the system as an invalid port.

A Helios message consists of three fields. The first is a message header, which has a fixed size of 16 bytes. The second is a **control** vector, a vector of 32-bit integers or **words**. In theory this vector can contain between 0 and 255 words, the exact size being defined in the **ContSize** field of the message header. In practice, GSP messages never involve more than 16 words, but Helios messages could be used for communication other than interaction between client and server. The third field is a **data** vector, containing between 0 and 65535 bytes. The exact size is defined in the **DataSize** field of the message header. The MCB structure contains a message header and pointers to the control and data vectors.

In addition to the control and data vector size, the message header contains four fields. The **flags** field controls various options in the message passing system. For servers there are two important flags:

1. **MsgHdr_Flags_preserve**: By default a message port can be used only once. If a message is sent to a particular message port, that port cannot be used again. Usually this is exactly what is required. For example, a server receives a request with a suitable reply port in the message header. It performs the service and sends back a reply to the client using that reply port. The transaction is now complete so the system can reclaim the message port, preferably without any further work required on the server's part. Occasionally it is necessary to send multiple replies to a single message port. For example, when replying to a **read** request for 200K of data, this data must be split into multiple messages, because each message is limited to 64K. All of the reply messages except for the final one should use the **preserve** flag to prevent the system from reclaiming the message port too early. The final reply should not use the flag because the transaction has now finished and the system should do the cleaning up.

2. **MsgHdr_Flags_sacrifice**: For some servers it does not matter whether or not a particular message actually reaches a client. For example, a mouse device can generate tens of messages every second while the user is moving the mouse, and if one of these messages is lost this does not matter greatly. Setting the **sacrifice** flag is a hint to the Kernel that the message can be discarded if necessary, usually because the network is being overloaded with message traffic and the Kernel is running out of communication bandwidth. If this flag is not set, the Kernel still does not guarantee delivery, but it will try harder.

The destination message port usually refers to the client that sent the original request, and was contained in the **Reply** field of the incoming message. For some requests a reply message port can be included which would allow the client to send messages directly to the server, the main use being for opening a stream connection. For most requests the reply message port should be set to **NullPort**, indicating that the transaction has completed. The **FnRc** field is used to hold the function request code for incoming messages, or the error code for replies. The structure of request and reply codes is described in more detail in section 12.4. The whole MCB contains one other field, a timeout for the message transaction.

There are four Kernel routines to perform message passing: **PutMsg()** sends a message; **GetMsg()** receives a message on a single message port; **XchMsg()** is used mainly by clients to send a request and receive a reply; and **MultiWait()** can be used to wait for messages sent to several different message ports. Of these four the first two are the most important for writing servers. Servers either call them directly or they are called automatically as a result of using the Server library.

An important aspect of Helios message passing is that the communication is not always totally reliable. The vast majority of messages sent will arrive at their destination without problems, but there are circumstances under which some messages will be lost.

1. The client program itself may crash while the server is handling a request. Given the absence of memory management hardware on some types of processors running Helios this possibility must be anticipated. Even if memory management hardware is available, hardware irregularities can cause the occasional crash. In this case the reply generated by the server must be discarded by the system, and the server must recover somehow from a client that disappears.

2. Some other program on the same processor as the client crashes the processor. Since Helios is a multi-tasking system this is possible. Servers cannot distinguish between this case and the previous one.

3. A server may crash while the client has an open stream to it. This is less likely than the previous case but it can happen. The System library routines will return error codes which should be propagated to the user's application. Sometimes the System library can find an alternative server offering the same service, and will transfer to that one automatically, but this is not always possible.

4. A processor used to route messages between the client and the server crashes as a result of an errant program or a hardware failure. In the worst case, this

processor crashes just when a message has been read from one link but before it can be sent out of another. Because there are timeouts on all messages, the system can detect such failures and take recovery action. This recovery action is always initiated from the client side, rather than from the server side. The route to recovery may be as simple as sending the message again, forcing a new distributed search to find an alternative route to the same server, but more complex recovery techniques are also built into the system. All of this is transparent to the application programmer.

5. The network may be so congested with message traffic that messages can no longer be transmitted. Again, the system will automatically take recovery action and the network should always continue running.

Given that message passing is inherently unreliable it is essential to design the client-server communication so that it can cope with failures, and servers must be written to allow for failures.

12.2.4 The General Server Protocol

The message passing routines allow communication between the System library and Helios servers. The format of the message header is clearly defined. The content of the **FnRc** field and of the control and data vectors is not defined by the message passing system. Instead it is defined by the higher-level General Server Protocol (GSP). GSP is defined in greater detail in chapter 13, *General Server Protocol*. The following is a brief introduction to the basics of the protocol.

The following GSP messages can be sent to servers for any objects.

Open

This is used to establish a stream connection between the client and the server. Essentially this is used to obtain another message port which can be used to interact with one thread in the server dedicated to handling this stream. This message port can then be used to send other requests such as **read** and **write** which can involve very large amounts of data. There are various open modes such as an implicit **create** if the specified object does not yet exist. The same open request can be used for files, directories, devices, and so on.

Create

This is used to create a new object, for example a new window on the screen.

Locate

This is used to test whether or not a particular object exists. It returns certain useful pieces of information, for example whether the object is a file or a directory.

ObjectInfo

This is used to obtain additional information about an object, for example its size and a date stamp for when the object last changed. Essentially this is used by the Posix **stat()** routine.

ServerInfo

This obtains information about an entire server rather than about a specific object within the server. The information supplied depends on the server. File servers return disc usage statistics such as the disc size and how much of it has been used. The Processor Manager gives performance statistics about the processor such as memory usage and link traffic.

Delete

This is used to remove an object from the name table. This might involve deleting a file and returning its disc blocks to the free pool. Alternatively it might involve eliminating a server's name table entry when that server is exiting.

Rename

This changes the name of an object within the server. It cannot be used to move an object from one server to another because a GSP message transaction interacts with a single server, not with several servers. Different servers may or may not allow objects to be moved from one directory to another.

Link

This is usually supported only by file servers which can store data permanently. It creates a symbolic link to some other object in the naming hierarchy together with the capability needed to access that object.

Protect

This is analogous to the Posix **chmod()** routine, which is used to change the access mask of an object. Since Helios uses capabilities rather than true Posix access masks, the match is not exact.

SetDate

This changes the timestamp of an object. It is used mainly by the **touch** command.

Refine

This can be used to generate a new capability for an object. It is described in more detail in section 12.4.1.

CloseObj

This is used by some servers as a hint that an object is no longer required. It can be thought of as a less drastic form of a **Delete** request. For example, when a piece of code loaded in memory is no longer required a **CloseObj** message is sent to the Loader. If no other application is using the code at the time, the Loader may remove the code from memory.

Revoke

This is also used to implement Helios protection. It invalidates all existing capabilities.

All of these messages operate on named objects. The System library routines corresponding to these GSP messages typically take the following form:

```
word ObjectInfo(Object *context, string name, byte *info);
```

The first argument is an object, usually the current directory. This object contains a capability giving the client a certain amount of access to that object and hence to objects within its subdirectories. The second argument is a string specifying the file or device name relative to this directory. The third argument is routine specific, in this case a buffer for the required information. The routine packs the details of the context object and the name into a message, together with any request specific information, and passes this message to the system. The system then forwards this message to the correct server, performing a distributed search for the server if necessary.

For example, a message might have a context object of **/Net/00/helios/include** and a filename of **stdio.h**. The system scans these names. If the client is in the same Helios network as the server, the current processor will already know about **/Net**, and the next part of the name can be scanned. If the client is in a different network which happens to be attached to the one containing the server, the system forwards the message to the nearest processor in **/Net**. The next part of the name is **/00**, which might be the processor running the client or it might be a different processor. If it is the latter, the message is forwarded to processor **/00**. Once the message is in processor **/00** it is passed to the server **/helios**, which can start processing the request.

Making the system search name tables for requests works well for operations such as **Delete** and **ObjectInfo** which are relatively infrequent and involve small amounts of data. For I/O intensive operations such as reading and writing files, GSP uses a different set of messages. These messages can be used only over stream connections created as a result of an **Open** request. The client opens a stream to an object within the server. The server starts a separate thread to handle the request, and the request returns a message port private to the stream connection. The client can now send **read** and **write** requests directly to this message port and hence to the appropriate thread within the server. The requests that can be used on open streams are listed below.

Read

This is used to extract data from the object.

Write

This is used to put data into the object.

Getsize

This indicates the amount of data in the object. The exact meaning of this can vary. For example, in a file server the size indicates the current size of the file, whereas in a pipe it indicates the amount of readable data currently buffered.

SetSize

This is rarely used, but it can truncate or expand files to a particular size.

Close

This terminates an existing stream connection. The message ports are released and the thread in the server handling the connection terminates.

Seek

This changes the current position within the object. It's usefulness is limited to files.

Getinfo

This obtains control information about an object. For example, if used on an RS232 port it returns details about baud rates, parity settings, and so on. The nature of the information depends on the server, but the **Attributes** structure is commonly used.

SetInfo

This changes control information about an object. For example, it can be used to change the baud rate of an RS232 port.

EnableEvents

This is used with event-driven devices such as mice, to start receiving event messages. The server receives a message port and will start sending messages to this port as soon as they are generated. The System library returns this message port to the application program, which should start receiving messages.

Acknowledge

This is used with event-driven devices to inform the server that certain messages have been received.

NegAcknowledge

This is also used with event-driven devices to indicate that one or more messages have been lost.

Select

This is used to determine whether the object has data ready to be read, or whether the object can receive data.

GSP has been designed to cope with lost messages. Servers should be **stateless**, in other words the server should not need to remember any information about the client between requests. Every request contains all the information needed by the server to perform the requested service. For example, consider a client that writes data to a file. Each **write** is a separate transaction, and each request contains the position within the file where the data is to be written, as well as the data itself. This copes with the following situation:

1. The client sends a **write** request for 1024 bytes at position 4096.

2. The server performs this **write** and sends back a reply.

3. The reply message is lost for some reason, and the client must retry. This involves another **write** message.

4. The server does not know that a message has been lost. If the data was sent to it without a position, it could only write this data at location 5120, after the previous data. Hence the file would contain corrupt information.

5. Since the file position is sent with the **write** message, the server can put the data at the right position and the output file is not corrupted.

In this example the essential piece of state information is the current position within the file. To achieve stateless servers the GSP messages must be **idempotent**: if a client sends two or more identical messages to a server, usually because of a failure, this has the same effect as sending the message exactly once without any failures, and the client must receive the same reply. The **Write** messages described above are an example of idempotentcy.

In practice, fully idempotent protocols are complicated and do not necessarily behave in the expected manner. For example, consider a message which renames a file **tom** to a file **dick**. The first time this operation will succeed and send back a suitable success code. If the message is repeated, the server will send back an error message because the file **tom** no longer exists. Hence two identical messages generate different replies, and the protocol is not fully idempotent. However the **Rename** operation does work in the expected manner. Similarly deleting a file which does not exist should, in theory, return a success code: a previous identical message might have deleted the file and generated a success code. For such cases GSP follows common sense rather than strictly obeying the theory.

12.2.5 The Server library

Many servers require similar facilities. These include the following:

1. Maintaining a directory tree in memory. Except for some file servers which hold their directory trees on a disc this is true for the vast majority of servers.

2. Accepting incoming requests aimed at the server and starting up a thread to handle those requests. For most operations this thread will perform the requested service and terminate. For an **Open** operation, the thread continues running to handle requests sent to the stream port, until a **Close** request is sent to that stream.

3. Walking down the directory tree and possibly adding and deleting entries. Care must be taken here to synchronise with other threads.

4. Many requests can be handled using identical code in different servers. For example, the information required to handle the **ObjectInfo** requested is usually all stored in the directory tree, and no special code is required in the server to access it. Hence different servers can share code to perform these operations.

5. All servers require miscellaneous operations such as sending back error messages.

To support these common requirements of servers, Helios provides a Server library, embedded in the Nucleus. Its use is described in the sections to follow, with the example programs.

12.3 A /Lock server

This section contains the source code of a simple server which provides a locking service between different programs. The lock server is not intended to be particularly useful, but it does illustrate many of the concepts of writing Helios servers. The full sources of this and a number of other servers are in the directory **/helios/users/guest/examples/servers**, shipped with the Helios system. A **lock** server needs to support three main operations: it must be possible to create a unique named lock, which should fail if that lock already exists: it must be possible to delete an existing lock; and it must be possible to examine the **/lock** directory to find out which locks currently exist.

12.3.1 Header files

Since the **lock** server is written in C it will need a number of header files, included at the start of the source.

```
#include <helios.h>
#include <string.h>
#include <codes.h>
#include <syslib.h>
#include <servlib.h>
#include <gsp.h>
#include <root.h>
#include <link.h>
#include <message.h>
#include <protect.h>
#include <event.h>
#include <nonansi.h>
```

When writing servers it is often necessary to consult these header files, if only because the header files tend to contain the most up to date information about parts of Helios. Some knowledge of what is in each header file is useful.

helios.h contains type definitions and macros which are used frequently by the Helios system programmers, including the example servers in this chapter. Among the most common macros used are **New()** to allocate some memory for a data structure, and **Null()** to check whether or not an operation such as memory allocation failed.

string.h contains prototypes for string and memory manipulation routines such as **strcpy()** and **memset()**.

codes.h holds the details of how request and error codes are encoded in the **FnRc** field of a message header. This header file is necessary if the server is to return sensible error codes.

syslib.h defines the System library, including options such as the various different modes in which a file or a device can be opened.

servlib.h contains the data structures and routines provided by the Server library.

gsp.h defines all the messages in the General Server Protocol.

root.h contains details of the root data structure, which is essentially where the Kernel holds all its private data. Occasionally a server may need to examine the current state of this data structure, using the **GetRoot()** macro.

Note: No program should ever change the root data structure.

link.h contains details of how to access processor links directly. Some servers interact with the actual hardware by sending and receiving data to and from a **dumb** link, typically because there is a processor dedicated to an I/O task, such as a disc controller attached to the link.

message.h has the data structures used for message passing, together with prototypes for the message passing routines.

protect.h contains various definitions and macros used to implement the capability based protection mechanism.

event.h contains data structures and function prototypes to handle hardware interrupts (the event line in computer based systems).

nonansi.h has prototypes for some miscellaneous routines: **Fork()** is used to start up another thread; **IOdebug()** provides a low-level debugging facility similar to **printf()**, which is useful mainly for system programmers in a single-user system; the use of on-chip memory (if available) is permitted by **Accelerate()** and **AccelerateCode()**.

12.3.2 Program startup

All servers tend to start in much the same way.

1. Some servers are ordinary C programs which have a normal environment, and hence they can examine their arguments and environment strings for options.

2. Other servers are designed to be run from the **initrc** file or from the network resource map, without an environment. During development it may be desirable to start these servers from the shell, so there is typically a compile-time option to receive an environment. Such servers are usually linked with the **s0.o** startup code instead of **c0.o**, as described in chapter 3, *Programming under Helios*.

3. Various bits of data must be initialised, notably the initial directory tree.

4. A message port must be obtained, so that the server has a way of receiving its incoming requests.

5. If the server controls a piece of hardware, usually this must be initialised. This may involve loading a device driver or interacting directly with the hardware. Often one or more threads must **Fork()** off to interact with the hardware. Details of this are specific to the server.

6. The server must be registered with the system. Unless the system knows that a particular program is willing to provide a service, distributed searches for that server will fail and clients will be unable to access the server.

7. If the server is to be embedded in the Nucleus, it must send an acknowledgement back to the Processor Manager. This is described in more detail in chapter 3, *Programming under Helios*.

8. The server calls a **Dispatch()** routine in the Server library, to start accepting messages from clients.

9. The **Dispatch()** routine usually does not return, because servers are usually started when the machine is booted and continue running until the machine is powered down or rebooted. Nevertheless, servers may provide one or more ways of being shut down, mainly as a debugging feature. If so, there is usually some tidying-up code at the end.

Hence the **main()** routine of a server tends to look like this.

```
int main(void)
{ Port     server_port;

#ifdef DEBUG
     /* If the program is to be started from the shell  */
     /* but still linked with s0.o then it must accept  */
     /* an environment.                                 */
  { Environ    env;
    GetEnv(MyTask->Port, &env);
  }
#endif

   /* Initialise directory tree                         */

   /* Obtain a message port for this server             */
  server_port = NewPort();

   /* Initialise the hardware                           */

   /* Register the server with the system               */

   /* If embedded in the nucleus, acknowledge start-up  */
#ifdef IN_NUCLEUS
  { MCB       m;
    InitMCB(&m, 0, MyTask->Parent, NullPort, 0x456);
    (void) PutMsg(&m);
  }
#endif

   /* Call the Dispatcher */

   /* Tidy up */

  return(EXIT_SUCCESS);
}
```

12.3.3 Initialising the directory tree

The following code is used to initialise the directory tree.

```
static DirNode       LockRoot;

int main(void)
{ ...
  InitNode((ObjNode *) &LockRoot, "lock", Type_Directory, 0,
           DefDirMatrix);
  InitList(&LockRoot.Entries);
  ...
}
```

Maintaining the directory tree involves two main data structures: **DirNode** is used for directories, and **ObjNode** is used for files or devices within a directory. A **DirNode** contains the following fields:

```
typedef struct DirNode {
    Node           Node;          /* link in directory list   */
    char           Name[NameMax]; /* entry name               */
    word           Type;          /* entry type               */
    word           Flags;         /* flag word                */
    Matrix         Matrix;        /* access matrix            */
    Semaphore      Lock;          /* locking semaphore        */
    Key            Key;           /* protection key           */
    struct DirNode *Parent;       /* parent directory         */
    DateSet        Dates;         /* dates of object          */
    word           Account;       /* owning account           */
    word           Nentries;      /* number of entries in dir */
    List           Entries;       /* directory entries        */
} DirNode;
```

Linked lists are used to hold the directory tree because they provide convenient data structures. The name of the directory can be up to 31 characters long, plus a \0 terminator, which should suffice for most uses. The directory has a type, usually just **Type_Directory**, but other types are described in section 12.4, as are the various possible flags. The **Matrix** and **Key** fields are used to implement the protection mechanisms. The semaphore guards against concurrent access to the directory. For example, if a server receives two simultaneous requests, one of which involves adding an entry to the directory and the other of which involves removing an entry, the server will contain two threads handling these requests. Unless these threads synchronise with each other, the data structure is likely to be corrupted. The **Parent** field points at the parent directory, if it exists. The data structure contains time stamps for when the object was created, when it was last changed, and when it was last accessed. These are usually updated automatically by the Server library. The meaning of the **account** field varies from server to server: it is one of the fields printed out by **ls -l**; for example, listing the Session Manager might give the following output.

```
% ls -l /sm
4 Entries
d r--x----   0    132 Sat May  4 15:51:35 1991 Windows/
f r------- 108      1 Sat May  4 16:29:18 1991 bart
```

In this case the **Windows** subdirectory has no account information, but the **account** field for user **bart** is **108**, corresponding to the Posix user id. A common use for the **account** field is to keep track of the number of open streams to an object. For listing the **/window** server might produce the following output:

```
% ls -l /window
2 Entries
f rw----da   4      0 Sat May  4 15:51:04 1991 User1
f rw----da   3      0 Sat May  4 15:51:04 1991 console
```

In this case window **User1** has four open streams, and window **console** has three. The final two fields in the **DirNode** structure are used to hold the contents of this directory: the number of entries, and a linked list holding the entries.

The **ObjNode** structure is used to hold details of files and devices, rather than of directories. It is almost identical to the **DirNode** structure.

```
typedef struct ObjNode {
    Node           Node;          /* link in directory list        */
    char           Name[NameMax]; /* entry name                    */
    word           Type;          /* entry type                    */
    word           Flags;         /* flag word                     */
    Matrix         Matrix;        /* access matrix                 */
    Semaphore      Lock;          /* locking semaphore             */
    Key            Key;           /* protection key                */
    struct DirNode *Parent;       /* parent directory              */
    DateSet        Dates;         /* dates of object               */
    word           Account;       /* owning account                */
    word           Size;          /* object size                   */
    List           Contents;      /* whatever this object contains */
} ObjNode ;
```

Most of the fields are the same and have exactly the same meaning. Instead of a count of the number of directory entries, the **ObjNode** structure has a **Size** field which can be the current size of a file, the number of characters buffered up and to be read, or some other piece of information relevant to the server. There is a linked list at the end to hold information specific to the server. The three words making up the linked list can be used as a list, for example to hold various buffers. Alternatively, by using suitable casts, the words can be assigned different meanings appropriate to the server. In addition it is possible to make the **ObjNode** part of a larger structure, for example:

```
    /* data structure for an Analog to Digital Converter */
typedef struct ATOD_Node {
   ObjNode     ObjNode;
   WORD        *HardwareBase;
   WORD        Mode;
   BYTE        Buffer[256];
} ATOD_Node;
```

Since this data structure contains all the correct pieces of information at its start, it will be recognised by the Server library as an ordinary file or directory. Routines which need to access the hardware are passed the **ObjNode** part of the structure, and hence the whole data structure. **DirNode** and **ObjNode** structures can be readily initialised with the **InitNode** routine of the Server library. This performs the following:

1. The **Name**, **Type**, **Flags**, and **Matrix** fields are completed using the arguments provided.

2. The semaphore is initialised to **unlocked**.

3. A random number is generated for the key, allowing the Server library to produce unique capabilities which can be passed back to clients.

4. The date stamps for **creation**, **last modified**, and **last access** are all set to the current time.

5. The account and size (or number of entries for a directory) are set to 0.

The list node, the pointer to the parent directory, and the final linked list are not initialised. The first two will be completed when the object is inserted into a directory. The contents of the linked list is specific to the server and hence it is the responsibility of the server's code. In the **lock** server the initial directory tree contains a top-level directory **LockRoot** which is initially empty. Entries will be added to this directory as a result of incoming **Create** requests, and entries will be removed following **Delete** requests. Some servers can build their entire directory tree when they start up and this directory tree remains fixed. For example, the following code fragment could be used to initialise an analogue-to-digital server which has six ports.

```
static DirNode     ATODroot;
static ATOD_Node   Ports[6];
int main(void)
{ int i;
  ...
  InitNode((ObjNode *) &ATODroot, "ATOD", Type_Directory,
                       DefDirMatrix);
  InitList(&ATODroot.Entries);

  for (i = 0; i < 6; i++)
   { char buf[4];
     buf[0] = '0' + i; buf[1] = '\0';
     InitNode(&(Ports[i].ObjNode), buf, Type_Stream, 0,
              DefFileMatrix);
     Ports[i].Mode          = ATOD_Idle;
     Ports[i].HardwareBase  = (WORD *) 0x20000000 + (0x10 * i);
     Insert(&ATODroot, &(Ports[i].ObjNode), FALSE);
   }
  ...
}
```

The root of this server's directory tree is initialised as before. Next, the six directory entries **0**, **1**, and so on, are initialised and inserted into the directory. Once the server has started the dispatcher a client program can open a stream to **/ATOD/5** and access the fifth port of the device.

12.3.4 Registering the server

Servers must make themselves known to the system. Unless the system knows that a particular program is willing to provide a service, distributed searches for that server will fail and clients will have no way of accessing the server. To register, a server needs to create an entry in the name table of the current processor and this name must be associated with the message port which the server will use to accept incoming requests. This can be achieved with the following code fragment:

```
int main(void)
{ Object          *nametable_entry;

  { char            mcname[IOCDataMax];
    NameInfo        nameinfo;
    Object          *this_processor;

    MachineName(mcname);
    this_processor = Locate(Null(Object), mcname);

    nameinfo.Port      = server_port= NewPort();
    nameinfo.Flags    = Flags_StripName;
    nameinfo.Matrix   = DefNameMatrix;
    nameinfo.LoadData = NULL;

    nametable_entry = Create(this_processor, "lock", Type_Name,
            sizeof(NameInfo), (BYTE *) &nameinfo);
    if (nametable_entry eq Null(Object))
     { IOdebug("Lock: failed to install name table entry");
       return(1);
     }
    Close(this_processor);
  }
  ...
}
```

To create a name table entry the System library **Create()** routine must be used. This routine returns an **Object** pointer which must be remembered, to allow the server to exit cleanly. Server names must be created inside the processor, so it is necessary to determine the current processor name and to obtain another **Object** to use with the **Create()** routine. The name table entry to be created must have exactly the same name as the root directory entry, and it must be of type **Type_Name**. If the server is unable to create the name table entry, this usually means that the server is already running. This may be unexpected, so the system administrator is informed through an **IOdebug()** call.

The Helios Nucleus automatically installs name table entries for the servers **/ram**, **/pipe**, **/fifo**, **/socket**, and **/null**. The code for these servers will be loaded automatically when an attempt is made to access the server, and these servers should not install their own name table entries. The **initrc** file provides a command called **auto** which achieves the same thing for user-defined servers. Once a server has been registered with the system, clients can start sending requests to it. Hence the next thing that a server should do is to call a dispatcher and thus prepare to accept such messages.

12.3.5 The dispatcher

The Server library uses one other major data structure in addition to **ObjNode** and **DirNode**. The **DispatchInfo** structure informs the library which routines can be called to handle specific routines. For the **lock** server this structure would contain the following:

```
      static void do_open(ServInfo *);
      static void do_create(ServInfo *);
      static void do_delete(ServInfo *);

      static DispatchInfo LockInfo = {
        (DirNode *) &LockRoot,
        NullPort,
        SS_LockDevice,
        NULL,
      { NULL,         2000 },
      {
        { do_open,     2000 },
        { do_create,   2000 },
        { DoLocate,    2000 },
        { DoObjInfo,   2000 },
        { InvalidFn,   2000 },  /* ServerInfo */
        { do_delete,   2000 },
        { InvalidFn,   2000 },  /* Rename     */
        { InvalidFn,   2000 },  /* Link       */
        { DoProtect,   2000 },
        { DoSetDate,   2000 },
        { DoRefine,    2000 },
        { InvalidFn,   2000 },  /* CloseObj   */
        { DoRevoke,    2000 },
        { InvalidFn,   2000 },  /* Reserved1  */
        { InvalidFn,   2000 },  /* Reserved2  */
      }
};

int main(void)
{ ...
  LockInfo.ReqPort = server_port;
  Dispatch(&LockInfo);
  ...
}
```

The components of the **DispatchInfo** structure are as follows:

1. A pointer to the root directory for this server.
2. The message port to which incoming requests will be sent.
3. A subsystem code used to identify the source of error messages. This is described in detail in section 12.4.

4. A string. This can be used to indicate the name of the parent directory, the default being the processor running this server. Occasionally, a server may wish to install itself somewhere other than under the processor level, typically to provide a network-wide service. In such cases a string can be supplied to indicate the parent directory, and the name table entry must be created inside this parent rather in the current directory, although the use of this facility is not recommended.

5. A function pointer to handle incoming messages which do not belong to the acceptable set of GSP messages. Such private protocol messages are used occasionally to enable or disable debugging options in the server. In addition to the function pointer the structure must contain a **stacksize**: the **Dispatch()** routine will **Fork()** off a separate worker thread to handle each incoming request, and this thread must be allocated its own stack.

6. A table of routines to handle the various possible GSP requests, each with a stack size. Note that there are two spare slots at the end of the table to allow for future expansion to the GSP protocol, should this be required.

This server defines three routines of its own: **do_open()**, **do_create()**, and **do_delete()**. Default Server library routines are used to handle incoming **Locate**, **ObjectInfo**, **Protect**, **SetDate**, **Refine** and **Revoke** messages. The **InvalidFn()** routine is also part of the Server library, and simply sends back an error code indicating that this server does not support the request. Hence the lock server does not support the renaming of locks, it provides no useful information as a result of a **ServerInfo** request, it cannot contain symbolic links, and it does not use the **CloseObj** request.

The dispatcher works as follows:

1. While the server's message port remains valid, it allocates a **MsgBuf** buffer suitable for an incoming GSP request. This buffer has a data vector of **IOCDataMax**, set to 512 bytes. The General Server Protocol defines this to be the largest message that can be sent directly to a server. Should an application manage to send more data somehow, the server's memory would be corrupted and the processor is likely to crash. The dispatcher waits for a message to be sent to the server message port.

2. When a message arrives, the server checks the request code. If it is not an accepted GSP message, it will examine the private function part of the **DispatchInfo** structure. Otherwise it will look at the appropriate entry of the GSP routines table. From this it can extract a stack size and **Fork()** off a worker thread to handle this particular request.

3. Once the worker thread has started, the dispatcher will allocate another message buffer and accept the next request. This ensures that there is a very small delay between accepting requests from several different clients, and the Kernel does not need to buffer such requests for any length of time.

4. The worker initialises a **ServInfo** structure which is used to monitor the way this request is being handled. For example, as the worker thread walks down the

server's directory tree to reach the target object the **ServInfo** structure is updated to monitor the current position.

5. The worker checks the validity of the message, and in particular it checks that the capability supplied by the client is valid.

6. If the message appears valid, the appropriate routine in the **DispatchInfo** structure will be called with the **ServInfo** structure as argument. This routine may be one of the default routines in the Server library or it may be code in the server itself.

7. Once the routine specified in the **DispatchInfo** structure has finished, the request has been handled and, after some tidying up, the worker thread terminates.

For example, suppose a client attempts to delete the object **/Net/00/lock/dbase.table12**. The System library routine **Delete()**, called explicitly or through a higher-level routine such as Posix **unlink()**, will produce a GSP message with the function code for **Delete** and pass this message to the system. The message may have to be routed through various processors to find the one running the server, but eventually it will arrive at the **lock** server's message port where the dispatcher accepts it. The dispatcher checks that the message is a valid GSP request, it checks the table to determine the stack size for **do_delete()**, and it will **Fork()** off a worker to handle this request. The worker performs some further validation checks and then calls the routine specified in the table, **do_delete()**. The server code can now handle this request, generate a reply, and return. The worker performs some tidying up and terminates. Meanwhile the dispatcher continued running and may have received a number of other messages, starting a separate thread for each of them.

12.3.6 Cleaning up

The dispatcher will only return if the server's main message port becomes invalid, usually as a result of a user request to terminate the server. Different servers have different ways of being terminated:

1. Some servers cannot be terminated at all. For example, there is no way of terminating the **/tasks** and **/loader** server in a processor.

2. Some servers have special commands which can be used to terminate them. For example, the Network server and Session Manager can be terminated only by using the **stopns** command. Such commands normally make use of private protocol messages.

3. Some servers can be terminated by sending them a signal. This is possible only if the server is an ordinary C program because the signal handling is usually done through the Posix routine **signal()**. The **kill** command can be used to send signals to such servers.

4. With some servers it is possible to delete the whole server, for example **rm /lock**. This is the mechanism supported by the lock server. It will only work if the root directory of that server is currently empty.

Normally the request to terminate is received by a separate thread within the server, either a signal handling thread or a worker thread **Fork()**ed off by the dispatcher. It is conventional for this thread merely to abort the dispatcher's message port, forcing the dispatcher to terminate, and letting the **main()** routine perform any cleaning up required. Most of this cleaning up happens automatically when the server program terminates: for example, any allocated parts of memory are released; owned message ports are freed; any threads that are still running are aborted. However, the system does not clean up everything and servers may need to do the following jobs:

1. The name table entry for the server must be deleted explicitly. Unless the entry is deleted it will not be possible to start another server with the same name. The following code fragment illustrates how this can be achieved:

```
int main(void)
{ ...
    /* Dispatch will return ifœ the server_port */
    /* has been aborted.                       */
  Delete(name, Null(char));

  return(0);
}
```

2. If the server interacts with a piece of hardware, this may need to be shut down. For example, a file server may need to sync the disc and park the heads. This is specific to the server.

3. If the event handling mechanism has been used to handle interrupts, it is necessary to use **RemEvent()** to release the event handler.

4. If a dumb link has been allocated, it must be released.

12.3.7 Using the lock server

So far the content of this section has been general. With only a small number of changes the same code can be used in most servers. The subsections to follow describe the pieces of code specific to this server. It is useful to look at the client side first of all. The following two routines can be used to create and remove locks.

```
typedef Object *Lock;
#define NullLock ((Lock) NULL)

Lock  GetLock(char *name)
{ Object *lock_server = Locate(Null(Object), "/lock");
  Lock   result;

  if (lock_server == Null(Object))
   { fputs("GetLock: fatal error, lock server is not running.\n",
           stderr);
     exit(EXIT_FAILURE);
   }
  result = Create(lock_server, name, Type_Stream, 0, Null(BYTE));
  Close(lock_server);
```

```
  return(result);
}

void FreeLock(Lock lock)
{ Delete(lock, Null(char));
  Close(lock);
}
```

The first routine tries to create a named lock, for example, **dbase.table12**. If it cannot create this lock it will return the value **NullLock**, typically causing the client to back off and try again some time later. The **lock** server requires no extra information when creating an object, so the final two arguments to the **Create()** call are null. The second routine removes the lock. The following code fragment illustrates how these could be used in a real program.

```
void manipulate_table(int table_number)
{ char buffer[Name_Max];
  Lock lock;

  sprintf(buffer, "dbase.table%d", table_number);
  while ((lock = GetLock(buffer)) == NullLock)
   Delay(OneSec);

  /* Manipulate the table */

  FreeLock(lock);
}
```

In addition to the **Create** and **Delete** requests which could be sent by the routines shown above, it is desirable to be able to examine the current state of the lock directory. For example, use the command **ls -l /lock** to determine which locks are currently in place, particularly when debugging clients that may or may not be leaving locks in place by accident. To examine a directory it is necessary to open that directory and read data from it. Hence the three requests that the lock server must accept are **Open**, **Create**, and **Delete**. The nature of these requests is such that the Server library cannot provide sensible default routines for them. For example, when a file in a RAM disc is deleted it is necessary to release the associated memory. The Server library does not know that the server is a RAM disc, nor how the data is held, so it would have no way of releasing this memory in a **DoDelete()** routine.

12.3.8 The Open routine

When the **Open** request arrives at the server, the dispatcher will **Fork()** off a worker thread to handle the request. This worker will perform some handling before invoking the server specific routine **do_open()**. The exact details of this are shown below.

1. A **ServInfo** structure will be allocated. This is used to hold information about the request as it is handled.
2. A **longjmp** buffer **Escape** in this structure will be initialised. This allows the user code to **longjmp()** back to the worker thread at any time, used mainly in emergencies to abort the handling of this request.

3. A pointer to the MCB holding the incoming request is put into the **ServInfo** structure. This allows the server specific code to examine the initial request.

4. A copy of the request code is placed in the **ServInfo** structure and the **FnRc** field of the MCB is overwritten with the subsystem code defined in the **DispatchInfo**. This makes it easier to send back error codes.

5. The worker routine will begin walking down the directory tree until it reaches the **Context** object. For file I/O this is usually the current directory. For the locking routines described earlier the Context object is **/lock**, the root of the directory tree. Once the context has been reached the Server library will check that the capability provided in the message was valid. Note that the Context object may have been reached, and hence the capability checked, before the message arrived at the server: the Context might have been the processor or even the root of the whole naming tree. In such a case the search will halt at the root of this server rather than at a subdirectory.

6. As the worker thread walks down the directory hierarchy it will fill in the **Pathname** field of the **ServInfo** structure. This is used to generate replies.

7. When the search ends, either at the root directory for this server or at a subdirectory or object lower down, the specified object is locked. This prevents other threads from modifying the directory tree at this point, but not at other parts of the naming tree.

8. The server specific routine will be called with a single argument, a pointer to the **ServInfo** structure.

The start of the **do_open()** routine would typically look like this:

```
static void do_open(ServInfo *servinfo)
{ MCB            *m          = servinfo->m;
IOCMsg2        *req        = (IOCMsg2 *)(m->Control);
byte           *data       = m->Data;
char           *pathname   = servinfo->Pathname;
DirNode        *d;
ObjNode        *f;

  d = (DirNode *) GetTargetDir(servinfo);
  if (d == Null(DirNode))
   { ErrorMsg(m, EO_Directory); return; }

  f = GetTargetObj(servinfo);
  if (f == Null(ObjNode))
   { ErrorMsg(m, EO_File); return; }

  ...
}
```

The user routine starts by extracting various pieces of information from the **ServInfo** structure. The MCB **m** contains the incoming request, apart from the **FnRc** field which has been overwritten with the subsystem code: the original **FnRc** will have been saved

in the **ServInfo** structure. Given the MCB it is possible to extract the control and data vectors of the message. The control vector will contain a data structure defined by GSP, in this case containing general information plus the mode used to open the file or directory. The **Pathname** field should now contain something like **/Net/00/lock**, the path name for the context object or the root directory of this server.

The routine **GetTargetDir()** returns the parent directory for the target object, updating various fields such as the current pathname if necessary. If there is no parent directory or if the client does not have access to the parent directory, the **FnRc** field of the MCB will have been completed with part of a suitable error code: the **ErrorMsg()** routine uses **or** in the second arguments and sends the error message to the client. There is a special case for the root of the server's directory tree: if this is the target object, **GetTargetDir()** will return the root **/lock**, not the processor **/00** which is its true parent.

For example, imagine that a client is attempting to access a file in the **include** disc server **/Net/00/include**. The client has a current directory of **/Net/00/include**, which is the context object. The file to access is **sys/errno.h**, so the subdirectory **sys** is the parent directory for the target object. When **do_open()** is entered, the pathname would be **/Net/00/include** and the root of the server would have been locked. After the call to **GetTargetDir()** the pathname would have been updated to **/Net/00/include/sys**, the **sys** subdirectory would have been locked, and the root directory would have been unlocked. This prevents any other thread from deleting the **sys** subdirectory while this thread is active.

The next set of lines go down the directory tree for one more step, to reach the actual target object. Again the pathname is updated, the target object is locked, and the parent directory is unlocked. Other threads can now add entries to, or remove entries from the parent directory, but they cannot manipulate the target object of the request. The value of **f** will be an **ObjNode** or a **DirNode** pointer created by the server. It could be the root directory of the server, a subdirectory, or an object within a directory. These various possibilities do not cause any problems for the Server library since the two data structures are almost identical. The **Type** field of the structure identifies whether it is a directory or not.

The **lock** server allows directories to be read but it is not possible to open locks: as far as this server is concerned reading from or writing to a lock is not sensible, so any attempt to open a lock should give an error message.

```
void do_open(ServInfo *servinfo)
{ ...
  if (f->Type != Type_Directory)
   { ErrorMsg(m, EC_Error + EG_WrongFn + EO_Object);
     return;
   }
  ...
}
```

The request has now been validated so it is necessary to send back a reply to the client. The structure of this reply is again determined by the GSP protocol. It could be constructed manually, but for convenience the Server library provides a routine to do it.

```
void do_open(ServInfo *info)
```

```
{ MsgBuf      *r;
  Port        reqport;
  ...
  r = New(MsgBuf);
  if (r == Null(MsgBuf))
   { ErrorMsg(m, EC_Error + EG_NoMemory); return; }

  FormOpenReply(r, m, f, Flags_Server | Flags_Closeable, pathname);
  reqport = NewPort();
  r->mcb.MsgHdr.Reply = reqport;
  PutMsg(&r->mcb);
  Free(r);
  ...
}
```

The **FormOpenReply()** routine requires a second message buffer, so this is allocated. Various fields, such as the pointers to the control and data vectors, are completed automatically by **FormOpenReply()**. Since the request involves establishing a stream connection the client will need a message port to communicate directly with this thread, so another port is allocated and put into the message header. The message is then sent back to the client, and the second message buffer can be released again.

Note that no attempt is made to check whether or not the reply message arrived safely back at the client. This is not necessary: if the reply message is lost, the client will time out and simply repeat the **open** request, starting up another thread. This thread will fail to receive any messages on its message port and can go away after a while: if necessary the client will re-open the connection.

The **do_open()** routine should now go into a loop, handling incoming requests such as **Read**, **Seek**, and **Close**, which can be sent directly to open streams. In practice it is known that the object that has been opened is a directory, and as far as the Server library is concerned all directories look the same: a linked list of **DirNode** and **ObjNode** structures. Hence the Server library provides another routine to handle the **Read** requests and so on, which terminates when the client closes the connection.

```
void do_open(ServInfo *servinfo)
{ ...
  DirServer(servinfo, m, reqport);
  FreePort(reqport);
  return;
}
```

Obviously if the server does support **Read**, **Write**, and other operations on objects, a great deal more work will be required here. When the **do_open()** routine returns, the worker thread tidies up, for example, releasing any locks that have been left, and terminates.

12.3.9 The Create routine

The next routine to consider is responsible for creating locks. The start of the routine is similar to the **do_open()** routine.

```
static void do_create(ServInfo *servinfo)
{ MCB       *m        = servinfo->m;
  MsgBuf    *r;
  DirNode   *d;
  ObjNode   *f;
  IOCCreate *req      = (IOCCreate *)(m->Control);
  char      *pathname = servinfo->Pathname;

  d = GetTargetDir(servinfo);
  if( d == Null(DirNode))
   { ErrorMsg(m,EO_Directory); return; }

  f = GetTargetObj(servinfo);
```

Again, various pieces of information about the request are obtained through the **ServInfo** structure and an attempt is made to walk down the directory tree to reach the target object. However, at this point the correct behaviour is different. If an attempt is made to open a non-existent object, this is usually an error. (Servers may support the **O_Create** flag for an implicit **create**.) Only existing objects may be opened. When creating a lock an error occurs if the target object already exists: another application has already claimed the named lock, so this attempt to create a lock must fail.

```
static void do_create(ServInfo *servinfo)
{ ...
  if (f != Null(ObjNode))
   { ErrorMsg(m, EC_Error + EG_InUse + EO_Name);
     return;
   }

   /* Reset FnRc field, changed by failure of GetTargetObj() */
  m->MsgHdr.FnRc = SS_LockDevice;
  ...
}
```

The request has now been validated so it is necessary to add a new lock to the directory. This involves allocating a new **ObjNode** structure, initialising it, and inserting it into the directory.

```
static void do_create(ServInfo *servinfo)
{ ...
  f = New(ObjNode);
  if (f == Null(ObjNode))
   { ErrorMsg(m, EC_Error + EG_NoMemory + EO_Object);
     return;
   }
  InitNode(f, objname(pathname), Type_Stream, 0, DefFileMatrix);
  Insert(d, f, TRUE);
  ...
}
```

The type, flags, and matrix arguments passed to **InitNode()** are described in the next section. The routine called **objname()** is a useful little routine which takes a full path name, for example **/Net/00/lock/dbase.table12**, and it extracts the final part of the path name, **dbase.table12**. The **Insert()** routine takes three arguments: a directory,

a new object to put into the directory, and a flag to indicate whether or not the directory is currently locked. The **GetTargetObj()** routine failed in this case, so the appropriate parent directory has remained locked.

It is fairly easy to support the creation of subdirectories as well as locks. This would allow a collection of programs such as a database package to create a subdirectory when they start up, and to keep all their locks in this subdirectory. Hence these programs are less likely to interfere with any other programs using the **lock** server, by accidentally using the same names for locks. The following code fragment shows how subdirectories can be included.

```
static void do_create(ServInfo *servinfo)
{ ...
  if (req->Type == Type_Stream)
   {
      /* Create a lock as before */
   }
  else
   { DirNode *subdir = New(DirNode);
     if (subdir == Null(DirNode))
      { ErrorMsg(m, EC_Error + EG_NoMemory + EO_Directory);
        return;
      }
     InitNode((ObjNode *)subdir, objname(pathname),
              Type_Directory, 0, DefDirMatrix);
     InitList(&subdir->Entries);
     Insert(d, (ObjNode *) subdir, TRUE);
     f = (ObjNode *) subdir;
   }
  ...
}
```

Note that the initialisation of the linked list of directory entries must be done explicitly. It is not done automatically by the Server library. Finally it is necessary to send back a reply to the client indicating that the operation has been successful.

```
static void do_create(ServInfo *servinfo)
{ ...
  r = New(MsgBuf);
  if (r == Null(MsgBuf))    /* %$#@! */
   { Unlink(f, TRUE);
     Free(f);
     ErrorMsg(m, EC_Error + EG_NoMemory + EO_Message);
     return;
   }

    /* A kind of magic, see next section */
  req->Common.Access.Access = AccMask_Full;

  FormOpenReply(r, m, f, 0, pathname);
  PutMsg(&r->mcb);
  Free(r);
}
```

Note that the **create** operation may fail at the last moment because the server is running low on memory. Since processors running Helios typically have no virtual memory support this is an ever-present possibility that must be allowed for. Usually it is possible to fill in a reply message with little effort, using the Server library, and this reply message can be sent off. A **Create** request does not involve opening a stream connection so there is no need to include a message port in the reply message.

There is a problem if the reply message is lost. The lock has now been created so no application can create the same lock. Hence if the client tries to repeat the request it will fail the second time around, and the application will hang. Given that message passing is inherently unreliable such problems are unavoidable, and there is a small but finite possibility that applications will fail as a result.

12.3.10 The Delete routine

The final routine which the **lock** server must provide is for **Delete** requests. Again, the start of the routine is fairly similar.

```
static void do_delete(ServInfo *servinfo)
{ MCB       *m   = servinfo->m;
  ObjNode   *f;
  IOCCommon *req = (IOCCommon *)(m->Control);

  f = GetTarget(servinfo);
  if (f == Null(ObjNode))
   { ErrorMsg(m, EO_File); return; }

  ...
}
```

Again, details of the request are extracted from the **ServInfo** structure. This routine, instead of calling **GetTargetDir()** and **GetTargetObj()**, calls **GetTarget()**, an alternative Server library routine. This alternative simply combines the other two, returning NULL if either the parent directory or the target object does not exist. If desired, this routine could have been used in the **do_open()** routine above.

The object to be deleted can be a file, a subdirectory, or the root directory. Directories can only be deleted if there are no entries. Deleting the root directory is special: it is the means used to terminate the whole server.

```
static void do_delete(ServInfo *servinfo)
{ ...
  if (f->Type == Type_Directory)
   { DirNode *d = (DirNode *) f;
     if (d->Nentries != 0)
      { ErrorMsg(m, EC_Error + EG_InUse + EO_Directory);
        return;
      }
   }

  if (f == (ObjNode *) &LockRoot)
   { ErrorMsg(m, Err_Null);    /* send back a success code */
     AbortPort(LockInfo.ReqPort, EC_Fatal + EG_Exception + EE_Abort);
   }
```

```
  else
   { Unlink(f,FALSE);
     servinfo->TargetLocked = FALSE
     Free(f);
     ErrorMsg(m,Err_Null);
   }
}
```

According to the General Server Protocol the reply to a **Delete** request is simply success or failure, not a full message. Hence there is no need to allocate another message buffer for the reply. The **AbortPort()** routine will terminate the dispatcher, thus activating the tidying-up code in **main()**. Note that the root directory will have been checked to ensure that the server does not terminate while there are still outstanding locks and, presumably, clients of this server. If a lock or a subdirectory is to be deleted, it must first be removed from its parent directory: this parent is not currently locked, so the second argument is FALSE. Next the **ServInfo** structure is updated: the target object no longer exists, so the worker thread should not unlock it when the **do_delete()** routine returns. Finally the piece of memory used to hold the lock or subdirectory can be freed, and the operation has finished.

12.4 More details

The previous two sections have given information about servers, including the source code for a simple **lock** server. At times certain details were simplified to avoid making the descriptions even more complicated than they already are. This section provides more detailed explanations.

This section is divided into two subsections: protection, giving details of those aspects of protection a server needs to worry about; and a brief description of the various Server library fields and more information on some of the available data structures.

12.4.1 Protection

This subsection explains how the Helios protection mechanism (described in detail in chapter 14, *Protection*) affects the writing of servers. It is assumed that the reader has some understanding of access matrices, and in particular that the terms, V access, X access, Y access, and Z access are familiar.

Associated with every file and directory within a server are two fields: an encryption key and an access matrix. These are held in the **ObjNode** and **DirNode** data structures. The encryption key is secret, known only to the server, and must never be made known to users or to other programs. If the encryption key associated with an object becomes known, that object is effectively unprotected. Encryption keys are usually generated automatically by the Server library in the call to **InitNode()**, and are suitable random numbers.

When a client attempts to access an object, it does so relative to a context object, which is typically the current directory. It will already have a capability for this context object. When a user logs in, that user's session is given capabilities for various objects including the user's home directory and the window server. These capabilities are inherited by any programs running inside the session. The capability defines the

access allowed to the context object, using an eight-bit access mask. This access mask contains bits for: **read** access; **write** access; **delete** access to remove the object; **alter** access to change the protection associated with an object; and **V X Y** and **Z** access for directories. Some objects may also support **execute** access if the object is an executable file. If the appropriate bit in the access mask is set, the capability grants that access to the context object. This access mask is encrypted within the capability using the encryption key stored by the server.

For example, a user attempts to access a file relative to his or her home directory. The message includes the home directory as a context object, and a capability for that directory. Since it is the user's home directory, that user is likely to have **read**, **write**, and **alter** access to the directory itself, and **V** access to get to the contents of the directory. The user is unlikely to have **delete** access to the home directory, since deleting a user is usually only possible for the file system administrator. When the file server receives the message, the dispatcher will **Fork()** off a worker thread, which follows the directory tree as far as the context object. It then decrypts the capability with the encryption key held in the appropriate **DirNode** or **ObjNode** structure to validate the capability. If some other user is attempting to fake access to this directory, by inventing a capability, the decryption will fail and an error message will be generated automatically.

The object actually being accessed is not the context object but something relative to it. The request will involve working down the directory tree from the context object, until the target directory and object are reached. During this process, the access mask sent with the request is updated using the access matrices held with the intermediate **DirNode** structures and the final **ObjNode**. When the final object has been reached, the message contains the actual access permitted to the object which this client is allowed to have.

Three requests involve sending back capabilities: **Locate**, **Open**, and **Create**. These capabilities are generated automatically by the **FormOpenReply()**, using the current access mask and the encryption key associated with the object. There is a special problem for **Create**: the target object which should be created does not exist, so the current access mask refers to the parent directory and not to the object; this must be changed to full access for the reply message, since the client is now the owner of the new object.

```
static void do_create(ServInfo *servinfo)
{ ...
  req->Common.Access.Access = AccMask_Full;
  FormOpenReply(r, m, f, 0, pathname);
  PutMsg(&r->mcb);
  ...
}
```

There are three requests to modify the protection. A **Protect** message is used to change the access matrix on a directory or an object. A **Refine** message takes an existing capability and returns a reduced capability which can be passed to other users or programs, allowing these other users some access to the object but not necessarily as much as the owner. A **Revoke** message changes the encryption key associated with an object, thus invalidating any existing capabilities.

In writing servers, the programmer must do two things. Firstly, it is necessary to put suitable access matrices in every **DirNode** and **ObjNode** when the object is created. Once the object has been created these access matrices are usually manipulated automatically through the **DoProtect()**, **DoRefine()** and **DoRevoke()** routines in the Server library, and the programmer need not worry about these any further. Secondly, the server code must check the current access mask before performing operations, to ensure that the client is actually permitted to perform the operation.

Choosing access matrices

One of the arguments to the **InitNode()** routine is the access matrix to put into the **DirNode** or **ObjNode** structure. The correct choice of access matrix is usually simple. The header file **protect.h** defines two default access matrices **DefDirMatrix** and **DefFileMatrix**, which suffice for the majority of servers. Some applications may need different access matrices, but these can be installed after the object has been created using the **Protect()** routine. For most servers there is little or no point in attempting to guess the access rights which the client wants the object to have.

DefDirMatrix is defined as *0x21134BC7*, or *darwv:drwx:rwy:rz*. Note that the matrix is defined for little-endian processors, so the owner category offering the most access **C7** is actually the last byte. This access matrix simply propagates the current permissions to the lower levels, so if a client currently has X access, it will continue to have X access after going through this directory. The owner of the object is allowed to delete, alter, read, and write the directory. Other users are given less access, depending on their current category.

DefFileMatrix is defined as *0x010343C3*, or *darw:drw:rw:r*. There are no access bits for V, X, Y and Z access since those only refer to directories. The owner of the file is allowed to delete it, alter the access permissions, and read and write it. Most users are only allowed to read the file.

As an example of a server for which the default access matrices are not suitable, consider a read-only filing system such as a ROM disc. For such a server nobody can write to the filing system, simply because that is the way the hardware works. Hence all **write** access must be removed, both on files and on directories. **Write** access includes **delete** and **alter**, because these involve writing to the directories. Hence directories should be given the access matrix *0x21110905* or *rv:rx:ry:rz*. Files should be given the access matrix *0x01010101* or *r:r:r:r*. If it is desirable to be able to terminate the server by deleting the root of its directory tree, the root must be given a different access matrix to allow deletions: *0x21114945* or *drv:drx:ry:rz*.

Another example could be a variant of the **lock** server. In the source shown **DefDirMatrix** and **DefFileMatrix** were used, so any client with **X** access has the right to delete locks. This may be undesirable. An alternative would allow only the program which created the lock to delete it. This program automatically has V access to the lock because it created it, so the correct access matrix for a lock would now be: *0x010343C3* or *darw:drw:rw:r*. Also, since a lock cannot be read or written it may be desirable to eliminate **read** or **write** access to the locks, but not to the directories since these must still be readable. Hence the access matrix for a lock would now be: *0x000000C0*, or *da:::*.

In practice the **lock** server would work well without the above modifications, and

choosing access matrices other than the default ones provides little or no benefit for the average user.

Testing access rights

It is mainly the responsibility of the server code to check that the requested operation is allowed. The Server library guarantees that the capability passed in the message was valid and hence that the client has the access currently held in the access mask. The rest has to be done in the server code. The following code fragments indicate how this can be done.

```
static void do_create(ServInfo *servinfo)
{ int       access;
  MCB       *m   = servinfo->m;
  IOCCreate *req = (IOCCreate *) m->Control;
  DirNode   *d;
  ObjNode   *f;

  d = GetTargetDir(servinfo);
  if (d == Null(DirNode))
   { ErrorMsg(m, EO_Directory); return; }

   /* Remember the access to the parent directory */
  access = req->Common.Access.Access;

  f = GetTargetObj(servinfo);
  if (f != Null(ObjNode))
   { ErrorMsg(m, EC_Error + EG_InUse + EO_Name); return; }

   /* Reset the FnRc field after failure of GetTargetObj()  */
  m->MsgHdr.FnRc = /* subsystem code */;

   /* Check that this client has write access to the parent */
  unless(CheckMask(access, AccMask_W))
   { ErrorMsg(m, EC_Error + EG_Protected + EO_Directory);
     return;
   }

   /* Reset the current access mask to full access          */
  req->Common.Access.Access = AccMask_Full;

   /* Create the object, do a FormOpenReply(), and reply    */
   /* to the client.                                        */
  ...
}
```

When creating a new object the important access rights are those for the directory which will contain the new object, not for the object itself since that does not exist yet. Hence the current access mask is saved after the call to **GetTargetDir()**. The call to **CheckMask()** checks that the current access mask allows all of the requested operations, in this case just writing, but several access bits can be checked in a single call. The client automatically becomes the owner of the new object so it must be given full access. Otherwise it is possible to create objects which cannot be deleted.

Some servers allow objects to be created with the same name as an existing object. For example, the **/window** server will accept several requests to create a window **Shell**. The first **Create** will produce **/window/Shell**, the second **/window/Shell.1**, and so on. Such servers need slightly different code in the **do_create()** routine.

```
static int next_window = 1;

static void do_create(ServInfo *servinfo)
{ ...
  f = GetTargetObj(servinfo);
  if (f != Null(ObjNode))
   {  /* Add a number to the current pathname */
     strcat(pathname, ".");
     addint(pathname, next_window++);

      /* Reset the current target back to the parent directory */
     UnLockTarget(servinfo);
     servinfo->Target = (ObjNode *d);
     LockTarget(servinfo);
   }
  ...
}
```

The **addint()** routine adds a number to the current pathname, the last bit of which will be used when creating the new object. In addition it is necessary to switch back to the parent directory and lock that.

The code for a **do_delete()** routine will look something like this.

```
static void do_delete(ServInfo *servinfo)
{ MCB       *m   = servinfo->m;
  IOCCommon *req = (IOCCommon *) m->Control;
  ObjNode   *f;

  f = GetTarget(servinfo);
  if (f == Null(ObjNode))
   { ErrorMsg(m, EO_File); return; }

  unless(CheckMask(req->Common.Access.Access, Accmask_D))
   { ErrorMsg(m, EC_Error + EG_Protected + EO_File);
     return;
   }
  ...
}
```

The routine merely checks whether or not the client has **delete** access to the specified object. Similarly the **Protect** routine would check whether or not the client has **Alter** access, and a **Locate** routine should check that the client has some access. For an **Open** request the situation is slightly more complicated because **open** could be used for **read**-only, for **write**-only, or for both **read** and **write**. Also, if the **O_Create** bit is supported, it may be necessary to check that the client has **write** access to the parent directory.

```
static void do_open(ServInfo *servinfo)
{ MCB     *m    = servinfo->m;
  IOCMsg2 *req  = (IOCMsg2 *) m->Control;
  DirNode *d;
  int     dir_access;
  ObjNode *f
  int     file_access;

  d = GetTargetDir(servinfo);
  if (d == Null(DirNode))
   { ErrorMsg(m, EO_Directory); return; }
  dir_access = req->Common.Access.Access;

  f = GetTargetObj(servinfo);
  if (f == Null(ObjNode))
   { if ((req->Arg.Mode & O_Create) == 0)
      { ErrorMsg(m, EO_File); return; }
     m->MsgHdr.FnRc = /* subsystem code */

     unless(CheckMask(dir_access, AccMask_W))
      { ErrorMsg(m, EC_Error + EG_Protected + EO_Directory);
        return;
      }

       /* Create and insert new object */
     f = /* new object */

       /* client is owner, so has full access */
     req->Common.Access.Access = AccMask_Full;
   }
  else
   { int mode = req->Arg.Mode & (O_Create + O_Exclusive);
      /* create+exclusive -> file must not yet exist ! */
     if (mode == (O_Create + O_Exclusive))
      { ErrorMsg(m, EC_Error + EG_InUse + EO_Name);
        return;
      }
   }

  file_access = req->Common.Access.Access;

    /* Check for a truncate request */
  if (req->Arg.Mode & O_Truncate)
   { unless(CheckMask(file_access, AccMask_W))
      { ErrorMsg(m, EC_Error + EG_Protected + EO_File);
        return;
      }
     /* truncate the object, if that makes sense */
   }

    /* Check that the requested Open mode is allowed. */
    /* This involves checking the bottom nibble only. */
  unless(CheckMask(file_access, req->Arg.Mode & Flags_Mode))
```

```
   { ErrorMsg(m, EC_Error + EG_Protected + EO_File);
     return;
   }

  /* Use FormOpenReply() and acknowledge the request. */
  ...

  forever
   { word errcode;
     m->MsgHdr.Dest    = stream_port;
     m->Timeout        = StreamTimeout;
     m->Data           = data;

     UnLockTarget(servinfo);
     errcode = GetMsg(m);
     m->MsgHdr.FnRc    = /* subsystem code */
     LockTarget(servinfo);

     /* validate the request code */

     switch(errcode & FG_Mask)
      { case FG_Read   :
            unless(CheckMask(file_access, AccMask_R))
             { ErrorMsg(m, EC_Error + EG_Protected + EO_File);
               break;
             }
            /* Handle read request */

        case FG_Write  :
            unless(CheckMask(file_access, AccMask_W))
             { ErrorMsg(m, EC_Error + EG_Protected + EO_File);
               break;
             }
            /* Handle write request */
          ...
      }


   }
}
```

This code fragment illustrates several more aspects. First the **open** mode can be quite complicated. There are **create**, **exclusive**, and **truncate** bits, which must be allowed for. Some servers may also need to take the **O_NonBlock** mode into account, which means that any **read** or **write** requests should be answered immediately and the server should not wait for data from the device before replying. The creation of a new object requires **write** access to the parent directory, and the truncation of an object requires **write** access to the object itself. If, for example, the client is trying to open the object in write-only mode when the client does not have **write** access, the **Open** request can fail immediately. It is conceivable that a client could open a stream in read-only mode and then attempt to write to it, so subsequent requests must also be checked.

The code fragments for **Create**, **Delete**, and **Open** requests illustrate fairly well the sort of checks that servers have to perform for different operations. Handling other requests such as **Locate** and **ObjectInfo** involves only minor variations.

12.4.2 The Server library

The Server library provides a considerable number of facilities that have not yet been described. This subsection gives an outline of all the available routines, plus some more information on several of the data structures. Full details on individual routines can be found in *The Helios Encyclopaedia* or by using the on-line help system.

Object types

Every object controlled by a server has a type, for example **Type_File** or **Type_Directory**. This type consists of two separate fields. The bottom nibble is used to identify the general nature of the object, whether it is a directory containing other objects, whether it behaves like a file or a device, or whether it supports some other protocol.

```
#define Type_Flags        0x0F   /* a mask */
#define Type_Directory    1
#define Type_Stream       2
#define Type_Private      4
```

All Helios objects are supersets of the above three basic types. In the previous section, the individual lock **ObjNode** structures were assigned the type **Type_Stream**. In practice, since locks cannot be opened and read like files it would have been better to make them supersets of **Type_Private**.

```
#define Type_Lock    (0x120 | Type_Private)
```

The remainder of the **type** field gives further information about the object, but this is rarely used. For example, the header file **gsp.h** defines data types for networks and for task forces which are both supersets of **Type_Directory**. User servers should avoid re-using any existing object types for new devices, and in particular **Type_Link**, **Type_Pipe**, **Type_Socket**, and **Type_Pseudo** are interpreted in a special way by the system.

Object flags

Associated with every object controlled by a server is a set of flags. This informs the System library as to how the object can be manipulated. These flags are returned in the directory entries when the parent directory is read. The flags are also returned as a result of the routine **FormOpenReply()**, together with some extra flags passed as an argument to that routine. At present the following flags are relevant to user servers. The full list can be found in the header file **syslib.h**.

More implies that more information can be obtained about the object by sending an ObjectInfo request. Failing to set this bit will not prevent clients from sending ObjectInfo requests, but some client programs may take account of this flag.

Seekable means that if a client opens a stream to this object, this stream can handle **Seek** requests. Typically this is true only for files, since seeking within an RS232 line, for example, does not make much sense.

NoIData is used with the **Write** protocol. When writing small amounts of data, less than 512 bytes, the System library can normally send the data with the **Write** request. For larger amounts of data the System library must send a preliminary message requesting that the server allocate suitable buffer space before the data is actually sent. If a server sets this flag, the System library will always use the second protocol, even for amounts of data less than 512 bytes. Typically this flag might be used by a file server which wants full control over the incoming messages to optimise its use of the cache.

CloseOnSend can be used by servers which do not want more than one client to have stream access to a given object. Normally when a program spawns another child program, typically using **vfork()** and **execve()** in the Posix library, the child inherits all open streams. Hence if the parent has an open stream to this server, the child will also have an open stream, and in theory both programs can now start interacting with the server. There will be a separate **Open** request resulting in a second worker thread, but both threads are operating on the same object. For most servers this is acceptable since the correct locking mechanisms are in place. Occasionally a server may be unable to cope with this, in which case setting the **CloseOnSend** will cause the stream in the parent program to be closed when the child is spawned.

OpenOnGet Helios uses lazy evaluation on streams inherited from the parent through the environment. For example, a program's standard error stream is not opened until the client actually attempts to write data to this stream. If a server needs to know immediately when a program inherits a stream, it can set the **OpenOnGet** flag: as soon as the child receives its environment there will be an **Open** request sent to the server.

Selectable should be set if the server supports **Select** requests on a stream. This is for compatibility with old servers which may not support **Select**.

Interactive is set on interactive streams such as windows and serial ports. It is used fairly often. For example, the C library uses this flag to determine the buffer size for C I/O: for non-interactive streams the buffer size is usually 1024 bytes and data is held until the whole buffer is full; for interactive streams a smaller buffer is used and data is flushed when a linefeed character is written.

MSdos indicates that the file server uses MS-DOS format for text files, with both carriage return and linefeed characters marking the end of a line instead of a single linefeed. Helios is defined to use linefeed characters only, not carriage return/linefeed. This flag is interpreted by Language libraries such as the C library, which will automatically convert to the Helios form on input and to the MS-DOS form on output.

Extended indicates that the server supports an extended protocol for **read** operations. If this flag is set, the System library will cause another message transaction once

all the data has been received from the server, as a way of informing the server that the data has arrived safely and can be discarded if appropriate.

NoReOpen can be set if the server does not support the re-opening of stream connections to cope with a broken route. Such servers are not inherently fault tolerant.

Fast indicates that the server guarantees a rapid response to all requests, and hence that the client side can use a relatively short timeout rather than the default ones of (typically) twenty seconds.

Closeable means that the client side should send a message to the server when a stream or object is closed.

Server is used together with the **Closeable** flag to work out what kind of **Close** request to send. If the **Closeable** flag is clear, this flag is ignored. Otherwise if this flag is clear, the client will send a **CloseObj** request on the object; if this flag is set, the client will send a **Close** request on the stream. Most servers set both **Closeable** and **Server** as a result of an **Open** request.

.. and symbolic links

Helios pathnames can include `.` and `..` to refer to the current directory and to the parent directory respectively. Normally these are handled automatically by the Server library and programmers need not worry about them. However, there is a problem at the root directory of a server. The parent of the root directory should normally be a processor, so the relevant request must be sent back from the server to a higher level. The most convenient way to do this is to install a symbolic link to the processor as the parent of the root directory.

```
int main(void)
{ BYTE      mcname[IOCDataMax];
  Object    *this_processor;

  MachineName(mcname);
  this_processor = Locate(Null(Object), mcname);

  { LinkNode *parent;
    parent = (LinkNode *) Malloc(sizeof(LinkNode) + strlen(mcname));
    InitNode(&parent->ObjNode, "..", Type_Link, 0, DefDirMatrix);
    parent->Cap = this_processor->Access;
    strcpy(parent->Link, mcname);
    Root.Parent = (DirNode *) parent;
  }
}
```

As far as the Server library is concerned a symbolic link is a special type of **ObjNode** structure with a **type** field **Type_Link**. In addition to the usual pieces of information in an **ObjNode** it must contain two other entries: the full name of the object the symbolic link points to, and a capability for that object. Creating a symbolic link involves essentially the above piece of code, but inserting the **LinkNode** structure into a directory rather than placing it as the parent of this server. If a request is sent for an

object that is actually a symbolic link to another object, for some requests the message is automatically forwarded to the correct server. Only **Delete**, **Protect**, **Rename**, and **ObjectInfo** affect the symbolic link itself rather than the object the link points to.

Server library routines

The Server library contains a considerable number of routines. This subsection gives a brief description of what they are for and why a server might want to use them. Full information on individual routines can be found in *The Helios Encyclopaedia* or in the on-line help system.

InitNode is used when building a new **ObjNode** or **DirNode** structure, either when the server is initialising or as the result of a **Create** request or an **Open** request with implicit **Create**.

Dispatch starts up a dispatcher routine. This receives incoming messages sent directly to the server, and will **Fork()** off worker threads to handle these messages. The worker will perform some initialisation and then call a routine defined in the **DispatchInfo** structure.

GetContext is called from inside the worker thread to start walking down the directory tree until the context object has been reached, allowing the capability sent with the message to be validated. This routine is rarely called directly by servers.

GetTargetDir returns the parent directory of the target object, if possible. On success it leaves the parent directory locked and the current pathname pointing to that directory. On failure it will fill in the error class and error group of the **FnRc** field of the incoming message. There are various possible reasons for failure, including insufficient access.

GetTargetObj continues walking down the directory tree after a call to **GetTargetDir()**, returning the object which the operation is attempting to access. The routine always updates the pathname. On success it locks the target object and unlocks the parent directory. On failure it leaves the parent directory locked and fills in the error class and error group of the **FnRc** field of the message.

GetTarget calls **GetTargetDir()**. If that succeeds it calls **GetTargetObj()**. This routine is useful mainly for operations which are guaranteed not to affect the parent directory, such as ObjectInfo.

pathcat takes an existing pathname and appends another string to it, inserting a slash character / if necessary. For example if the first argument is **/Net/00/fs** and the second argument is **include/stdio.h**, the new pathname becomes **/Net/00/fs/include/stdio.h**. The routine assumes that the first string is part of a sufficiently large buffer, preferably **IOCDataMax** bytes.

objname takes a full pathname and extracts the final component, for example **stdio.h**.

addint takes a pathname and an integer, turns the integer into a string, and appends this string to the pathname. This routine is useful when generating unique names in response to a **Create** request. An example was given earlier.

Lookup checks whether a particular name is already present in a directory. It is used mainly by the **GetTarget** routines, but could be useful to server programmers.

Insert puts a new **ObjNode**, **DirNode**, or **LinkNode** into an existing directory. The new structure is added to the tail of the linked list for that directory, the number of directory entries is updated, the **parent** field of the object is completed, and date stamps are updated.

Unlink performs the inverse function, removing an object from a directory.

DirServer can be used in response to an **Open** request if the target being opened is a directory. The routine will handle **Read**, **Close**, and **GetSize** requests and send back error codes for any other operation, because those operations are normally inappropriate for a directory.

FormOpenReply is used to generate the reply message for an **Open**, **Create**, or **Locate** request. The corresponding System library routines return a new Object structure, and this routine fills in the information required to build that structure.

MarshalInfo is called by **DoObjInfo()** to fill in the information required for an **ObjectInfo** reply.

DoLocate is a default handling routine for **Locate** requests. If the target object exists and the client has some access to it, the routine will return the required information, otherwise it will send back an error code.

DoRename handles incoming **Rename** requests. These are quite complex.

1. The client must have **write** access to the current parent directory.
2. The target parent directory must exist and the client must have **write** access to this also.
3. Both the current target and the new position must be relative to the same context object.
4. The **Rename** request cannot perform an implicit **delete** of an existing object.

If these conditions are met, the **DoRename()** routine will move the target object to the new position, updating all the various fields.

DoLink is a default routine for incoming **Link** requests to create a symbolic link. Usually symbolic links are supported only by file servers, including the RAM disc. Some devices may attach special meanings to symbolic links, for example, the **/loader** server uses symbolic links as a way of caching commands. When the symbolic link is accessed the target code is actually loaded, and will remain loaded until explicitly deleted.

DoProtect is a default routine for incoming **Protect** requests, used to update the access matrix of a directory or an object.

DoRevoke is a default routine for incoming **Revoke** requests. This routine changes the encryption key held in the **ObjNode** or **DirNode** structure, thus invalidating any existing capabilities. It requires **Alter** access to the target object, and returns a new capability.

DoObjInfo can be used for incoming **ObjectInfo** requests. It extracts all the information it needs from the appropriate **ObjNode** or **DirNode** structure. If the size or account of an object can change, it is the server's responsibility for maintaining this information.

DoSetDate is a default routine for incoming **SetDate** requests. It requires **Write** access to the target object.

DoRefine is a default routine for incoming **Refine** requests, and produces a new capability allowing less access than the current one.

InvalidFn is a default routine for sending back error codes if the server does not support a particular type of request, such as **CloseObj**. It returns the error code **"EC_Error + EG_WrongFn + EO_Object"**.

NullFn returns an empty message. It can be used as a default routine for **ServerInfo** requests if the server does not attach any special meaning to such requests.

ErrorMsg takes an existing MCB, usually that of an incoming message, and part of an error code. It or's the current **FnRc** field of the MCB, usually just the subsystem code, with the error code and sends the appropriate error message with no data or control vectors.

UpdMask is called by the **GetTarget** routines while walking down the directory tree. It updates the current access mask in an incoming request using the access matrix held in the **DirNode** or **ObjNode** structure. This routine is rarely called explicitly.

CheckMask checks that the client program has the specified amount of access to some object. Several access bits can be checked in one call.

GetAccess is used to validate a capability, using the encryption key held in an **ObjNode** or **DirNode**.

Crypt encrypts or decrypts a block of data using the specified key. This is used when generating or validating capabilities, but could be used for other reasons. For example, it could allow data to be transmitted in an encrypted format.

NewKey generates a random number which can be used as the encryption key for **ObjNode** and **DirNode** structures. It is called automatically by **InitNode()**.

NewCap takes an existing object and an access mask specifying the amount of access a client should have, and generates a suitable capability. This routine is called automatically by **FormOpenReply()**.

ServMalloc is a memory management routine. When a server runs out of memory the result can be disastrous. To avoid this problem the Server library automatically maintains a small safety net, which it can use to attempt to cope gracefully with lack of memory. This routine is automatically used by servers instead of **Malloc()**.

UnLockTarget unlocks the current target object, maintained in the **ServInfo** structure. It is used mainly in a **do_open()** routine just before waiting for incoming stream requests, to permit other clients access to the same object.

LockTarget locks the target currently defined in the **ServInfo** structure. Again it is used mainly in a **do_open()** routine after a stream message has been received, to ensure that this thread now has sole access to the target object.

AdjustBuffers can be used by some servers to maintain data in a linked list of buffers. Essentially, the routine maintains a list of blocks of the specified size and can be used to add data to the end of the buffer or remove data from the start of the buffer.

GetReadBuffer is used in conjunction with **AdjustBuffers()**.

GetWriteBuffer is also used in conjunction with **AdjustBuffers()**.

DoRead can be used to handle incoming **Read** requests, provided that the server makes use of the Server library's buffering scheme.

DoWrite can be used to handle incoming **Write** requests, provided that the server makes use of the Server library's buffering scheme.

12.5 The /include disc

This section describes a more complex server, the **/include** disc. This server provides a read-only filing system containing the Helios header files. It ensures that all the header files are held permanently in memory somewhere in the network of processors, thus speeding up most compilations. Following the description of the **/include** disc is a discussion of how this server can be extended to become a more general RAM disc server, supporting **write** operations as well as **read**. The sources of the **/include** disc are shipped with every release of Helios, in the directory **/helios/users/guest/examples/servers**, together with the associated utility **buildinc**.

The main purpose of this section is to give example code for the majority of requests, which can be incorporated into user servers. For simplicity, the code has been reduced in places and does not perform some of the tests it should such as running out of memory. The sources shipped with Helios do include such checks.

12.5.1 /include disc preamble

The preamble of the **/include** disc, declaring the various data structures and functions, is straightforward.

```
    /* Usual header files */
#include <helios.h>
#include <syslib.h>
#include <servlib.h>
#include <nonansi.h>
#include <message.h>
#include <gsp.h>
#include <task.h>

    /* Name of include disk binary image */
#define IncludeDisk        "/helios/lib/incdisk"

    /* Function prototypes */
static BYTE *extract_files(DirNode *root, int number, BYTE *buffer);
static void do_open(ServInfo *);
static void do_private(ServInfo *);    /* for debugging */

    /* Data type used by the include disk */
typedef struct FileEntry {
    ObjNode ObjNode;
    BYTE    Data[1];
} FileEntry;

    /* Root directory and DispatchInfo structure */
static DirNode      Root;
static DispatchInfo IncludeInfo = {
    &Root,
    NullPort,
    SS_RomDisk,
    NULL,
    { do_private, 0 },
    {
      { do_open,   2000 },  /* FG_Open       */
      { InvalidFn, 2000 },  /* FG_Create     */
      { DoLocate,  2000 },  /* FG_Locate     */
      { DoObjInfo, 2000 },  /* FG_ObjectInfo */
      { InvalidFn, 2000 },  /* FG_ServerInfo */
      { InvalidFn, 2000 },  /* FG_Delete     */
      { InvalidFn, 2000 },  /* FG_Rename     */
      { InvalidFn, 2000 },  /* FG_Link       */
      { InvalidFn, 2000 },  /* FG_Protect    */
      { InvalidFn, 2000 },  /* FG_SetDate    */
      { InvalidFn, 2000 },  /* FG_Refine     */
      { InvalidFn, 2000 },  /* FG_CloseObj   */
      { InvalidFn, 2000 },  /* FG_Revoke     */
      { InvalidFn, 2000 },  /* Reserved1     */
      { InvalidFn, 2000 }   /* Reserved2     */
    }
};
```

There is the usual set of header file includes, function prototypes, and data structures for the root directory and the dispatcher. Given the shortage of subsystem codes the **/include** disc re-uses the code allocated to the ROM disc, fairly reasonable since

this server provides another read-only filing system with all data held in memory. The server only supports Open, Locate, and ObjectInfo requests. **Create**, **Delete**, **Rename**, **Link**, **Protect**, **Revoke**, and **SetDate** all involve modifying the data, and this server is defined to be a read-only system. **Refine** is of little or no use given that the other protection requests are not supported. **CloseObj** is not required. **ServerInfo** could be supported if desired, such that the **df** command would indicate a disc of the appropriate number of Kbytes, all of which are used. A special protocol is supported for debugging, using the private protocol support.

There are three lines in the above piece of code unique to the **/include** server. The define of **IncludeDisk** specifies the file to read during initialisation, which contains all the information required. The routine **extract_files()** fills in a directory structure. The data structure **FileEntry** defines the representation of a file in this server: a standard **ObjNode** structure followed by an arbitrary amount of data.

12.5.2 Initialising the /include disc

The file **/helios/lib/incdisk** is a binary file containing all the header files in a format that can be converted easily into a server's directory tree. The first word of the file is a count of the number of entries in the root directory. This is followed by **ObjNode** and **DirNode** structures for each entry, and the linked list holding the directory entries can be initialised simply by adding these nodes. The actual file data is stored immediately after the **ObjNode**, and since the size is stored with the **ObjNode** it is possible to move to the next entry. Some simple recursion copes with subdirectories. For example, Figure 12.1 contains the file layout for a root directory containing three entries: file **root1**, subdirectory **root2**, and file **root3**. **root2** contains two other files, **sub1** and **sub2**. The code to read this file and turn it into a directory structure is fairly

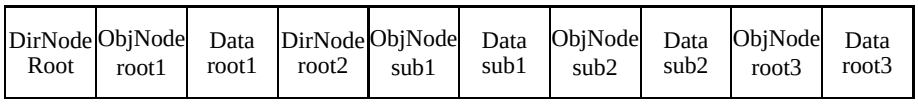


Figure 12.1: **incdisk** file layout

straightforward, if possible errors (such as having insufficient memory to allocate the buffer) are disregarded.

```
int main(void)
{ Object        *inc_disk;
  Stream        *file;
  BYTE          *buffer;
  int           size;
  int           number_entries;

#ifdef DEBUG
     /* If the program is to be started from the shell  */
     /* but still linked with s0.o then it must accept  */
     /* an environment.                                 */
  { Environ        env;
```

```
    (void) GetEnv(MyTask->Port, &env);
  }
#endif

    /* Find and open the IncDisk binary image */
  inc_disk = Locate(Null(Object), IncludeDisk);
  file     = Open(inc_disk, Null(char), O_ReadOnly);
  Close(inc_disk);

    /* Read the file into a suitable buffer   */
  size = GetFileSize(file);
  buffer = (BYTE *) Malloc(size);
  Read(file, buffer, size, -1);
  Close(file);

   /* Initialise the root directory for /include */
  number_entries = *((int *) buffer);
  InitNode((ObjNode *) &Root, "include", Type_Directory, 0,
           0x21110905);
  InitList(&(Root.Entries));

   /* extract all the files and subdirectories */
  extract_files(&Root, number_entries, &(buffer[sizeof(int)]));
  ...
}

static BYTE *extract_files(DirNode *parent, int number, BYTE *buffer)
{ ObjNode   *objnode;
  int       size;

  while (number-- > 0)
   {    /* Insert next entry into the current directory */
     objnode = (ObjNode *) buffer;
     Insert(parent, objnode, FALSE);
     objnode->Dates.Creation = objnode->Dates.Access =
     objnode->Dates.Modified = GetDate();

        /* If a file, move to the next entry */
     if (objnode->Type == Type_File)
      { size = sizeof(FileEntry) + objnode->Size;
        size = (size + 3) & ~3;
        buffer  = &(buffer[size]);
      }
     else
        /* If a directory, extract its contents. */
      { DirNode *subdir    = (DirNode *objnode);
        int number_entries = subdir->Nentries;
        subdir->Nentries   = 0;
        InitList(&(subdir->Entries));
        buffer = &(buffer[sizeof(DirNode)]);
        buffer = extract_files(subdir, number_entries, buffer);
      }
   }
```

```
  return(buffer);
}
```

Most of the work of constructing the **/include** directory hierarchy had been done already when the **include** disc binary image was produced by the **buildinc** command. The above code essentially just reads the whole file into memory and walks down the buffer, getting the pointers right and resetting some time stamps. The information held in the binary image will contain pointers to memory within the **buildinc** address space, which is not the address space of this server.

12.5.3 Dispatching

Once the binary image of the **include** disc data has been read and processed the whole directory tree is in memory. Hence it is possible to add an entry into the name table and call the dispatcher. The code for this is almost identical to the code used in the lock server: only the name of the server has changed.

```
int main(void)
{ BYTE   processor_name[IOCDataMax];
  Object *name_entry;

#ifdef DEBUG
  { Environ env;
    (void) GetEnv(MyTask->Port, &env);
  }
#endif

  MachineName(processor_name);

  IncludeInfo.ReqPort = NewPort();

  { NameInfo name;
    Object   *this_processor;

    this_processor = Locate(Null(Object), processor_name);

    name.Port     = IncludeInfo.ReqPort;
    name.Flags    = Flags_StripName;
    name.Matrix   = DefNameMatrix;
    name.LoadData = Null(WORD);

    name_entry = Create(ThisProcessor, "include", Type_Name,
          sizeof(NameInfo), (BYTE *) &name);
    Close(ThisProcessor);
  }

 Dispatch(&IncludeInfo);
 Delete(name_entry, Null(char));
}
```

12.5.4 The Open handler

The only request handling routine that this server handles itself is Open requests. All other requests can be handled by default routines or are invalid for a read-only filing system. The **do_open()** routine will look something like this:

```
static void handle_read(MCB *m, ObjNode *f);
static void handle_seek(MCB *m, ObjNode *f);

static void do_open(ServInfo *servinfo)
{ MCB     *m        = servinfo->m;
  MsgBuf  *r;
  ObjNode *f;
  IOCMsg2 *req      = (IOCMsg2 *) m->Control;
  Port    reqport;
  BYTE    *data     = m->Data;
  char    *pathname = servinfo->Pathname;

  f = GetTarget(servinfo);
  if (f == Null(ObjNode))
   { ErrorMsg(m, EO_File_Null); return; }

  r = New(MsgBuf);
  if (r == Null(MsgBuf))
   { ErrorMsg(m, EC_Error + EG_NoMemory + EO_Message);
     return;
   }

  FormOpenReply(r, m, f,
                Flags_Server | Flags_Closeable | Flags_Selectable,
                pathname);
  reqport = NewPort();
  r->mcb.MsgHdr.Reply = reqport;
  PutMsg(&r->mcb);
  Free(r);

  if (f->Type == Type_Directory)
   { DirServer(servinfo, m, reqport);
     FreePort(reqport);
     return;
   }

  f->Account++;
  f->Dates.Access = GetDate();

  forever
   { WORD        e;

     m->MsgHdr.Dest = reqport;
     m->Timeout     = StreamTimeout;
     m->Data        = data;

     UnLockTarget(servinfo);
```

```
      e = GetMsg(m);
      m->MsgHdr.FnRc = SS_RomDisk;
      LockTarget(servinfo);

      if (e < Err_Null) break;  /* abort on any error */

      f->Dates.Access = GetDate();

      switch(e & FG_Mask)
       { case FG_Read    :
              handle_read(m, f); break;

         case FG_Close   :
              if (m->MsgHdr.Reply != NullPort)
               ErrorMsg(m, Err_Null);
              goto done;

         case FG_GetSize :
              InitMCB(m, 0, m->MsgHdr.Reply, NullPort, Err_Null);
              MarshalWord(m, f->Size);
              PutMsg(m);
              break;

         case FG_Seek    :
              handle_seek(m, f); break;

         case FG_Select  :
              e &= O_ReadOnly;
              if (e == 0)
               ErrorMsg(m, EC_Error +EG_Protected + EO_File);
              else
               ErrorMsg(m, e);
              break;

         default :
              ErrorMsg(m, EC_Error + EG_FnCode + EO_File);
              break;
       }
    }

done:
  f->Account--;
  FreePort(reqport);
}
```

Again most of this code should be familiar by now. The target directory or file is identified. Since the **/include** server is intended to be a generally available read-only system there is no point in checking that the client can access it. The Server library is used to handle subdirectories. For files this worker thread will go into a loop waiting for a stream request. Any type of error will cause the thread to terminate, forcing the client to re-open the connection if it needs to use the file again. **Read** and **Seek** requests

are handled in separate routines. **Close** involves terminating the loop. **GetSize** simply returns the fixed size of the file. The **Selectable** flag is sent in the reply, so the stream may receive **Select** requests: if the client is doing a **select** for reading, this returns immediately, anything else is an error.

The **account** field is used to keep track of the number of open connections. If there is no better use for the **account** field, this is a sensible use, possibly providing some useful debugging information.

12.5.5 Read requests

Read requests contain three parameters: the amount of data to read, where to start the read, and a timeout. For file and similar servers, the timeout can be ignored. For other servers the timeout indicates the maximum amount of time that the client is willing to wait, and the server must send a reply within that time.

The **/include** disc handles reads easily. The amount of data in the file is known, and the file is held in a single buffer following the **ObjNode** structure, so testing parameters passed to the **read** is easy.

```
static void handle_read(MCB *m, ObjNode *f)
{ ReadWrite *rw   = (ReadWrite *) m->Control;
  WORD      pos   = rw->Pos;
  WORD      size  = rw->Size;
  FileEntry *file = (FileEntry *) f;
  if (pos < 0)
   { ErrorMsg(m, EC_Error + EG_Parameter + 1); return; }
  if (size < 0)
   { ErrorMsg(m, EC_Error + EG_Parameter + 2); return; }
  if (pos >= f->Size)
   { m->MsgHdr.FnRc = ReadRc_EOF;  /* Clear subsystem */
     ErrorMsg(m, 0);
     return;
   }
  if ((pos + size) > f->Size) size = f->Size - pos;
  ...
}
```

The code performs various tests, mostly unnecessary for well-behaved clients. It is illegal to read before the start of the file or to read negative amounts of data. Reads past the end of the file should generate an EOF message. Note that the **FnRc** field of the message currently contains a subsystem code which must be cleared. If a read involves more data than is left in the file, the size of the read is corrected. Sending back the data to the client is more complicated, as it may be split into several messages. Any message could be lost on the way to the client, so the protocol uses sequence numbers to monitor which messages have arrived. This sequence number is held in the **FnRc** field of the message, with a code in the bottom nibble describing how the **Read** is progressing. Since the bottom four bits are used for this code, the sequence number must increase by 16 with every message. **ReadRc_SeqInc** is defined to be 16 in **gsp.h**. The code for returning arbitrary amounts of data would look something like this:

```
static void handle_read(MCB *m, ObjNode *f)
```

```
{ int  i;
  int  sequence = 0;
  Port reply    = m->MsgHdr.Reply;
  ...
     /* Send the data in chunks of up to 65535 bytes */
  for (i = 0; i < size; i+= 65535)
   { m->MsgHdr.Dest     = reply;
     m->MsgHdr.Reply    = NullPort;
     m->MsgHdr.ContSize = 0;
      /* Is this the last message ? */
     if ((i + 65535) < size)
      {    /* Not the last message yet */
        m->MsgHdr.DataSize = 65535;
        m->MsgHdr.Flags    = MsgHdr_Flags_preserve;
        m->MsgHdr.FnRc     = sequence + ReadRc_More;
        sequence          += ReadRc_SeqInc;
      }
     else
      { m->MsgHdr.DataSize = size - i;
        m->MsgHdr.Flags    = 0;
        m->MsgHdr.FnRc     = sequence + ReadRc_EOD;
      }
       /* point to next bit of buffer */
     m->Data    = &(f->Data[i + pos]);
     m->Timeout = 5 * OneSec;
       /* and send the data to the client */
     PutMsg(m);
   }
  ...
}
```

For example, suppose one file in the **/include** disc contains 150K. A client wants to read 200K of data starting from a position 50K into the file. The amount of data requested will be truncated to 100K, because that takes the **Read** request to the end of the file. The first message sent will contain 65535 bytes starting at position 51200 within the buffer. This message will have a sequence number of 0 and a function code of **ReadRc_More**, indicating that another message will follow shortly. The message header flags field is set to **Preserve**, because the same message port will be used for another message. The second message will contain (100K - 65535) bytes starting at the appropriate offset. The **FnRc** field will have a sequence number of 16 indicating that it is the second message, plus **ReadRc_EOD** specifying the end of the data that will be returned for this **read** request. **ReadRc_EOF** could be used instead since the end of the file has been reached, but in practice this offers little or no benefit. The second message will have the message header flags field cleared, indicating the end of this transaction.

12.5.6 Seek requests

Strictly speaking **Seek** requests are not necessary for the GSP protocol. Every **Read** and **Write** request contains the position within the file from which to perform the operation. However, using Seek messages may enable some servers to optimise caching

strategies and the like by giving advance warning of the position of the next **Read** or **Write** request.

The request contains the current position within the file, a seek mode, and an offset. The seek mode may be relative to the start of the file, the end of the file, or the current position. The reply contains a single word in the control vector, the new position within the stream.

```
static void handle_seek(MCB *m, ObjNode *f)
{ SeekRequest *req   = (SeekRequest *) m->Control;
  WORD        curpos = req->CurPos;
  WORD        mode   = req->Mode;
  WORD        newpos = req->NewPos;

  switch(mode)
   { case S_Beginning : break;
     case S_Relative  : newpos += curpos; break;
     case S_End       : newpos += f->Size; break;
   }
  if (newpos > f->Size) newpos = f->Size;
  if (newpos < 0) newpos = 0;
  InitMCB(m, 0, m->MsgHdr.Reply, NullPort, Err_Null);
  MarshalWord(m, newpos);
  PutMsg(m);
}
```

12.5.7 Private protocols for debugging

Many servers are critical to the behaviour of Helios, and if and when things go wrong it is often desirable to provide users with the ability to enable some debugging options within the server. The networking software is a good example: there are commands **diag_ns** and **diag_tfm** to control debugging within the Network server and the Task Force Manager respectively. The **/include** disc is hardly of equal importance to those servers, but can be used to illustrate the techniques used.

The private protocol handler will be invoked for any incoming IOC-format request which has a function code outside the accepted range. These function codes currently vary from 0x0010 for Open requests, up to 0x00f0. Hence sending a message with function code 0x1070, **GetInfo**, or 0x1080, **SetInfo**, is acceptable. The following code fragment could be incorporated into a debugging command such as **diag_inc**.

```
static word inc_private(char *name)
{ Object     *server = Locate(Null(Object), name);
  MCB        m;
  WORD       control[IOCMsgMax];
  BYTE       data[IOCDataMax];
  Port       reply_port;
  word       rc;

  if (server == Null(Object))
   { fprintf(stderr, "diag_inc: failed to locate server %s\n", name);
     exit(EXIT_FAILURE);
   }
```

```
  reply_port = NewPort();
  InitMCB(&m, MsgHdr_Flags_preserve, NullPort, reply_port,
          FC_GSP + FG_GetInfo);
  m.Control  = control;
  m.Data     = data;
  MarshalCommon(&m, server, Null(char));

  SendIOC(&m);
  m.MsgHdr.Dest = reply_port;
  rc = GetMsg(&m);
  FreePort(reply_port);
  return(rc);
}
```

The first step in the diagnostics program is to locate the target server, using either a default name or the name provided as an argument to the program. Given this object it is possible to construct a message in the right format and send it off to the server through the program's IOC controller. The server will send back a reply.

Suppose that the **include** disc supports up to 32 debugging options, conveniently encoded bit-wise in a single word. The current set of debugging options can be obtained by a **GetInfo** message, a new set can be enabled with a **SetInfo** message, and a Terminate message causes the server to exit. The above client code can be modified easily to provide these three facilities. The code in the server would look something like this:

```
static word DebugOptions = 0;

static void do_private(ServInfo *servinfo)
{ ObjNode *f;
  MCB     *m   = servinfo->m;
  IOCMsg2 *req = (IOCMsg2 *) m->Control;

    /* The message must refer to the root directory of /include */
  f = GetTarget(servinfo);
  if (f == Null(ObjNode))
   { ErrorMsg(m, EO_Directory); return; }
  if (f != (ObjNode *) &Root)
   { ErrorMsg(m, EC_Error + EG_WrongFn + EO_Object); return; }

    /* Cope with the three different debugging requests */
  switch(servinfo->FnRc & FG_Mask)
   { case FG_GetInfo   :
       InitMCB(m, 0, m->MsgHdr.Reply, NullPort, Err_Null);
       MarshalWord(m, DebugOptions);
       PutMsg(m);
       break;

     case FG_SetInfo   :
       DebugOptions = req->Arg.Mode;
       ErrorMsg(m, Err_Null);
       break;
```

```
      case FG_Terminate :
        ErrorMsg(m, Err_Null);
        AbortPort(IncludeInfo.ReqPort,
                  EC_Fatal + EG_Exception + EE_Abort);
        break;

      default :
        ErrorMsg(m, EC_Error + EG_WrongFn + EO_Object);
        break;
    }
}
```

The diagnostics program and the server must obviously agree on which bits in the debugging mask correspond to which debugging options. Typically this is done by putting the known options into a shared header file, such as **include.h**.

```
#define dbg_Open   0x0001
#define dbg_Close  0x0002
#define dbg_Read   0x0004
#define dbg_Seek   0x0008
```

At the appropriate places within the server code it is now possible to compare the current value of the **DebugOptions** variable with one of these constants, and conditionally produce some debugging output. Commonly this is done through a macro rather than by explicit code. Using macros makes it easier to produce debugging and non-debugging versions of servers if desired, the latter tending to be rather smaller.

```
#ifdef Debugging
#define Debug(a,b) if (DebugOptions & a) report(b)
#else
#define Debug(a,b)
#endif

static void do_open(Servinfo *servinfo)
{ ...
  Debug(dbg_Open, ("in the do_open routine, target %s"\
                   f->Name));
  ...
}
```

Servers linked with the C library can obviously call the C library formatted output routines such as **fprintf()**. Servers designed not to be linked with the C library typically contain their own. The example code shipped with Helios contains such routines.

12.5.8 A RAM disc

The **/include** disc provides a simple read-only filing system. It is convenient to examine the amount of work involved in making it a more general server, like a RAM disc, supporting **write** operations as well.

The first thing to note is that a more complex server will need to support more of the GSP requests. The **include** disc only supports **Open**, **Locate**, and **ObjectInfo** requests, plus **Read**, **Seek**, **Close**, **GetSize**, and **Select** on streams. A writeable server may also have to support **Create**, **ServerInfo**, **Delete**, **Rename**, **Link**, **Protect**, **SetDate**, **Refine**, **Revoke**, **Write**, **SetSize**, and possibly **GetInfo** and **SetInfo** for control operations such as setting baud rates. In addition the **Open** request can be rather more complicated than before. These requests are outlined below. Formal specifications of the individual requests can be found in chapter 13, *General Server Protocol* or in *The Helios Encyclopaedia*, for example the entry **FG_Open** contains the specification for **Open** requests.

Locate

The purpose of the **Locate** request is to test whether or not the target object exists, and, if it does, to return enough information to the client side to let the System library construct an **Object** structure. The Server library contains a **DoLocate()** routine which essentially does the following:

```
static void do_locate(ServInfo *servinfo)
{ MCB     *m        = servinfo->m;
  MsgBuf  *r;
  ObjNode *f;
  IOCMsg1 *req      = (IOCMsg1 *) m->Control;
  char    *pathname = servinfo->Pathname;

  f = GetTarget(servinfo);
  if (f == Null(ObjNode))
   { ErrorMsg(m, EO_File); return; }

  unless(CheckMask(req->Common.Access.Access, AccMask_R))
   { ErrorMsg(m, EC_Error + EG_Protected + EO_File); return; }

  r = New(MsgBuf);
  if (r == Null(MsgBuf))
   { ErrorMsg(m, EC_Error + EG_NoMemory + EO_Message); return; }

  FormOpenReply(r, m, f, 0, pathname);
  PutMsg(&r->mcb);
  Free(r);
  f->Dates.Access = GetDate();
}
```

The information returned is similar to that for **Create** and **Open** requests, which will return **Object** or **Stream** structures in the client.

ObjInfo

The **ObjectInfo** request is used to get additional information about some file or directory, such as its size and the date stamps. The Server library contains a **DoObjInfo()** routine to handle such requests. The following code fragment performs much the same operation.

```
static void do_objinfo(ServInfo *servinfo)
{ MCB     *m   = servinfo->m;
  IOCMsg1 *req = (IOCMsg1 *) m->Control;
  ObjNode *f;

  f = GetTarget(servinfo);
  if (f == Null(ObjNode))
   { ErrorMsg(m, EO_File); return; }

  unless(CheckMask(req->Common.Access.Access, AccMask_R))
   { ErrorMsg(m, EC_Error + EG_Protected + EO_File); return; }

  MarshalInfo(m, f);
  PutMsg(m);
  f->Dates.Access.Access = GetDate();
}
```

The **MarshalInfo()** routine extracts the object type, name, flags, access matrix, account, size, and date stamps from the current **ObjNode** or **DirNode** structure, and packs them into the reply message. When writing servers for specific devices there is usually a need to update the **Size** field, either in the **ObjNode** structure before the call to **MarshalInfo()** or in the actual reply message afterwards, to reflect the amount of data actually available.

Create

Not all servers need support **Create** requests, for example it is difficult to create another **/rs232** port if the underlying hardware is missing. Different servers also vary in the way that **Create** requests are handled if the target object already exists. A file server would normally return an error code. The **/window** server accepts the requests and creates a new window with a unique name, not necessarily matching the name requested. Since this new name is returned in the reply message and held in the resulting **Object** structure the client will not become confused about which window it is supposed to use. The Server library provides an **addint()** routine which is useful here.

A newly created object must be fully initialised. Note in particular that the **Entries** or **Contents** field of the **DirNode** or **ObjNode** is not initialised by **InitNode()**, and hence this must be done by the server code itself. The Server library contains various routines for storing arbitrary amounts of data in a linked list of buffers, which is particularly useful for devices such as RAM discs. The use of these routines will be described later.

The **Create** request can come with some additional, server-defined, block of data. For example, to load a program into memory the **Load()** routine of the System library sends a **Create** request to the appropriate **/loader** server, with the name of the program to be loaded in this block of data. The size of this block of data is limited to **IOCDataMax**, 512 bytes, minus the space required to hold the generic IOC message data. Servers which do not expect to receive such extra data can just ignore it. Other servers have to check that the data has actually been sent.

To create an object the client must usually have **write** access to the parent directory. The server must ensure that the client is given full access to the newly created

object, by resetting the access mask in the incoming message to **AccMask_Full** before constructing the reply message.

ServerInfo

The **ServerInfo** request is used to get additional data about the server. For example, a file server responds to this request with disc usage statistics, using the **FSInfo** structure. The **df** command sends such a request to a file server, and expects to get back the correct data structure. At present every server can define its own reply structure, subject to a maximum data size of **IOCDataMax**, since there has to be a separate program requesting the information from particular types of servers. This situation may change in future versions of Helios. The **ServerInfo** request usually only provides information to the client, so only read access is required.

The following code fragment illustrates how a file server could handle **ServerInfo** requests.

```
static void do_serverinfo(ServInfo *servinfo)
{ MCB     *m   = servinfo->m;
  IOCMsg1 *req = (IOCMsg1 *) m->Control;
  ObjNode *f   = GetTarget(servinfo);
  FSInfo  data;

  if (f == Null(ObjNode))
   { ErrorMsg(m, EO_File); return; }
  unless(CheckMask(req->Common.Access.Access, AccMask_R))
   { ErrorMsg(m, EC_Error + EG_Protected + EO_File); return; }

  data.Flags = Flags_Server;
  data.Size  = /* something */;
  data.Avail = /* something */;
  data.Used  = /* something */;

  InitMCB(m, 0, m->MsgHdr.Reply, NullPort, Err_Null);
  MarshalData(m, sizeof(FSInfo), (BYTE *) &data);
  PutMsg(m);
  f->Dates.Access = GetDate();
}
```

Rename

The **Rename** request is rarely handled explicitly, since the Server library has a **DoRename()** routine to perform the operation. The request message contains two different names, both of which must be relative to the same context object which must be inside the server. Extracting and manipulating both names is complicated, requiring the following steps:

1. Determine the current target directory, and check that the client has **write** access to it.

2. Find the target object and ensure that it is not the root directory of this server.

3. Reset the target object held in the **ServInfo** structure to be the original context object. Change the **Next** field of the message to the **ToName** field of the **Rename** request so that it is possible to look for the destination object.

```
static void do_rename(ServInfo *info)
{    /* Remember access to context object */
  WORD mask = req->Common.Access.Access;
  ...
  req->Common.Access.Access = mask;
  req->Common.Next          = req->Arg.ToName;
  servinfo->Target          = (ObjNode *) servinfo->Context;
  Wait(&servinfo->Target->Lock);
  ...
}
```

4. Search for the destination directory, and check that the client has write access to this directory as well.

5. Check that the target object does not exist, in other words that the **Rename** is not overwriting some existing object.

6. Change the name of the target object to the new name.

7. If the current and target directories are different, **Unlink** the target object from its current directory, and **Insert** it into the new directory.

8. Unlock the previous directory!

Link

Symbolic links are usually supported only by file servers. A **Link** request is like a **Create** request, but the data structure to be entered in the directory tree should be a **LinkNode** structure rather than an ObjNode or DirNode. This structure should contain the name and capability sent with the message, which will refer to the destination of this link. Symbolic links do not usually require special treatment by servers, since the Server library's **GetTarget()** routines will automatically forward messages to the appropriate server when a link is encountered. The exception is **Delete** requests which should remove the symbolic link and not the target object: the delete handler must recognise objects of **Type_Link**. Like **Rename** requests, **Link** messages are not usually handled explicitly in server code since the Server library has a suitable **DoLink()** routine.

Protect

The **Protect** request is used to install a new access matrix in the **ObjNode**, **DirNode**, or in the **LinkNode** structure corresponding to the target object. The client should have Alter access to the target object. It is usually desirable to check that it is still possible to either delete the object or alter its access matrix again, or it will be impossible to get rid of this object. The following code fragment achieves this:

```
static void do_protect(ServInfo *servinfo)
{ MCB     *m    = servinfo->m;
  IOCMsg2 *req  = (IOCMsg2 *) m->Control;
  ObjNode *f;
  Matrix  new   = req->Arg.Matrix;

  f = GetTarget(servinfo);
  if (f == Null(ObjNode))
   { ErrorMsg(m, EO_Object); return; }

  unless( CheckMask(req->Common.Access.Access, AccMask_A))
   { ErrorMsg(m, EC_Error + EG_Protected + EO_Object);
     return;
   }

  if ((UpdMask(AccMask_Full, new) & (AccMask_A + AccMask_D)) == 0)
   { ErrorMsg(m, EC_Error + EG_Invalid + EO_Matrix); return; }

  f->Matrix = new;
  f->Dates.Access = f->Dates.Modified = GetDate();
  ErrorMsg(m, 0);
}
```

Again the Server library has a **DoProtect()** routine which does the same work as the above code, and most servers can use this routine instead.

SetDate

The **SetDate** request is almost identical to the **Protect** request, and is usually handled by the **DoSetDate()** routine of the Server library. The client should have **write** access to the target object. Any non-zero dates in the request should be used to overwrite the date stamps in the **ObjNode** or **DirNode** data structure, and usually only the last-modified date is changed. The reply message consists of just the message header, with no additional data.

Refine

The **Refine** request is more complicated. The purpose of the request is to produce another capability for the object, usually permitting less access, which can then be passed on to other users. The client may or may not have alter access to the target object and the resulting behaviour is different: essentially the owner, with alter access, should be able to produce any capability. If this was not the case, the owner would first have to change the protection matrix to provide the required access, then use **refine**, and finally revert the protection matrix. While this is happening some other client might be able to use the temporarily-relaxed access conditions to gain greater access to the object. The following code does much the same work as the **DoRefine()** routine of the Server library.

```
static void do_refine(ServInfo *servinfo)
{ MCB        *m      = servinfo->m;
```

```
  IOCMsg2    *req    = (IOCMsg2 *)(m->Control);
  ObjNode    *f;
  Capability cap;
  AccMask    newmask = req->Arg.AccMask;

  f = GetTarget(servinfo);
  if (f == Null(ObjNode))
   { ErrorMsg(m, EO_File); return; }

     /* prepare reply message */
  InitMCB(m,0,m->MsgHdr.Reply,NullPort,Err_Null);

    /* If the client has Alter permission to the object */
    /* use the new mask as given, otherwise restrict it */
    /* by the actual access.                            */
  unless( CheckMask(req->Common.Access.Access,AccMask_A) )
   newmask &= req->Common.Access.Access;

  NewCap(&cap, f, newmask );
  MarshalCap(m,&cap);
  PutMsg(m);
  f->Dates.Access = GetDate();
}
```

Revoke

The purpose of the **Revoke** request is to change the encryption key stored with the **ObjNode** or **DirNode**, invalidating outstanding capabilities. The handler must return a new capability for the object, or it may become inaccessible. The following code fragment performs essentially the same operation as the **DoRevoke()** routine of the Server library.

```
static void do_revoke(ServInfo *servinfo)
{ MCB       *m   = servinfo->m;
  IOCCommon *req = (IOCCommon *)m->Control;
  ObjNode   *f;
  Capability cap;

  f = GetTarget(servinfo);
  if (f == Null(ObjNode))
   { ErrorMsg(m, EO_Object); return; }
     /* only allow revocation if the target is the     */
     /* context object AND the client has alter rights */
  unless (f == (ObjNode *)servinfo->Context &&
              CheckMask(req->Access.Access,AccMask_A) )
   { ErrorMsg(m,EC_Error+EG_Protected); return; }
     /* change the key, and construct a new capability */
   f->Key = NewKey();
   NewCap(&cap, f, req->Access.Access);
   InitMCB(m,0,m->MsgHdr.Reply,NullPort,Err_Null);
   MarshalCap(m,&cap);
   PutMsg(m);
```

```
    f->Dates.Access = f->Dates.Modified = GetDate();
}
```

Private

Private protocol messages acting on named objects were described earlier in this chapter, as a way of providing server debugging facilities. In addition they can be used to handle requests specific to a server, should this be necessary. It is recommended that such use of private protocols be avoided, as the whole point of having a General Server Protocol is to make all servers behave in the same way wherever possible and thus provide a consistent interface.

Open

Open request handlers are generally the most complicated of requests acting on names. A full handler routine may have to allow for the following possibilities:

1. The target object may not exist yet. If the **O_Create** bit is set, the object must be created, provided that the client has **write** access to the parent directory.
2. If both the **O_Create** and the **O_Exclusive** bits are set the object must not exist yet.
3. If the **O_Truncate** bit is set, the file should be truncated, in other words set to 0 length, if that makes sense for the server. This requires **write** access to the object.
4. If the **O_NonBlock** bit is set, the server should always reply immediately to **Read** and **Write** requests, and not wait until some timeout expires before replying.
5. The access has to be checked for each additional stream request sent to the handler routine. For example, a client might open a file in read-only mode and then attempt to write to it.
6. If the target object is a directory, the **DirServer()** routine of the Server library should be invoked to handle stream requests.

Read

The **/include** disc read routine described earlier is useful if all the data to be read is conveniently held in a single buffer in memory, or even in several buffers. For a server such as a RAM disc where more data can be written at any position within the file at any time, handling **Read** requests becomes more complicated. To cope with such requirements the Server library has built-in routines to manipulate arbitrary amounts of data, using linked lists of buffers. The linked list header is held in the **Contents** field of the appropriate **ObjNode**.

The main routine for manipulating these buffers is **AdjustBuffers()**. This routine takes four arguments: a linked list pointer, usually the address of the appropriate **Contents** field; a starting position; a final position; and a size for the amount of data held

in each buffer within the linked list, typically 1024 or 4096. The routine ensures that there is enough buffer space within the linked list to hold data between the start and end position. Buffers will be added to or removed from the linked list as required.

For example, suppose that the truncate bit is set in the open mode. This means that the entire contents of the file should be deleted, in other words the buffer list should be emptied. The following code fragment achieves this:

```
#define BufSize 1024

static void do_open(ServInfo *servinfo)
{ ...
  if (mode & O_Truncate)
   AdjustBuffers(&(f->Contents), 0, 0, BufSize);
  ...
}
```

For files in a RAM disc the starting position is always fixed at 0. For devices like fifos the starting position moves as data is read, and buffers are released by calls to **AdjustBuffers()** at the end of a **Read** request. Device servers such as **/rs232** typically behave in the same way.

Assuming that all the data is held in such buffers, the Server library provides a routine **DoRead()** which can be used to handle read requests. The routine assumes that the request has been validated, in other words that the client is not attempting to read past the end of file. The following code fragment illustrates its use.

```
static void do_read(MCB *m, ObjNode *f)
{ ReadWrite *rw = (ReadWrite *) m->Control;

  if ((rw->Pos < 0) || (rw->Size < 0) || (rw->Pos > f->Size))
   { ErrorMsg(m, EC_Error + EG_Invalid + EO_Message); return; }

    /* Cope with reads at the end of file */
  if (rw->Pos == f->Size)
   { m->MsgHdr.FnRc = ReadRc_EOF; ErrorMsg(m, 0); return; }

  if ((rw->Pos + rw->Size) > f->Size)
   rw->Size = f->Size - rw->Pos;

  DoRead(m, GetReadBuffer, &f->Contents);
  f->ObjNode.Dates.Access = GetDate();
}
```

When interacting with devices rather than file servers it is important to allow for the timeout sent in the **Read** request. The protocol specifies that the server should return data in response to a **Read** request as soon as that data is available. For example, if the **Read** request is for 100 bytes and only one byte of data is available, the server should send that one byte, rather than wait for the remaining 99 to be generated somehow. If the non-blocking flag is set in the open mode, the server should always reply immediately, even if there is no data. Otherwise the server should wait for up to the specified timeout for data, and return either that data or a timeout. This may involve a certain amount of polling. The server is permitted to generate a timeout message before the timeout has actually expired, if that is convenient.

Write

The **Write** request involves the most complicated message transactions in the GSP protocol. Some or all of the following steps are involved.

1. The control vector of the first message contains a position, the amount of data to be written, and a timeout.

2. The data vector may contain the data to be written if the amount involved is less than or equal to IOCDataMax, since that is the size of the message buffer normally used by the **do_open()** routine. If the server has set the **NoIData** flag, this will not happen.

3. If the server cannot handle the amount of data to be written, it should send back an error message within the specified timeout. For example, a serial line might be in use for another write. This error message should be of class **EC_Recover** since retrying the same message may cause a success.

4. If the server can handle the **Write** request and has not yet received the data, it must send back a message indicating how it wants to receive the data. This initial reply contains sizes for the amount of data to be sent in the first message and for subsequent messages, to allow the server to make optimal use of its buffers or caches.

5. The client will now send the data in one or more messages, using the format requested by the previous message. These messages will be sent as quickly as possibly, so the server must be ready to receive them all.

6. If the server can no longer perform the **Write** operation it should send back a recoverable error code at this point.

7. If the server has received all the data and can perform the **Write** operation, it should reply with a success message.

8. The server should now perform the actual write. Note that as far as the client is concerned the **Write** finishes when the data has been received by the server, not when the underlying hardware operation finishes.

Coping with all these variations can be rather difficult. The following code fragment shows how some of these can be handled in user code, ignoring the possibility that the device may not be ready yet.

```
static void do_write(MCB *m, ObjNode *f)
{ ReadWrite  *req       = (ReadWrite *) m->Control;
  bool       ownbuf     = FALSE;
  Port       reply      = m->MsgHdr.Reply;
  Port       myport     = m->MsgHdr.Dest;
  BYTE       *buffer, *ptr;
  word       amount, fetched;

  amount = req->Size;
```

```
     /* Cope with immediate and non-immediate data */
  if (m->MsgHdr.DataSize ne 0)        /* immediate */
   buffer = m->Data;
  else
   { if ((buffer = Malloc(req->Size + 1)) eq Null(BYTE))
      { ErrorMsg(m, EC_Error + EG_NoMemory + EO_Message); return; }
     ownbuf = TRUE;
       /* Send the initial reply, requesting the data */
#define ChunkSize 16384
     InitMCB(m, MsgHdr_Flags_preserve, reply, NullPort,
             WriteRc_Sizes);
     MarshalWord(m, (amount > ChunkSize) ? ChunkSize : amount);
     MarshalWord(m, ChunkSize);
     PutMsg(m);
       /* Receive all the data */
     ptr = buffer; fetched = 0;
     while (fetched < amount)
      { m->MsgHdr.Dest  = myport;
        m->Data         = ptr;
        if (GetMsg(m) < 0)
         goto done;
        fetched += m->MsgHdr.DataSize;
        ptr = &(ptr[m->MsgHdr.DataSize]);
      }
   }
      /* Acknowledge the Write */
  InitMCB(m, 0, reply, NullPort, WriteRc_Done);
  MarshalWord(m, amount);
  PutMsg(m);
  /* Do the actual write. There are "amount" */
  /* bytes in buffer "buffer".               */
done:
  if (ownbuf) Free(buffer);
}
```

Alternatively, the Server library provides a **DoWrite()** routine analogous to the **DoRead()** routine. Obviously this is useful mainly if the data is simply to be held in memory, and does not require any further processing. If the data must be sent on to a piece of hardware, either the **Write** request has to handled completely by user code, or the server has to walk down the linked list of buffers to get to the data to be written. The following code fragment illustrates how the **DoWrite()** routine can be used in a RAM disc or similar server.

```
static void do_write(MCB *m, ObjNode *f)
{ ReadWrite *rw = (ReadWrite *) m->Control;

  if ((rw->Pos < 0) || (rw->Size < 0) || (rw->Pos > f->Size))
   { ErrorMsg(m, EC_Error + EG_Invalid + EO_Message); }

   /* Add more buffer space if necessary and possible */
  if ((rw->Pos + rw->Size) > f->Size)
   { if (!AdjustBuffers(&f->Contents, 0, rw->Pos + rw->Size,
```

```
                        BufSize))
      { f->Size = ((Buffer *) f->Contents.Tail)->Pos + BufSize;
        ErrorMsg(m, EC_Error + EG_NoMemory + EO_File);
        return;
      }
     else
      f->Size = rw->Pos + rw->Size;
   }

  DoWrite(m, GetWriteBuffer, &f->Contents);

  f->ObjNode.Dates.Modified = f->ObjNode.Dates.Access = GetDate();
}
```

GetSize

The **GetSize** request is used to determine the amount of data currently available. For a file server or a RAM disc this means the amount of data in the file. For something like a serial line server it means the amount of data buffered up and ready to be read by the client. Note that the server need not guarantee that a subsequent read will actually return this amount of data, for example another client might be reading from the same device and obtain the data first.

SetSize

The **SetSize** request is rarely used. File servers may need to support it for truncating files to zero length, or conceivably to some non-zero length. Changing the size of a file cannot cause that file to grow. The server should return the new size of the file in its reply message.

GetInfo

The **GetInfo** request is used to obtain some control information about a particular device. Its main purpose is for window and rs232 devices, to determine baud rates and the like. Servers which do not use such control information can ignore this request. The amount and nature of the data returned can vary from server to server, but should never exceed IOCDataMax bytes. Window and rs232 servers use the Attributes structure.

SetInfo

SetInfo is used to change control information such as baud rates for serial lines. The same arguments apply as for **GetInfo**.

12.6 Device drivers

Helios is designed to be independent of any particular hardware. This applies also to servers for particular types of hardware. For example, the Helios file server can be used with a variety of SCSI disc interfaces, with the M212 disc controller, and with a rawdisc device provided by an I/O Server. Most of the complexity of the file server,

such as disc caching and block allocation, are common to all the hardware. The code to drive one specific type of disc is held in a separate code module, known as the device driver, which is loaded at run-time by the main file server.

Using device drivers when writing servers has several advantages.

1. There is no need to ship different binary versions of the server for the different types of hardware. A single binary version suffices for all systems.

2. All the hardware-specific details are isolated in a separate file, with a clearly defined interface between the main server and the hardware-specific details. These hardware-specific details typically involve between 1 and 20 percent of the overall code size.

3. The server can be ported to different hardware simply by producing a new device driver. There is no need to change the server, and in fact the server sources are not usually required to do the port.

4. In theory debugging is easier. Exactly the same server can be tried with different bits of hardware, to determine whether some problem is generic to the server or specific to one device driver or one type of hardware.

This chapter describes the use of device drivers for a simple **/keyboard** server. The programming techniques and restrictions for writing device drivers have been discussed already, in chapter 3, *Programming under Helios*. For existing Helios servers with device driver interfaces, such as the file server, the ethernet server, the X window system, and the Network server, specifications of the device drivers together with example code are usually shipped with the appropriate Helios package.

12.6.1 The /keyboard server

The initialisation code for the **/keyboard** server is slightly different from servers described so far, for the following reasons:

1. The root object for this server is a stream object, not a directory. It is not possible for clients to access **/keyboard/0** or other entries within a subdirectory of this server.

2. The server needs to determine the name of the device driver somehow. In addition it may need to supply the device driver with some options. This information could come from various sources:

   (a) If the server is an ordinary C program it can receive options from the command line through **argc** and **argv**, in the usual manner.

   (b) The server could read a configuration file, for example **keyboard.con** in the usual **/helios/etc** directory. The format of this configuration file can be determined by the server, but ideally it should be relatively simple and use the same syntax as some existing Helios configuration file such as **host.con**. Using a configuration file does not eliminate the problem: there may be several **/keyboard** servers in a network, each requiring a different

configuration file, so the server now needs the name of the configuration file as an option.

(c) The server could load a fixed device driver, for example **keyboard.d**, and as part of the installation procedure the real device driver is copied to that file. Again this does not solve the problem for multiple servers.

(d) Some servers can use the system's **DevInfo** file to get information. This offers little benefit over having a server-specific configuration file, apart from reducing the number of files with different syntaxes.

This example server uses the first option, obtaining the name of the device driver from an argument. If no argument is supplied, the server defaults to using **keyboard.d**.

3. The device driver has to be loaded, and the interface between the main server and the device driver has to be initialised.

The start-up routine for the **/keyboard** server should look something like this:

```
int main(int argc, char **argv)
{ char *driver_name;

  if (argc > 2)
   usage();
  elif (argc == 2)
   driver_name = argv[1];
  else
   driver_name = "keyboard.d";

   /* Do the hardware initialisation */
  init_hardware(driver_name);

   /* Initialise the root of this server to be a Stream */
   /* instead of a directory. Read-only access suffices.*/
  InitNode(&Keyboard_Root, "keyboard", Type_Stream,
           Flags_Interactive, 0x01010101);

   /* Set the Root object's parent to be a LinkNode to */
   /* the processor, create a name table entry, and    */
   /* call the Dispatcher as usual.                    */

   /* If and when the dispatcher terminates, tidy up.  */
   tidy_hardware();
}
```

Most of this involves small modifications to the servers described so far. First, consider exactly how the device driver interface should work. A **/keyboard** server provides an event interface. The client side, typically the X window system server, opens a stream to the keyboard and enables an event on this stream. If successful it gets back a message port, and it should wait on this message port. The server should send event messages to this port when keyboard events occur (whenever a key is pressed or released).

The device driver must monitor the actual hardware and, as soon as an event occurs, report this event to the main server. Typically this would be done by a thread **Fork()**ed off inside the device driver which either polls the hardware or waits for an interrupt to happen. The simplest way for the device driver to report an event is to call a routine **new_keyboard()** inside the main server code, so when the device driver is started up it must be given the address of this routine. To cope with **Acknowledge** and **NegAcknowledge** messages the main server code will have to maintain some history information about recent events.

The **do_open()** routine should look like this:

```
static Port      KeyboardPort = NullPort;
static Semaphore KeyboardLock;

static void keyboard_open(ServInfo *servinfo)
{ MCB     *m            =  servinfo->m;
  MsgBuf  *r;
  ObjNode *f;
  IOCMsg2 *req          = (IOCMsg2 *) (m->Control);
  BYTE    *data         = m->Data;
  char    *pathname     = servinfo->Pathname;
  Port    stream_port;
  Port    my_event_port = NullPort;

  f = GetTarget(servinfo);
  if (f eq Null(ObjNode))
   { ErrorMsg(m, EO_File); return; }

  unless (f eq &Keyboard_Root)
   { ErrorMsg(m, EC_Error + EG_WrongFn + EO_Object); return; }

  r = New(MsgBuf);
  if (r eq Null(MsgBuf))
   { ErrorMsg(m, EC_Error + EG_NoMemory + EO_Message); return; }

  FormOpenReply(r, m, f, Flags_Closeable | Flags_Interactive,
                pathname);
  r->mcb.MsgHdr.Reply = stream_port = NewPort();
  PutMsg(&r->mcb);
  Free(r);

  f->Account++;
  UnLockTarget(servinfo);
  forever
   { word errcode;

     m->MsgHdr.Dest = stream_port;
     m->Timeout     = StreamTimeout;
     m->Data        = data;

     errcode        = GetMsg(m);
     m->MsgHdr.FnRc = SS_Keyboard;
```

```
if (errcode < Err_Null)
 {   /* Event streams cannot time out if an event is enabled. */
   if (errcode eq EK_Timeout)
    { Wait(&KeyboardLock);
      if ((KeyboardPort eq my_event_port) &&
          (KeyboardPort ne NullPort))
       { Signal(&KeyboardLock); continue; }
      Signal(&KeyboardLock);
      break;
    }
   errcode &= EC_Mask;
   if ((errcode eq EC_Error) || (errcode eq EC_Fatal))
    break;
   else
    continue;
 }

if ((errcode & FC_Mask) ne FC_GSP)
 { ErrorMsg(m, EC_Error + EG_WrongFn + EO_Stream); continue; }

switch(errcode & FG_Mask)
 { case FG_Close :
     if (m->MsgHdr.Reply ne NullPort)
      { m->MsgHdr.FnRc = 0; ErrorMsg(m, Err_Null); }
     goto done;

   case FG_EnableEvents :
     { WORD mask = m->Control[0] & Event_Keyboard;

       Wait(&KeyboardLock);
       if (mask eq 0)  /* disable */
        { if ((KeyboardPort eq my_event_port) &&
              (KeyboardPort ne NullPort))
           { AbortPort(KeyboardPort, EC_Error);
             KeyboardPort = my_event_port = NullPort;
           }
          InitMCB(m, 0, m->MsgHdr.Reply, NullPort, Err_Null);
          MarshalWord(m, 0);
          PutMsg(m);
        }
       else
        { if (KeyboardPort ne NullPort)
           AbortPort(KeyboardPort, EC_Error);
          KeyboardPort = my_event_port = m->MsgHdr.Reply;
          InitMCB(m, MsgHdr_Flags_preserve, m->MsgHdr.Reply,
                  NullPort, Err_Null);
          MarshalWord(m, mask);
          PutMsg(m);
        }
       Signal(&KeyboardLock);
       break;
     }
```

```
        default :
          ErrorMsg(m, EC_Error + EG_WrongFn + EO_Stream);
          continue;
      }
   }

done:
  f->Account--;
  FreePort(stream_port);
  Wait(&KeyboardLock);
  if ((KeyboardPort eq my_event_port) && (KeyboardPort ne NullPort))
   { AbortPort(KeyboardPort, EC_Error);
     KeyboardPort = NullPort;
   }
  Signal(&KeyboardLock);
}
```

Again, this **do_open()** routine varies in a number of places from similar routines in previous servers. Since the **/keyboard** server supports rather different requests, this is not entirely surprising.

1. There a message port **KeyboardPort** to which events will be sent. This port must be accessible to the device driver thread calling the **new_keyboard()** routine. At any one time a keyboard server can send event messages to at most one client, but this client could vary. As a complication, it is conceivable that several clients might try to access the keyboard at the same time, and hence the server must be able to cope with this.

2. Since **KeyboardPort** is a resource shared by at least two threads, the worker thread running **do_open()** and the device driver thread, a semaphore is needed to prevent simultaneous access to the port.

3. There is another message port local to the **do_open()** routine which can be used to check whether or not this client currently holds the event.

4. The error handling is slightly different. A timeout usually means that this worker thread can go away because the client has requested no activity for half an hour or so, and may have died. For event based servers it is intended that the client performs as little interaction with the server as possible and merely accepts messages, thus halving the message traffic. Hence a timeout should be ignored if this thread currently holds the event port.

5. Similarly, non-serious errors are best ignored by this server.

6. Following a **close** request, if this thread has control of the keyboard event port, it is desirable to send back an error to the client. Usually the client side will have a separate thread receiving the messages. Certainly it is necessary to clear the main keyboard event port, so that no further messages will be sent to a client that has closed the stream.

7. The **EnableEvents** message can be used for two purposes. If the requested event is 0 or unrecognised, the message serves to disable the keyboard event. Otherwise it enables the keyboard event.

It is now possible to consider the routines responsible for interacting with the device driver.

```
#include <ioevents.h>
#define         KeytabSize        32
static  word    KeyboardCounter = 1;
static  word    KeyboardTail    = 0;
static  word    KeyboardHead    = 0;
static  IOEvent Keytab[KeytabSize];
static  DCB     *KeyboardDCB;

static void init_hardware(char *device_driver)
{ int i;

  InitSemaphore(&KeyboardLock, 1);
  for (i = 0; i < KeytabSize; i++)
   { Keytab[i].Type  = Event_Keyboard;
     Keytab[i].Stamp = 0;
   }

  KeyboardDCB = OpenDevice(device_driver, NULL);
  if (KeyboardDCB == Null(DCB))
   { fprintf(stderr, "/keyboard: failed to loader driver %s\n",
             device_driver);
     exit(EXIT_FAILURE);
   }

  Operate(KeyboardDCB, &new_keyboard);
}

static void tidy_hardware()
{ CloseDevice(KeyboardDCB);
}

static void new_keyboard(bool up, int scancode)
{ MCB m;

  Wait(&KeyboardLock);
  if (KeyboardPort eq NullPort) goto done;

  Keytab[KeyboardHead].Counter = KeyboardCounter++;
  Keytab[KeyboardHead].Key     = scancode;
  Keytab[KeyboardHead].What    = up ? Keys_KeyUp : Keys_KeyDown;

  InitMCB(&m, MsgHdr_Flags_preserve, KeyboardPort, NullPort, 0);
  m.Data            = (BYTE *) &(Keytab[KeyboardHead]);
  m.MsgHdr.DataSize = Keyboard_EventSize;
  m.Timeout         = 5 * OneSec;
  (void) PutMsg(&m);
```

```
  KeyboardHead = (KeyboardHead + 1) & (KeytabSize - 1);

done:
  Signal(&KeyboardLock);
}
```

A table of recent events is kept so that **Acknowledge** and **NegAcknowledge** requests could be handled, if desired. The **init_hardware()** routine initialises this table, loads the specified device driver, and initialises the device driver with the address of the **new_keyboard()** routine. No extra information is available to be passed to the device driver when it is loaded, so the second argument to **OpenDevice()** is NULL. With some servers this second argument might point to information obtained from a configuration file or the **DevInfo** file. The second argument to **Operate()** is just a function pointer, since that is the only information required by this device driver. With other servers this second argument is usually a **DevReq** structure as defined in the **device.h** header file, or some superset of that structure.

The device driver name passed as argument to **OpenDevice()** can be either an absolute pathname, for example **/helios/local/lib/keyboard.d**, or a simple name referring to a file in the **/helios/lib** directory.

When the server exits it must call **CloseDevice()**, to give the device driver a chance to change the hardware back to a sensible state. If this is not done and the device driver has made use of facilities such as interrupt handling, the processor may crash.

The **new_keyboard()** routine is called from a separate thread within the device driver, when some event data is available. This routine needs exclusive access to some of the server's data, so the semaphores are used. If currently no client has enabled the keyboard event then the data is ignored. A possible improvement might be to invoke the device driver every time the event is enabled or disabled, and leave the device driver to throw away keyboard data that is not going to be used, but the performance improvement would be small. If there is currently a client for the keyboard data, this data is put into the table, in case it has to be sent again following a **NegAcknowledge**, and a message is sent to the client.

Essentially that is all the code required to implement a hardware independent keyboard server with a suitable device driver interface. Now, consider some possible device drivers.

12.6.2 Example device drivers

The initial parts of most device driver code tend to be much the same, no matter what the hardware looks like. The following code shows what is involved:

```
typedef struct KeyboardDCB {
        DCB       DCB;
        VoidFnPtr new_keyboard;
        bool      running;
        /* Any hardware-specific information ... */
} KeyboardDCB;

KeyboardDCB *DevOpen(Device *dev, void *info)
```

```
{ KeyboardDCB *dcb  = Malloc(sizeof(KeyboardDCB));
  dcb->DCB.Device   = dev;
  dcb->DCB.Operate  = &DeviceOperate;
  dcb->DCB.Close    = &DeviceClose;
  dcb->new_keyboard = NULL;
  dcb->running      = FALSE;
  return(dcb);
}

static word DeviceClose(KeyboardDCB *dcb)
{ shutdown_hardware(dcb);
  return(Err_Null);
}

static word DeviceOperate(KeyboardDCB *dcb, VoidFnPtr fn)
{ if (dcb->new_keyboard != NULL)
   { /* Error, device already initialised */ }

  dcb->new_keyboard = fn;
  dcb->running      = TRUE;
  startup_hardware(dcb);
  return(Err_Null);
}
```

Code like the above would be complemented by some hardware-specific functions, plus an assembler file to provide the required calling stubs. Suppose that the keyboard hardware is connected to one of the processor's links, and whenever a key is pressed the hardware will transmit a single byte down the link. If the top bit of this byte is set, a key has been pressed, otherwise it has been released. The remaining seven bits constitute the scancode of the key. Code to support such hardware, including setting the link to the right mode and **Fork()**ing off the process to monitor the hardware would look like this, ignoring error conditions for simplicity:

```
#define KeyboardLink 3

static void startup_hardware(KeyboardDCB *dcb)
{ LinkInfo     info;
  LinkConf     conf;

    /* 1) Reconfigure link 3 to dumb, so that it can be used */
  if (LinkData(KeyboardLink, &info) < Err_Null)
   { /* error, link does not appear to exist */ }
  conf.Flags = info.Flags;
  conf.Id    = info.Id;
  conf.Mode  = Link_Mode_Dumb;
  conf.State = Link_State_Dumb;
  if (Configure(conf) < Err_Null)
   { /* error, link cannot be set to the correct mode */ }

    /* 2) Obtain sole access to the link */
  if (AllocLink(KeyboardLink) < Err_Null)
   { /* error, another program owns this link */ }
```

```
    /* 3) Fork off a thread to monitor the link */
  unless(Fork(1000, &link_monitor, 4, dcb))
   { /* error, out of memory */ }
}

static void link_monitor(KeyboardDCB *dcb)
{ BYTE data[1];

  while(dcb->Running)
   if (LinkIn(1, KeyboardLink, data, 2 * OneSec) >= Err_Null)
    { bool up = (data[0] & 0x0080) == 0;
      (*(dcb->new_keyboard))(up, data[0] & 0x007F);
    }
}

static void shutdown_hardware(KeyboardDCB *dcb)
{ dcb->Running = FALSE;
  Delay(3 * OneSec);   /* to let the monitor exit */
  FreeLink(KeyboardLink);
}
```

During the initialisation the link is set to the right mode for interacting with the device, the device driver obtains sole access to the link, and a separate thread is spawned to get the data from the link. This thread polls the link, passing data to the keyboard server as soon as it arrives. Note that to terminate it is necessary to abort this monitor thread, which can be achieved fairly easily by using timeouts and a suitable flag.

A different type of keyboard hardware might generate an interrupt whenever a key is pressed or released, using the Transputer's event pin. When such an interrupt occurs the key event can be read from a location within the processor's address space. Code for such a piece of hardware would look like this, including a more complicated **KeyboardDCB** structure:

```
typedef struct KeyboardDCB {
        DCB       DCB;
        VoidFnPtr new_keyboard;
        Semaphore wait;
        Event     event;
} KeyboardDCB;

static void startup_hardware(KeyboardDCB *dcb)
{ Event *event = &(dcb->event);

  InitSemaphore(&(dcb->wait), 0);
  event->Pri  = StandardPri;
  event->Code = &event_handler;
  event->Data = dcb;
  if (SetEvent(event) < Err_Null)
   { /* error, failed to install event handler */ }

  unless(Fork(1000, &keyboard_monitor, 4, dcb))
   { /* error, out of memory */ }
```

```
}

static void event_handler(KeyboardDCB *dcb)
{ Signal(&(dcb->wait));
}

static void keyboard_monitor(KeyboardDCB *dcb)
{ BYTE key;
  bool up;

  forever
   { Wait(&(dcb->wait));
     key = *((BYTE *) 0x00006000);
     up = (key & 0x0080) == 0;
     (*(dcb->new_keyboard))(up, key & 0x007F);
   }
}

static void shutdown_hardware(KeyboardDCB *dcb)
{ RemEvent(&(dcb->event));
}
```

Considerable care has to be taken when handling interrupts. The interrupt handling routine will be called from inside the Kernel, so it must do as little work as possible. In this case it simply signals a semaphore. A short time later the **keyboard_monitor()** process will be rescheduled, and it can pass the keyboard event on to the main server. When the server is shutting down the interrupt handler is deactivated, ensuring that the **keyboard_monitor()** process will never be restarted.

12.6.3 The DevInfo file

Helios has a file **/helios/etc/devinfo** which can be compiled from the corresponding **devinfo.src** file with the **gdi** program. The binary file can contain configuration information for certain servers and their device drivers. The main purpose of this configuration file is to support system Helios servers such as the file server and the internet server, and users cannot change the syntax of the source file to reflect the needs of their servers. Nevertheless some users may wish to use the **DevInfo** file as a way of configuring their systems.

The **DevInfo** file supports five different types of entries: file servers, file system device drivers, serial port servers, event-driven servers, and ethernet devices. More entries may be added in the future. A typical **devinfo.src** file might contain the following:

```
fileserver fs
{
   device        m212  # discdevice to use
   cachesize     100   # approx cache size in K
   syncop        1     # synchronous operations
```

```
   volume {             # define a volume
      name       fs1    # volume name
      partition  0      # maps to partition 0 of disc device
   }
   volume {
      name       fs2
      partition  1      # multi-partition volume
      partition  2
   }
}

discdevice m212
{
   name            m212.dev  # device name in /helios/lib
   controller      3         # through link 3
   addressing      1         # addresses are in bytes
   mode            0x11      # MULTI buffered read & write

   # partitions...
   partition {             # partition 0
      drive        0       # partition is on drive 0
      start        2       # starts at cylinder 2
                           # end at last cylinder of drive
   }
   partition {             # partition 1
      drive        1       # occupies whole drive
   }

   partition {             # partition 2
      drive        2       # occupies whole drive
   }

   # disc drives...
   drive {                 # define a physical disc drive
      id           1       # id within controller
      type         1       # type in controller
      sectorsize   512     # size of sectors in bytes
      sectors      17      # sectors per track
      tracks       4       # tracks per cylinder
      cylinders    612     # cylinders
   }

   drive {                 # drive 1
      id           2
      type         1
      sectorsize   512
      sectors      17
      tracks       6
      cylinders    1034
   }

   drive {                 # drive 2
      id           2
```

```
      type         1
      sectorsize   512
      sectors      17
      tracks       6
      cylinders    1034
   }
}

serialserver RS232
{
   name      rs232         # server name
   device    rs232.d       # device
   address   0x00cc0000    # device base address

   line {                  # line 0
      name    line0        # addressed by /RS232/line0
      offset  0            # offset = line within device
   }

   line {                  # line 1
      name    line1
      offset  1
   }
}

eventserver KEYBOARD
{
   name      keyboard      # server name
   device    keyboard.d    # device driver name
   address   0x00dd0000    # device base address
}
```

This **devinfo.src** contains all the configuration information needed by a fairly complex file system, running on two separate discs, one of which has two different partitions. Details of all the file server configuration options can be found in the file system documentation. In addition there is information about a serial line server and about an event driver server. The serial line server provides a directory **/rs232** with two entries, **line0** and **line1**. The server needs to load device driver **/helios/lib/rs232.d** to interact with the hardware, which can be found at location **0x00cc0000** within the processor's address space. The event driven server provides a single server, **/keyboard**, using the device driver **/helios/lib/keyboard.d** to interact with the hardware which can be found at location **0x00dd0000** within the processor's address space.

The above text file has to be compiled with the **gdi** program before it can be used. The server needs to read the resulting binary object into memory, scan it to find the required entry, and extract the data. The binary file consists of a chain of **InfoNode** structures, as defined in the header file **device.h**. These structures contain indices to the entry name, for example **fs** in the fileserver entry above, plus indices to the entry specific data structures: **FileSysInfo**, **DiskDevInfo**, **SerialInfo**, **EventInfo**, and so on.

Suppose that the **/keyboard** server needs to use the **DevInfo** file to get information about the device driver and the base address of the hardware. The required entry in the

file has the name **KEYBOARD**. The following code fragment can be used to extract this information.

```
int main(void)
{ char devname[IOCDataMax];
  int  hardware_address;
  ...
  unless(read_devinfo("KEYBOARD", devname, &hardware_address))
   { /* Error, missing devinfo entry */ }

  /* Open the device "devname", and give it the hardware address */
  ...
}

static bool read_devinfo(char *entry, char *devname, int *addr)
{ Stream   *s       = Null(Stream);
  Object   *o       = Null(Object);
  BYTE     *devinfo = Null(BYTE);
  ImageHdr hdr;
  bool     result   = FALSE;
  InfoNode *info;

    /* Look in various places for the DevInfo file */
    /* 1) a ROM disk embedded in the nucleus       */
  o = Locate(NULL, "/rom/DevInfo");
    /* 2) a file embedded in the nucleus           */

  if (o == Null(Object))
   o = Locate(NULL, "/loader/DevInfo");

    /* 3) a file in the /helios server             */
  if (o == Null(Object))
   o = Locate(NULL, "/helios/etc/DevInfo");

  if (o == Null(Object)) return(FALSE);

  s = Open(o, Null(char), o_ReaadOnly);
  Close(o); o = Null(Object);
  if (s == Null(Stream)) return(FALSE);

   /* Check the file header information */
  if (Read(s, (BYTE *) &hdr, sizeof(hdr), -1) != sizeof(hdr))
   goto done;
  if (hdr.Magic != Image_Magic) goto done;

   /* Allocate space to hold the whole file */
  devinfo = Malloc(hdr.Size);
  if (devinfo == NULL) goto done;

  if (Read(s, devinfo, hdr.Size, -1) != hdr.Size)
   goto done;
  Close(s); s = Null(Stream);
```

```
   /* Search through the devinfo information to get to */
   /* right device entry.                              */
  info = (InfoNode *) ((Module *) devinfo + 1);
  forever
   { if ((strcmp(entry, RTOA(info->Name) == 0) &&
          (info->Type == Info_Event))
        /* FOUND IT */
      { EventInfo *event_info = (EventInfo *) RTOA(info->Info);
        strcpy(devname, RTOA(event_info->DeviceName));
        *addr  = event_info->Address;
        result = TRUE;
        goto done;
      }
     if (info->Next == 0) break;
     info = (InfoNode *) RTOA(info->Next);
   }

done:
  if (o != Null(Object)) Close(o);
  if (s != Null(Stream)) Close(s);
  if (devinfo != NULL)   Free(devinfo);
  return(result);
}
```

12.7 Standalone servers

For the vast majority of Helios servers the Server library makes programming much easier. However, for some problems it is inappropriate. In particular if the directory structure is not held in the processor's memory, the Server library becomes unusable. Essentially this applies to all non-volatile file servers, whether based on hard discs, floppy discs, non-volatile RAMS, and so on. Since the directory structure is not held in memory it is not possible for the Server library to walk down it to get the context object, the target directory, and the target object in the usual manner.

Writing servers without the Server library is not significantly more complicated than writing ordinary servers. In particular it is fairly easy to use routines similar to the ones in the Server library and thus implement a standalone server. For simplicity, it is probably desirable to use exactly the same data structures as the Server library. This section outlines solutions for some of the problems likely to be encountered when writing standalone servers.

12.7.1 The dispatcher

The Server library dispatcher works as follows:

1. As long as the server's message port remains valid, it receives incoming messages into a suitable buffer.

2. The dispatcher determines the type of the message, for example an **Open** request or a private protocol message, and it will spawn a worker thread using the stack

size specified in the **DispatchInfo** structure for that request.

3. The dispatcher goes around the loop again, waiting for the next request.
4. The worker initialises a **ServInfo** structure, used to maintain information about this request.
5. The worker searches down the directory tree for this server to reach the context object, and validates the capability sent with this message. Typically this context object would be the current directory.
6. If the message appears valid, the appropriate handler routine specified in the **DispatchInfo** structure will be called.
7. Once the handler routine has finished, the worker will do some cleaning up, and the worker thread will terminate.

For standalone servers it is not possible to search the directory tree for the context object, because the context may not currently be held in memory. Hence such servers need their own versions of the dispatcher routine and the corresponding worker. The **DispatchInfo** structure must not contain references to default handler routines provided by the Server library, since the default handlers assume that the directory tree is in memory. It is possible to use the error handling routines **InvalidFn()** and **NullFn()** if desired. The following code fragment illustrates what is required in the dispatcher routine.

```
static void my_Dispatch(DispatchInfo *info)
{ MsgBuf        *m = NULL;
  word          fn;
  word          stacksize;
  DispatchEntry *e;

  forever
   { m = Malloc(sizeof(MsgBuf));
     if( m == Null(MsgBuf) ) { Delay(OneSec); continue; }

     m->mcb.MsgHdr.Dest = info->ReqPort;
     m->mcb.Timeout     = OneSec*30;
     m->mcb.Control     = m->control;
     m->mcb.Data        = m->data;
lab1:
     while ((fn = GetMsg(&m->mcb)) == EK_Timeout);

     if( fn < 0 ) break;
     if ((fn & FC_Mask) != FC_GSP)
      { m->mcb.MsgHdr.FnRc = info->SubSys;
        ErrorMsg(&m->mcb, EC_Error + EG_FnCode);
        goto lab1;
      }

     fn &= FG_Mask;
```

```
     if((fn < FG_Open) || (fn > FG_LastIOCFn))
      e = &info->PrivateProtocol;
     else
      e = &info->Fntab[(fn-FG_Open) >> FG_Shift];

     stacksize = (e->StackSize < 1024) ? 1024 : e->StackSize;
     unless( Fork(stacksize, my_Worker, 12, m, info, e) )
      { m->mcb.MsgHdr.FnRc = info->SubSys;
        ErrorMsg(&m->mcb, EC_Error + EG_NoMemory);
        goto lab1;
      }
   }

 if( m != NULL ) Free(m);
}
```

Essentially this is just an infinite loop, terminated only if an error other than a time-out occurs. Such an error can usually be generated only if the server's message port is aborted or freed. A message buffer is allocated and initialised. When a message arrives the **FG** part of the request is examined. If it is a request that the server is expecting (**Open**, **Create** and so on), the appropriate entry in the **DispatchInfo** structure is extracted. Otherwise the private function entry is used. This gives the stack size for the worker thread, with the system imposing a minimum lower limit, and the worker thread is started. Note that the worker thread may use the stack for the **ServInfo** structure, rather than allocating another chunk of memory dynamically, and the size of this structure is over 500 bytes. Hence it is essential that the stack sizes specified in the **DispatchInfo** structure are reasonable. The corresponding worker routine looks something like this:

```
static void my_Worker(MsgBuf *m,DispatchInfo *info, DispatchEntry *e)
{ DirNode  *d;
  ServInfo servinfo;

  if( setjmp(servinfo.Escape) != 0 ) goto done;
  servinfo.Context      = info->Root;
  servinfo.m            = &m->mcb;
  servinfo.Target       = (ObjNode *)info->Root;
  servinfo.TargetLocked = false;
  servinfo.FnCode       = m->mcb.MsgHdr.FnRc;
  servinfo.DispatchInfo = info;
  MachineName(servinfo->Pathname);
  m->mcb.MsgHdr.FnRc    = info->SubSys;

#ifdef ProtectedServer
  d = my_GetContext( &servinfo );
  if (d == Null(DirNode))
   ErrorMsg(&m->mcb,0);
  else
#endif
   (*e->Fn)(&servinfo);

done:
```

```
  UnLockTarget(&servinfo);
  Free( m );
}
```

The first half of this routine simply initialises the **ServInfo** data structure. This includes a jump buffer, so that at any time while handling this request the server code can abort and terminate this thread. The root of the server's directory tree is always in memory, so this can be used as the current target even if the rest of the directory tree is held somewhere on a hard disc.

The equivalent Server library routine would now call a **GetContext()** routine which starts to walk down the directory tree until the context object has been reached. The message will contain a capability which defines the client's access to the context object. Once the capability object has been checked the Server library can continue walking down the directory tree until the target object has been reached, modifying the client's current access using the access matrices encountered along the way. Not all servers can support a capability based protection mechanism. For example, if the server is intended to support MS-DOS compatible floppy discs, the disc format is fully defined and there is no way of storing the required encryption keys and access matrices within the directory structure. Hence the server is inherently unprotected and there is no way of supporting the Helios protection mechanisms. However the Helios file server uses its own disc format and hence it can store the required encryption keys and access matrices. Both cases are dealt with below.

12.7.2 Name handling without protection

The first case deals with unprotected file servers. For such servers there is no point in walking down the directory tree, carefully manipulating access matrices along the way. Instead the server code can simply build up the entire pathname and perform whatever operation is required. For example, the **do_open()** routine could look something like this:

```
static void do_open(ServInfo *servinfo)
{ char *object_name;
  ...
  object_name = BuildName(servinfo);

  /* object_name now points to "include/stdio.h"       */
  /* This is all the information needed to manipulate  */
  /* the specified object. N.B. the object may not     */
  /* actually exist...                                 */
  ...
}
```

The routine **BuildName()** extracts all the required information from the request message to build up the name of the object within this server. This name is appended to the current pathname held in the **ServInfo** structure, so that it is still possible to send back the full pathname in the reply messages to **Open**, **Create** and **Locate** requests. The **BuildName()** routine itself is rather complicated, having to cope with several different cases.

1. The incoming message contains three offsets within the Control vector, referring to strings in the data vector.

   (a) **Context** refers to the start of the context object, usually the current directory. This may be set to -1 if there is no context object, in other words when the target object has been specified with an absolute pathname.

   (b) **Name** refers to a pathname relative to the context object. This may be set to -1 if the target object is the context object.

   (c) **Next** contains an index to the remainder of the pathname. This may be inside the Context name, if the context object is contained within the server. Alternatively it may be inside the Name string if the context object was reached before the message arrived at the server.

2. When the message arrives at the server it is guaranteed that **Next** will point to just past the server name.

3. The **Pathname** field of the **ServInfo** structure will contain the current processor name, because this was filled in in the worker routine earlier. The current server name can be appended to this.

4. The remainder of the **Next** string can be added.

5. If **Next** currently points inside the Context name, it may be necessary to append the **Name** string as well.

6. The incoming message may contain references to `.` and `..` which have to be filtered out. This could cause the message to refer to objects outside this server, for example **/msdos/../fs**, and in theory it is desirable to forward the message to that server. In practice this can be difficult, so in this example such requests will generate errors.

7. If the file server supports symbolic links to objects outside the server, everything becomes much more complicated. It is necessary to detect a symbolic link as soon as it has been reached, and forward the remainder of the message on to the right server. Again this complication is not dealt with here.

Given these requirements it is now possible to show the **BuildName()** routine. The Server library's **pathcat()** routine is particularly useful here: it is like the C library **strcat()** string concatenation routine, but inserts the directory separator / where appropriate.

```
static char *BuildName(ServInfo *servinfo)
{ char      *pathname = servinfo->Pathname;
  MCB       *m        = servinfo->m;
  IOCCommon *req      = (IOCCommon *) m->Control;
  BYTE      *data     = m->Data;
  char      *result;
  char      *local;
  ObjNode   *f        = servinfo->Target;

   /* pathname currently hold the processor name.    */
```

```
   /* "f" is guaranteed to point at the server root. */
   /* This routine should return a pointer to the    */
   /* object relative to the server name.            */
  pathcat(pathname, f->Name);
  result = pathname + strlen(pathname) + 1;

   /* copy the remainder of the current string */
  if (data[req->Next] != '\0')
   pathcat(pathname, &(data[req->Next]));

   /* if Next is part of the Context string, and if there is a */
   /* Name string as well, append the name string.             */
  if ( ((req->Next < req->Name) && (req->Context < req->Name)) ||
       ((req->Next > req->Name) && (req->Context > req->Name)) )
   if (req->Name != -1)
    pathcat(pathname, &(data[req->Name]));

   /* Eliminate occurrences of . and .. in the name            */
  Flatten(servinfo, result);

  return(result);
}
```

The **Flatten()** routine has to walk through the resulting pathname eliminating any occurrences of `.` and `..`. This may take the pathname to some object outside the server, which is treated as an error. A suitable error message is generated, and the jump buffer in the **ServInfo** structure is used to abort the handling of this request.

```
static void Flatten(ServInfo *servinfo, char *start)
{ char *source = start;
  char *dest   = start;
  MCB  *m      = servinfo->m;

  until(*source == '\0')
   { if (*source == '.')
      {
         /* case 1 : xyz/./yyy */
       if ((source[1] == '/') || (source[1] == '\0'))
        { source += 2; continue; }

         /* case 2 : xyz/../yyy */
       if ((source[1] == '.') &&
           ((source[2] == '/') || (source[2] == '\0')))
        { if (dest <= start)
           { ErrorMsg(m,  EC_Error + EG_Name + EO_Server);
             longjmp(servinfo->Escape, 1);
           }

           /* backtrack one directory level */
         dest--;
         until (*(--dest)) == '/');
         dest++;
         continue;
        }
```

```
      }

      /* default, just copy the string */
     until ((*source == '/') || (*source == '\0'))
      *dest++ = *source++;
     if (*source == '/')
      *dest++ = *source++;
   }

 *dest = '\0';
}
```

It is not possible to use the **FormOpenReply()** routine to generate the reply message to **Open**, **Create** and **Locate** requests because this routine assumes there is an **ObjNode** structure. However it is relatively easy to produce a version of **FormOpenReply()** suitable for the server.

```
void FormOpenReply(MsgBuf *r, MCB *m, ObjNode *o,
                   word flags, char *pathname)
{ IOCCommon *req = (IOCCommon *)(m->Control);
  Capability cap;

  if( m->MsgHdr.Reply & Port_Flags_Remote )
   flags |= Flags_Remote;

  r->mcb.MsgHdr.Flags    = 0;
  r->mcb.MsgHdr.DataSize = 0;
  r->mcb.MsgHdr.ContSize = 0;
  r->mcb.MsgHdr.Dest     = m->MsgHdr.Reply;
  r->mcb.MsgHdr.Reply    = NullPort;
  r->mcb.MsgHdr.FnRc     = Err_Null;
  r->mcb.Timeout         = IOCTimeout;
  r->mcb.Control         = r->control;
  r->mcb.Data            = r->data;

  NewCap(&cap, o, req->Access.Access);
  MarshalWord(&r->mcb,   o->Type);
  MarshalWord(&r->mcb,   o->Flags|flags);
  MarshalCap(&r->mcb,    &cap);
  MarshalString(&r->mcb, pathname);
}
```

The reply message contains the object type, flags, capability, and full pathname. The pathname has been generated already in the **BuildName()** routine. The type and flags are no different from servers written with the Server library. The capability will never be checked, since checking it is always the responsibility of the server and this server does not test capabilities. The access mask must be constructed, for example **AccMask_R AccMask_W** for files, or **AccMask_Full** to give complete access. The access mask may be checked by some applications to check that, for example, a **delete** operation is likely to succeed before attempting it, so some effort should be made to ensure that the access mask is sensible.

12.7.3 Name handling with protection

For file servers which do support the Helios protection mechanism, writing standalone servers is more complicated. Consider a server which works roughly as follows:

1. The information held on the hard disc consists of **ObjNode** and DirNode structures, or supersets thereof.

2. These structures do not contain pointers to other **ObjNode** and DirNode structures, since the others are not held in memory. Instead the list nodes and the parent pointers refer to entries within disc blocks.

3. There are some cache management routines, possibly interacting with a separate cache management thread. There is at least one routine which takes a disc block reference, loads the relevant block into memory, and returns a real pointer. There is another routine which releases the disc block again, allowing the block to be removed from necessary.

4. The disc block management is fairly sensible, and keeps directory entries in one block or in contiguous blocks wherever possible.

Walking down a directory tree is now possible, as illustrated by the following pseudo-code:

```
until the target has been reached
   follow the current name string until the directory
    separator character / is found

   if the required name is . then loop

   if .., the target becomes the parent. Get the parent block,
    and release the current block

   find the appropriate directory entry
   get the disk block for that directory entry
   release the current block

   if the entry is missing, generate an error message and
    escape with a longjmp
```

All this is fairly standard for file servers and should not present any major problems. To handle Helios protection as well, the following points have to be considered:

1. If the current string is in the Name rather than the Context than the capability has been verified already.

2. If the current string is in the Context, it is necessary to validate the capability. This involves:

   (a) Walk down the directory until the Context object has been found.

   (b) Use the encryption key of the Context object and the capability passed with the message as arguments to **GetAccess()**.

(c) If **GetAccess()** fails the capability is invalid and an error message must be generated.

(d) **GetAccess()** will update the **Access.Access** mask of the incoming request to be the access permitted to the Context object by this client. If this is zero, the client has no access and an error message should be generated.

3. Continue walking down the directory tree following the Name string, until the target object has been reached. The client must have Read access to go down a directory. The **pathname** field of the **ServInfo** structure must be updated at every step.

4. At every level, update the access mask with the access matrix associated with the current object, using **UpdMask()**.

5. When the target object has been reached, the field **Access.Access** in the request message will contain the access allowed to this object. This can now be checked by the handler routine.

The details of all this will vary greatly from server to server, depending on the underlying hardware and the disc organisation. The above information is intended only to provide the necessary hints.

12.7.4 Directory reads

A final significant difference between writing servers with the Server library and writing standalone servers is in the handling of directories. The Server library has a routine **DirServer()** which can take care of open directory streams. A standalone server has to do this work manually.

A directory appears to Helios like a read-only file. It supports **Read**, **Close**, and **GetSize** requests. Every entry in the directory consists of the following structure:

```
typedef struct DirEntry {
        word     Type;
        word     Flags;
        Matrix   Matrix;
        char     Name[32];
} DirEntry;
```

The values of these various fields are simply copies of what could normally be found in the **ObjNode** or **DirNode** structures. The simplest way to handle directory operations is as follows:

1. When a directory is opened, a chunk of memory is allocated large enough to hold all the directory entries, plus entries for `.` and `...`

2. This chunk is filled in with the current directory contents, starting with `.` and `..` followed by all the directory entries.

3. **Read** and **GetSize** requests simply operate on this chunk of memory, in exactly the same way as a file in the **/include** disc is read. It is not necessary to update the information every time entries are added to or removed from the directory.

4. A **Close** request involves freeing the chunk of memory.

Directory operations simply give a snapshot of the world at the time that the directory was opened, and the information may not be accurate by the time that it is actually read.

Chapter 13

General Server Protocol

The General Server Protocol (GSP) is the glue which binds the Helios system together. It is the means by which client programs obtain services from servers and is designed to be orthogonal, fault tolerant, idempotent and extendable. This section provides an overview of the protocols as currently implemented.

GSP is based on the client-server principle: a client sends a message containing a function code plus some arguments to a server, the server performs the requested operation (or refuses to) and returns a message containing a return code plus some results. Some operations may involve the passing of extra messages between client and server, and the server which eventually replies may not be the one to which the message was originally sent.

13.1 Function and return codes

The encoding of function and return codes is a prime feature in the extendability and orthogonality of GSP. All codes are 32-bit values and follow the same basic rules. A code with zero in the most significant bit is a function code, and a code with the figure one in the same place is an error code. In replies any positive value indicates a successful result, and may contain some extra information (for example, most servers return their id code in all successful replies). With one (non-space) character per bit position, a function code is divided into the following fields:

```
0 CC SSSSS RRRR GGGGGGGGGGGGGGGG FFFF
```

C Protocol class, currently 0 = GSP protocol, 3 = private, 1,2 = reserved.

S Subsystem or server identification code. This may be zero, but if not, a server is free to reject requests which do not contain its own code.

R Retry counter. Used to indicate how many times this request has been retried.

G Generic function code.

F Subfunction or modifier, often server or function specific.

Error codes are divided into the following fields:

```
1 CC SSSSS GGGGGGGG EEEEEEEEEEEEEEEE
```

C Error class:

- **Recover** A transient error such as a timeout or congestion; a simple retry may succeed.
- **Warn** A slightly more serious error, but more fundamental recovery action may allow a retry to succeed.
- **Error** An error from which no automatic recovery is possible.
- **Fatal** A serious error which should probably cause the program to terminate.

S Subsystem or server identification code. This indicates where the error originated, it may not necessarily match the same field in a request function.

G General error code. Indicates what has gone wrong.

E Specific error code. If the most significant bit of this field is set then it is interpreted as an object name code. This indicates the object to which the error occurred.

The rationale behind the encoding of errors is that with only a few values for each field, a very large number of error codes can be generated. The **C** and **S** fields select an error class and subsystem name, the **G** field selects a verb phrase and the **E** field selects a noun phrase. These can then be filled into a prototype sentence of the form:

```
From <S>:  <C>, <G> <S>
```

This is exactly what the Fault library does, selecting the appropriate words or phrases from the faults database. Not all general errors expect an object code to be given. Those that **do not** are listed here, with the interpretation placed on the **E** field.

Parameter Indicates a bad or out of range parameter. The **E** field indicates which parameter of the request message is bad.

Exception Indicates a number of exception conditions. The **E** field contains one of the following codes:

- **Kill** Interpreted by the Processor Manager as an instruction to terminate the sending task. The control vector contains an error code.
- **Abort** Sent by the Kernel when aborting or deleting ports to any threads waiting for message transfers.
- **Signal** A range of values whose least significant 8 bits indicate a signal number. By sending this a server can provoke a signal handler to be called in a client.

All other codes are unused and reserved.

Errno Where a server is based on or derived from Unix code, instead of translating all its error codes into Helios codes, they are passed directly to the client in the **E** field of this general error code.

CallBack If a server knows that it will not be able to service a request for some time it can return this error, causing the client to wait for the number of seconds in the **E** field.

13.2 GSP fundamentals

GSP is designed for use over unreliable communications systems. This is best achieved by making the server stateless and the messages idempotent, allowing a request to be repeated until either it works or fails conclusively.

When the term stateless is applied to servers, it has a somewhat special meaning. It does not imply that the server maintains no state at all: a stateless file server would be quite useless. Instead it means that the server does not maintain any state associated with each client. Hence each request is treated in isolation and any state associated with it is destroyed when the reply is generated. Any state which must persist between requests is kept by the client and sent to the server with each request. This has the effect of insulating both the server and the client from a failure of the other. If the client fails the server simply receives no more requests, it does not need to detect the failure and tidy up any state. If the server crashes the client can simply wait for it to restart, or find another one, and continue from where it left off.

Unfortunately, a purely stateless protocol is cumbersome and requires all the relevant information (object identifier, protection information, operation parameters, etc.) to be presented in every request [1]. To alleviate this, GSP has been defined as a semi-stateless protocol. Essentially, this allows the server to perform the object location and protection operations once, for each client. Once this has been done the client can operate on the object using a much more lightweight protocol.

There are two major request message groups, **direct** and **indirect**. Direct messages are sent by the client directly to the server controlling an object. The object is identified implicitly by the message port to which the requests are sent. Indirect messages are send initially to the task's **IOCPort** and are eventually delivered by the Processor Manager on the server's processor. The target object is identified by information in the message.

Indirect messages operate on the object as a whole, with operations such as **delete**, **rename**, etc. These tend to be generic, and all servers must support the standard set. Direct messages operate on the internal representation of the object, with operations such as **read**, **write**, etc. These are often more specific to the object's representation, but the set of file access operations is common. The indirect operation **Open** returns a port descriptor for direct operations and thus forms the bridge between the two sets.

All operations in the two sets are idempotent, and may be repeated. However, the server is at liberty to destroy a direct request port and all resources associated with it at any time. Since all state is kept with the client, it can obtain a new direct request port at any time by repeating the **Open** request.

[1] For example, the smallest NFS read request is 138 bytes!

The exact set of operations which may be applied to a particular object depend on a large number of factors. The object type is the most important, and certain functions make no sense for particular kinds of object (for example, seek on a serial line). Certain objects may allow access to only one user at a time, so the presence of other clients may restrict the operation set. A distinction must also be drawn between the set of operations which **could** be applied, and those which are allowed by the protection mechanism. For example, the **Write** operation is allowed on files, but if the file has been made read-only then it will be rejected. See chapter 14, *Protection*, for further information.

13.3 Message formats

GSP makes use of the control and data vectors in a particular way. The control vector always contains a fixed-size data structure whose format depends on the message type. The data vector contains variable sized data such as strings. Fields in the control vector structure are either explicit values, or the offset of data in the data vector. If there is nothing in the data vector for a particular offset field, the offset is set to -1.

In the protocol descriptions which follow, offset fields are given one of the following types:

Offset A simple offset into the data vector.

String The offset of a null terminated string.

Struct The offset of a sized data item. This consists of a 4-byte size field followed by that many bytes of data.

All data vector offsets should be aligned to 4-byte boundaries. Hence there may be up to 3 bytes of padding between data vector items.

13.4 Object types

All objects are typed. The type of an object defines the set of GSP requests which may be applied to it and the semantics of the operations performed.

An object **type** is a 32-bit value divided into a flags field in the least significant 4 bits and a type code in the remaining 28 bits. The flags place the object into one of three generic types:

Directory This is a container for other objects. All indirect operations may be applied to it, but only **Read**, **GetSize** and **Close** direct operations must be supported.

Stream This is a source, sink or container for raw data. All indirect operations with the exception of **Create** may be applied. The exact set of direct operations supported is dependent on the open mode, underlying hardware and server implementation, but in general this is an object which may be treated rather like a file.

Private This supports a private interface. All indirect operations may be applied, but some may be rejected as inappropriate, or accepted silently. The direct operation interface is largely server defined, but may contain some standard operations.

13.5 Object flags

In addition to a type, all objects also have an associated set of flags bits. These define various features or options which are not encoded in the type field. The full set of flags is divided into three classes: those which are the responsibility of the server supporting an object, those which are the responsibility of the System library, and those which are the responsibility of the application. These are all stored in a single 32-bit word in the System library data structures.

The flags controlled by the System library are defined in chapter 10, *The System libraries*. The flags controlled by the servers are as follows:

More Indicates that more data about this object is available through **ObjectInfo**.

Seekable Indicates that the current read/write pointer of this object may be repositioned.

StripName For a name table entry, indicates that the object name index in the GSP request should point after the server name.

CacheName For a name table entry, indicates that the name is a cache entry for a server on another processor.

LinkName Indicates that this is a name table entry for a hardware link.

NoIData Indicates that for **Write** requests, no data should be sent in the request message.

ResetContext In cached names, indicates that the object name index should be reset before forwarding the request.

CloseOnSend Indicates that the stream should be closed in the current task if it is to be passed on to another task through **SendEnv**.

OpenOnGet Indicates that the stream should be opened when received through **GetEnv**.

Selectable Indicates that the stream supports the **Select** operation.

Interactive Indicates that the stream is interactive (for example, a terminal).

MSdos Indicates that the object is an MS-DOS file or directory. This enables the necessary character translations needed to make all file systems appear similar.

Extended Indicates that the extended Read protocol should be used.

NoReOpen Indicates that this object cannot be re-opened.

Fast Shorter than normal timeouts may be used.

Closeable Indicates that the server expects the stream/object to be closed.

Server Indicates that the server has allocated resources for the client and thus expects a **Close** direct operation to terminate the session. If this flag is clear but **Closeable** is set, the client must close it with a **CloseObj** indirect operation.

13.6 Indirect operations

Indirect requests are sent initially to a task's own **IOCPort**, the reply is returned directly from the server to the client. The request may have to be passed through several servers before reaching its destination due to the presence of explicit routing in the request, or symbolic links in the servers.

To allow these messages to be processed without intimate knowledge of all possible formats, all indirect messages conform to a common format. This consists of a **context** object plus a **target** object name relative to the context object. Additional operation specific parameters are added only after these structures have been defined.

The context object is described by two items: its name and a capability. The name is a string giving the context object's full pathname in the naming hierarchy. This name must be in **canonical** form, where all symbolic links have been removed. In general this name should have been created by the server which supports the object. The capability is a 64-bit protection handle, with which the server can validate the client's access rights over the object. This consists of an 8-bit access mask field plus a 56-bit validation field. See chapter 14, *Protection*, for more detailed information.

The target name is the name of the object upon which the operation is to be performed. Normally the context object would be a directory and the target an object within that directory. If no target name is given then the context object is the target for the operation. If the target name begins with a `'/'` character, it is an absolute reference and does not need a context object. In this case the context name string is not present and the capability, except for its access mask field, is unused.

The first four words of the control vector are formatted as follows:

String	Context	Context object name
String	Name	Target object name
String	Next	Current name index
Capability	Access	Capability for Context object

This is known as an **IOCCommon** structure and must be present in all GSP messages sent to the **IOCPort**. The **Context** and **Name** fields are offsets into the data vector. If both are present then the context string must appear before the target name string. The **Next** field is an offset into either the context or target name strings. It indicates how far through the name resolution the sender has reached, and where the receiver should start. This allows the request to be passed from server to server as the path name is followed without needing to re-format the message. This field should be initialised

by the client to point to the first character of the context name string, or to the first character of the target name string if no context is provided.

Within the request message header the control and data vector sizes must be at least large enough to contain the IOCCommon structure and its strings. No flags need be set, although the preserve flag will be set by the IOC as a matter of course. The destination port must be the task's own **IOCPort** passed to it at startup. The reply port should normally be a local port to which the reply will be sent. It should not be possible for other messages to be received on this port unless they can be differentiated unambiguously from the reply. Hence the port should be one that is specific to the request, or to the context object, if it can be locked. Exceptionally, the reply port may be a surrogate port, in which case the eventual reply will be routed back to the original provider of the surrogate port without further interaction with the sender.

These operations may be retried in the face of errors. If a retry is needed then the client should increment the **R** retry counter field in the function code. Once this reaches a sufficiently high value, it will trigger higher-level recovery action in the Processor Manager. This will attempt to find an alternative route to the server.

The following sections describe each request in detail. Each section contains the formats of the request and reply messages along with a description of the operation and any important points to be noted. The comments about protection are recommendations only. Since servers are responsible for implementing the protection mechanism themselves, they can choose not to do so. The recommendations follow the practice with all current servers.

13.6.1 Open

Request *Message Header:*

Flags	preserve
Dest	IOC port
Reply	local reply port
FnRc	FG_Open

Control Vector:

IOCCommon	Common	Common part of GSP request
word	Mode	Open mode

Reply *Message Header:*

Flags	NONE
Dest	Request.Reply
Reply	direct port
FnRc	Return Code

Control Vector:

word	Type	Object type code
word	Flags	Object Flag bits
Capability	Access	Access rights to object
String	Pathname	Canonical pathname of object
word	Object	(Optional) Object value

This operation is primarily used to obtain a direct operation port for the target object. The **Mode** is the logical OR of the following set of mode bits:

ReadOnly	Open for reading only, writes will be rejected.
WriteOnly	Open for writing only, reads will be rejected.
ReadWrite	Open for both reading and writing.
Execute	Open for execution.
Private	Open for use through a private interface. It is possible for an object to present two completely different interfaces dependent on this bit.
Create	Create the object if it does not exist.
Exclusive	If **Create** is set, and the file exists, fail.
Truncate	If the object exists, and it makes sense to do so, truncate it to zero length.
NonBlock	Do not block on read and write operations, effectively forces the time-out on these operations to zero.

The client should have **Read** access to all directories between the context and target objects, and the server should ensure that the client has sufficient access rights to allow the mode requested.

In the reply, the **Pathname** is the canonical form of the object's name. It may bear no relationship with the original context and target pathnames supplied in the request and should be used if the stream needs to be reopened. The capability encodes the intersection of the rights the client has to the object and the access rights sought through the requests **Mode** field. The **Object** field is optional and is used by some servers to return an object identifier and avoid opening a direct request port.

In addition to the data in the reply message vectors, two fields of the message header constitute reply parameters. The **Reply** port descriptor, if present, is the direct operation port and may be used to apply direct operations to the object. The **FnRc** field will be greater than zero if the operation was successful, and should be kept and bitwise **OR**ed with the function code of all subsequent direct operations.

13.6.2 Create

Request *Message Header:*

Flags	preserve
Dest	IOC port
Reply	local reply port
FnRc	FG_Create

Control Vector:

IOCCommon	Common	Common part of GSP request
word	Type	Type of object to create
word	Size	Size of **Info** data
Offset	Info	Type specific information

Reply *Message Header:*

Flags	NONE
Dest	Request.Reply
Reply	NullPort
FnRc	Return Code

Control Vector:

word	Type	Object type code
word	Flags	Object Flag bits
Capability	Access	Access rights to object
String	Pathname	Canonical pathname of object

This operation is used to create new objects in a server. It is used not only for the normal creation of files and directories, but also to load code into the loader, execute tasks, and to create name table entries for servers. It is server defined whether an attempt to create an object which already exists will succeed. Three outcomes are possible: return information on the object as if the **create** succeeded, fail the operation indicating that the object already exists or create a new object with a different name and return information on that. This last option is used by the Processor Manager, for example to allow several instances of the same program to co-exist on a single processor.

The client should have **Read** access to all directories between the context and target objects, and have **Write** permission for the parent directory of the new object.

The request defines the type of the object to be created and optionally provides some additional information. A target **Name** field must be present in the **IOCCommon** structure, and is used as the name of the new object. The server is at liberty to modify or totally alter this name as it sees fit, so the returned pathname must be used to re-access the object. The **Info** field is only used in the creation of complex objects such as name table entries and tasks (see section 11.1 on the Processor Manager and section 11.2 on the Loader for details). For most servers, the **Size** field may be zero.

The reply is identical to that produced as a result of an **Open** request, except that no direct operation port will be returned, and no **Object** field is allowed. The returned pathname and capability may be used as the context for subsequent indirect operations. If this is the case the **FnRc** field of this reply should be ORed with the function code of all such operations.

13.6.3 Locate

Request *Message Header:*

Flags	preserve
Dest	IOC port
Reply	local reply port
FnRc	FG_Locate

Control Vector:

IOCCommon	Common	Common part of GSP request

Reply *Message Header:*

Flags	NONE
Dest	Request.Reply
Reply	NullPort
FnRc	Return Code

Control Vector:

word	Type	Object type code
word	Flags	Object Flag bits
Capability	Access	Access rights to object
String	Pathname	Canonical pathname of object

This operation is used for two main purposes: to test whether a particular object exists, and to acquire the canonical pathname and capability of an object for use as the context of subsequent indirect operations. The client should have **Read** access to all directories between the context and target objects, and have some access rights to the target object itself. The request is minimal and simply identifies the object to be located. The reply is identical to that supplied as a result of a **Create** request and the same comments apply.

13.6.4 ObjectInfo

Request *Message Header:*

Flags	preserve
Dest	IOC port
Reply	local reply port
FnRc	FG_ObjectInfo

Control Vector:

IOCCommon	Common	Common part of GSP request

Reply *Message Header:*

Flags	NONE
Dest	Request.Reply
Reply	NullPort
FnRc	Return Code

This operation asks for more information about an object. This operation is only supported if the **More** flag is set in the object flags. The client should have **Read** access to all directories between the context and target objects, and have some access rights to the target object itself. The reply is server and type specific. However, most servers which support this operation will return an **ObjInfo** structure in the data vector.

The **ObjInfo** structure is:

DirEntry A copy of this object's directory entry structure, which is read from directory streams (see **Read** below).

Account A server defined account identifier. This is used by some system servers to return other information.

Size The object size.

Dates A structure comprising three dates indicating its creation, last modified and last accessed times.

If the object is a symbolic link then a **Link_Info** structure is returned:

DirEntry Object's directory entry.

Cap Capability for linked object.

Name Canonical pathname of linked object.

The name and capability returned by this request may be used as the context object in subsequent requests.

13.6.5 ServerInfo

Request *Message Header:*

Flags	preserve
Dest	IOC port
Reply	local reply port
FnRc	FG_ServerInfo

Control Vector:

IOCCommon	Common	Common part of GSP request

Reply *Message Header:*

Flags	NONE
Dest	Request.Reply
Reply	NullPort
FnRc	Return Code

This request obtains some information about the server supporting the target object. It should not matter which object in the server is targeted, and access to this information is not normally checked for access rights. Like **ObjectInfo**, it simply identifies the target object or server. The reply is server specific, but most file systems return a **FSInfo** structure in the data vector:

Flags Flag word, currently unused.

Size File system size in bytes.

Avail Free space in bytes.

Used Used space in bytes.

13.6.6 Delete

Request *Message Header:*

Flags	preserve
Dest	IOC port
Reply	local reply port
FnRc	FG_Delete

Control Vector:

IOCCommon	Common	Common part of GSP request

Reply *Message Header:*

Flags	NONE
Dest	Request.Reply
Reply	NullPort
FnRc	Return Code

This operation requests that the target object should be deleted. The server should ensure that the client has adequate access rights to allow this operation. The client should have **Read** access to all directories between the context and target objects, and **Delete** permission on the target object itself. The request simply identifies the object to be deleted. The reply has no result parameters, the **FnRc** field of the message header will indicate whether the operation was carried out successfully.

13.6.7 Rename

Request *Message Header:*

Flags	preserve
Dest	IOC port
Reply	local reply port
FnRc	FG_Rename

Control Vector:

IOCCommon	Common	Common part of GSP request
String	ToName	New name for target object

Reply *Message Header:*

Flags	NONE
Dest	Request.Reply
Reply	NullPort
FnRc	Return Code

This operation requests that the target object be renamed, and possibly repositioned in the directory hierarchy. An object may only be moved or renamed within its supporting server, it may not be transferred to a different server. The success of an attempt to rename an object over the top of an existing object is server-defined.

The client should have **Read** access to all directories between the context and the target object, and between the context object and the new parent directory. The client should also have **Write** permission to both the target's parent directory and the new parent directory. The request identifies the target in the normal way. The **ToName** field is the new name for the object and is evaluated relative to the context object supplied in the **IOCCommon** field. The reply has no result parameters, the **FnRc** field of the message header will indicate whether the operation was carried out successfully.

13.6.8 Link

Request *Message Header:*

Flags	preserve
Dest	IOC port
Reply	local reply port
FnRc	FG_Link

Control Vector:

IOCCommon	Common	Common part of GSP request
String	Name	Canonical pathname of linked object
Capability	Cap	Capability of linked object

Reply *Message Header:*

Flags	NONE
Dest	Request.Reply
Reply	NullPort
FnRc	Return Code

This operation creates a symbolic link to the object whose name and capability are provided. This is only supported by servers which can store symbolic links in this form. Hence, the MSDOS and Unix file systems cannot support this operation, while the Helios File System and the RAM file system can.

A symbolic link is essentially just a context object stored in a server. When a server encounters one in the resolution of a path name, it rebuilds the original request using the contents of the link as the new context object and the remainder of the target name string as the new target name. It then copies any remaining parameters from the original request, and delivers the new request to its own **IOCPort**. The request will now find its way to the server which supports the new context object, and hence the eventual target. Note that since context names are canonicalised, a link cannot be encountered while resolving the context pathname of a request.

The client should have **Read** access to all directories between the context and target objects, and **Write** permission to the new link's parent directory.

As with **Create**, the **Common** field identifies the object to be created, it must not already exist. The **Name** and **Cap** fields should have been obtained as a result of a **Open**, **Create** or **Locate** operation. No check is made to ensure that the described object exits. The reply has no result parameters. The **FnRc** field of the message header will indicate whether the operation was carried out successfully.

13.6.9 Protect

Request *Message Header:*

Flags	preserve
Dest	IOC port
Reply	local reply port
FnRc	FG_Protect

Control Vector:

IOCCommon	Common	Common part of GSP request
Matrix	Matrix	New access matrix

Reply *Message Header:*

Flags	NONE
Dest	Request.Reply
Reply	NullPort
FnRc	Return Code

This operation alters the protection status of the target object. See chapter 14, *Protection*, for more detailed information. The client should have **Read** access to all directories between the context and target objects, and **Alter** access rights to the target object. The reply has no result parameters, the **FnRc** field of the message header will indicate whether the operation was carried out successfully.

13.6.10 SetDate

Request *Message Header:*

Flags	preserve
Dest	IOC port
Reply	local reply port
FnRc	FG_SetDate

Control Vector:

IOCCommon	Common	Common part of GSP request
DateSet	Dates	Set of dates to set

Reply *Message Header:*

Flags	NONE
Dest	Request.Reply
Reply	NullPort
FnRc	Return Code

This operation requests that one or more of the dates associated with the object are changed. These are defined by the **DateSet** structure:

Creation Date object was created

Access Date object was last accessed

Modified Date object was last modified

A date is measured in the number of seconds since 00:00:00 on 1 January 1970. Only the dates which are non-zero in the request will be altered. The client should have **Read** access to all directories between the context and target objects, and **Write** permission to the target object itself. The reply has no result parameters, the **FnRc** field of the message header will indicate whether the operation was carried out successfully.

13.6.11 Refine

Request *Message Header:*

Flags	preserve
Dest	IOC port
Reply	local reply port
FnRc	FG_Refine

Control Vector:

IOCCommon	Common	Common part of GSP request
AccMask	AccMask	New access mask

Reply *Message Header:*

Flags	NONE
Dest	Request.Reply
Reply	NullPort
FnRc	Return Code

Control Vector:

Capability	Cap	Refined capability

This operation requests that a new capability be created for the target object. The main use for this operation is to enable a program to restrict the access rights to an object when it needs to pass some access over to another, less trusted program. For example, the network server, having booted all processors, **Protect**s them all against external access while keeping a full access rights capability for itself. When a processor is allocated to a user, the Network Server uses **Refine** to manufacture a medium-level access capability to pass to the TFM.

The **AccMask** parameter defines the access mask to be used in the new capability. The client should have **Read** access to all directories between the context and target objects. If the client has **Alter** access rights to the target object then the new mask is used as it stands, otherwise the new mask is bitwise ANDed with the client's actual access mask for the object. This prevents the user obtaining greater access rights than they already have, while recognising that a user with **Alter** permission can give themselves any access rights they want with the **Protect** operation in a much less secure way.

The reply contains a new capability which encodes the new access rights. This may now be used with a canonical name for this object as the context in any subsequent operation.

13.6.12 CloseObj

Request *Message Header:*

Flags	preserve
Dest	IOC port
Reply	local reply port
FnRc	FG_CloseObj

Control Vector:

IOCCommon	Common	Common part of GSP request

Reply *Message Header:*

Flags	NONE
Dest	Request.Reply
Reply	NullPort
FnRc	Return Code

This operation requests that a **Close** operation be performed on the target object. Normally **Close** is sent to the direct operation port of an opened object. However, certain servers do not need to retain an open stream to the client since there are no direct operations needed. However, the server still needs to be informed when the stream is closed.

For example, the **pipe** server acts as a rendezvous point for the two ends of a pipe. Once the rendezvous has been made the two communicating tasks communicate directly without involving the server. However, the server must maintain the rendezvous point in case the clients lose contact and need to re-rendezvous. Thus when the pipe is finally closed, the clients each send a **CloseObj** operation to the server to inform it that the rendezvous point may be destroyed. The client should have **Read** access to all directories between the context and target objects.

In all cases **Close** is merely a hint to the server that the object is no longer open. Hence it does not matter unduly that it is lost. For this reason the client need not supply a **Reply** port, and no reply need be returned. If a reply port is supplied, the generation of a reply is server-dependent.

13.6.13 Revoke

Request *Message Header:*

Flags	preserve
Dest	IOC port
Reply	local reply port
FnRc	FG_Revoke

Control Vector:

IOCCommon	Common	Common part of GSP request

Reply *Message Header:*

Flags	NONE
Dest	Request.Reply
Reply	NullPort
FnRc	Return Code

Control Vector:

Capability	Cap	New capability for target

This operation requests that the server revoke all existing capabilities for the target object and issue a new capability to the client. The client should have **Read** access to all directories between the context and target objects, and **Alter** permission to the target object. The request must not contain a **Name** field in the **IOCCommon** structure, so the context object is the target. The reply contains a new capability for the object with the same access rights as the original supplied in the request. This capability will be the **only** capability which can access the object.

13.7 Direct operations

These requests are sent directly to a server through a direct operation port obtained from an **Open** operation. There is no fixed protocol.

The protocols used here are designed to cope with communications failures and to allow the implementation of a fault recovery strategy at the client when the server crashes or communications are lost.

If an error return code is generated from either **PutMsg** or **GetMsg** the initial recovery action is controlled by the error code's class. If the class is **Recover** then the operation should be retried from the beginning. If the class is **Error** or **Fatal** then the operation should be abandoned, and the code returned to the application. If the class is **Warn** then the client can attempt to re-open the direct operation port.

To re-open the direct operation port the client must construct an **Open** request using the name and capability returned in the original open reply, and send it to its **IOCPort**. The function code of this message should have the **ReOpen** bit set in the **F** field. This is then routed to and treated by the server as a normal open, and a new direct operation port is returned. To allow for transient errors, the client may need to retry the open several times.

13.7.1 Read

Request *Message Header:*

Flags	preserve
Dest	direct port
Reply	local reply port
FnRc	FG_Read

Control Vector:

word	Pos	Position in file to start reading
word	Size	Maximum number of bytes to read
word	Timeout	Time to wait for data to be available

Replies *Message Header:*

Flags	preserve (all except last)
Dest	Request.Reply
Reply	NullPort
FnRc	Return Code OR seqno|ReadRc_More (ReadRc_[EOD|EOF] on last)

Directory Read Replies *Message Header:*

Flags	preserve (all except last)
Dest	Request.Reply
Reply	NullPort
FnRc	Return Code OR seqno|ReadRc_More (ReadRc_[EOD|EOF] on last)

Data Vector:

DirEntry	Entries[...]	Array of directory entry structures

This operation requests data to be transferred from the object to the client. The object must have been opened with **Read** permission for this operation to succeed. The **Pos** field indicates where in the object the read operation should start. For sized objects such as files this must lie within the bounds of the object. For objects which are simply data sources, such as a pipe or serial line, this parameter is used as a sequence counter. In this case the server will interpret a request with a certain **Pos** value as implicit proof that all preceding data has been received successfully. A request with a repeat **Pos** field is interpreted as a retry after some communication error. If such a retry is not an exact duplicate of the original then the server's behaviour may be undefined. Thus to retain the ability to retry a serial stream server needs to retain enough data to repeat each client's last read operation. This can be avoided, at the cost of some extra messages, by employing the extended read protocol described later.

The **Size** field indicates the maximum quantity of data to be transferred. The server must not interpret this as an exact requirement. Any amount of data up to this size, including zero, may be returned. However, the server must **never** return more than this size.

The **Timeout** field indicates how long the client is willing to wait for data to become available. It is specified in microseconds, although the server may impose its own grain size of up to a second. If the timeout is zero then a reply must be generated immediately, whether data is available or not, to allow clients to poll the server. There is no provision for an infinite timeout. If clients want to wait indefinitely they should submit a **Read** request with a finite timeout and simply retry it when it returns with a timeout. In this way a low-level idle exchange is maintained between client and server, allowing either to detect the other's failure. If an infinite timeout were allowed here, the client would not be able to detect whether a lack of response from the server was simply a result of no data being available, or because the server, or communications with it, had failed.

The response to a **Read** request is one or more reply messages. If the message header **FnRc** field of any of these is an error code, it will provoke the necessary recovery action depending on its class. Otherwise the **FnRc** contains a return code in its least significant four bits, and a message sequence number in its most significant 28 bits. The sequence number starts at zero and is incremented for each successive message in the transfer.

The possible return codes are:

ReadRc_More Indicates that there are more messages to follow.

ReadRc_EOD End Of Data. Indicates that this is the last message in the transfer.

ReadRc_EOF End Of File. Indicates that this is both the last message in the transfer, and that the end of the file has been reached.

The data being transferred is contained in the data vectors of the messages. As a special feature, the control vector may contain a number of whole words of data. The extra words in the control vector should be copied into the data buffer immediately after the end of the message's data vector. This is primarily to allow a full 64K to be transferred in a single message instead of 64K-1.

The data messages should be delivered to the client in ascending sequence number order. This allows the client to simply advance the MCB data vector pointer along its buffer, receiving the data directly into the memory area required without needing to copy it out of a message buffer. If a message arrives with a wrong sequence number, or a communications failure occurs, the client should free the original reply port and allocate a new one before retrying the operation. This ensures that any messages still in transit, or yet to be sent, are disposed of and do not interfere with the retry.

If the object's **Extended** flag is set, then the server expects the client to make use of an extended read protocol. This is identical to the normal read protocol except that it contains an extra acknowledge interaction at the end. This acknowledgment is seen by the server as an extra **ReadAck** request:

Control Vector:

word	Result	Size of data received or error code.

The result is either the quantity of data received, or if an error occurred during the transfer, the error code. The server will respond with a header-only message acknowledging receipt. In the face of **Recover** class messages, the client need only retry this message and not the entire transfer.

The effect of the **ReadAck** message is to inform the server that the client has received the data and avoids the need to keep it until the next **Read** operation in case it is a retry.

If the object being read is of type **Directory**, or some derivative, then there are some additional constraints upon the form of the request and reply messages. In the request, the **Pos** and **Size** fields must be a multiple of the size of a **DirEntry** structure. The reply will consist of an array of a whole number of **DirEntry** structures in the data vectors. The **DirEntry** structure contains the following fields:

Type Object type.

Flags Object flags.

Matrix Access matrix for object.

Name Name of this entry.

13.7.2 Write

Request *Message Header:*

Flags	preserve
Dest	direct port
Reply	local reply port
FnRc	FG_Write

Control Vector:

word	Pos	Position in file to start writing
word	Size	Number of bytes to write
word	Timeout	Time to wait for write to start

Sizes Reply *Message Header:*

Flags	NONE
Dest	Request.Reply
Reply	NullPort OR data port
FnRc	Return Code

Control Vector:

word	First	Size of first message data vector
word	Rest	Size of remaining message data vectors
word	Max	(Optional) Maximum transfer size

Already and Done Reply *Message Header:*

Flags	NONE
Dest	Request.Reply
Reply	NullPort
FnRc	WriteRc_Already OR WriteRc_Done

Control Vector:

word	Got	Number of data bytes received
word	Wrote	(Optional) Number of data bytes actually written

Data Messages *Message Header:*

Flags	preserve (all except last)
Dest	Request.Dest OR Sizes.Reply
Reply	NullPort
FnRc	Seqno|ReadRc_More (ReadRc_EOD on last)

This operation requests a data transfer from the client to the object. The object must have been opened with **Write** permission for this operation to succeed. This is the most complex protocol interaction between client and server. It is necessary to ensure data delivery on unreliable communications and to support the semantics of non-blocking writes. In summary, it consists of an initial request message followed by a first reply from the server. If the interaction is not terminated there, the client transfers the data to be written to the server and awaits a final reply from the server.

The **Pos** field indicates where in the object the write operation should start. For sized objects, such as files, this should lie within the bounds of the object, or be exactly one more that the object's upper bound. In this case the write operation will extend the object appropriately. For objects which are simple data sinks, such as a pipe or a serial line, this parameter is used as a sequence counter. In this case if a repeat request is received the server can respond immediately with an **Already** reply as described below.

The **Size** field indicates how much data the client wants to send the server. The server need not accept this much data and may modify the actual transfer size with

the **Max** field in the **Sizes** reply. The **Timeout** field indicates how long the client is prepared to wait for the write operation to complete. This is intended to be used to wait for devices to get ready. The server should not terminate a transfer in the middle simply because the timeout has been reached. A timeout of zero indicates that the writer does not expect to be blocked by the server. In this case the server should accept all the data sent, but only write as much as it can without blocking, reporting the discrepancy in the final reply. As with **Read** there is no provision for an infinite timeout, for the same reasons.

If the data to be transferred is less than or equal to **IOCDataMax** it may be sent to the server in the data vector of the request. However, if the object's **NoIData** flag is set, then this should not be done. Once the request message has been sent the client should await a response from the server. If the **FnRc** field contains an error code the appropriate recovery action is taken, as described earlier. Otherwise it will be one of the following codes from the server:

WriteRc_Done Indicates how much data was received and written.

WriteRc_Sizes Defines how the client should ship the data to the server.

WriteRc_Already This is a response to a retry of a write which has actually succeeded the first time.

If the response is **WriteRc_Done** then the client must have sent data in the request message. If the control vector is empty then all the data has been written. If the control vector contains just the **Got** field then the server will only have written that much data. When sent data in the request, the server is not able to use the **WriteRc_ Sizes** response to alter the transfer size, this feature has the effect of doing that. If the response contains both the reply fields then the application should be informed that not all the data was written.

If the response is **WriteRc_Already** then the server is informing the client that it already has the data and it need not be re-sent. This is a response to a retry of a write operation whose final reply got lost. For this to work correctly, the retry should be an exact copy of the original request. This response effectively shortcuts the remainder of the protocol and should be treated like the final reply described later.

If the response is **WriteRc_Sizes** then the server is ready for data to be transferred. This reply indicates how the transfer should be made. The **First** field indicates what size the data vector of the first message should be. The **Rest** field indicates what size the rest of the messages should be. The last message will be some size less than or equal to **Rest**. The reason for forcing the transfer into this pattern is so that the server can optimise the placement of data into its memory. For example, consider a file server which maintains a block cache. If it could not control the message sizes it would have to receive each message into a buffer and then copy the data out into each cache block. By controlling the message sizes, the server can receive the data messages directly into the cache blocks, eliminating a copy operation. The **First** field allows for the fact that the write may start part way through a cache block. Under no circumstances must the client disobey the server in this matter.

Certain servers may be unable to accept write operations above a certain size. The optional **Max** field in the **Sizes** reply allows the server to reduce the write transfer size

to a more reasonable value. Although the client must obey this demand, it is at liberty to issue another write request for the remaining data as soon as the first operation is completed. This feature is of use to servers which have limited buffering space, and/or slow devices. It allows them to accept as much data as possible, and then to block the client on its timeout while it disposes of it. When buffer space becomes available, the server can then quickly re-start the client to fill it.

If the **Sizes** reply contains a message port in its message header **Reply** field, then the client must send the data messages to this port. Otherwise, the client should send the data to the original request port. This allows the server to redirect the data to a special port, or even to another server.

Once given the instruction to proceed by the server the client can transfer the data. This must be done in the message data sizes defined by the server and sent to the data port defined by the server. The **FnRc** field of these messages must follow exactly the format of the data messages used for the **Read** request. Thus, the least significant 4 bits of the field must be set to **ReadRc_More** for all but the last message, which must be set to **ReadRc_EOD** (**ReadRc_EOF** has no meaning and should never be used). The most significant 28 bits should be set to the serial number of the message starting from zero. The message header **preserve** flag must also be set on all messages, except if the server has provided a different data port in the **Sizes** reply. In this case the last message (the one indicating **EOD**) must have this flag cleared. If the client gets an error from the **PutMsg** operations sending this data, it should abandon the entire transfer and either retry it, reopen the connection, or report an error.

Once the client has sent the data it must wait for a final confirmation reply from the server. This will be a **Done** reply containing up to two fields. The first field, **Got** should match exactly the amount of data the client sent in the data transfer messages. If it does not, then an error occurred and the entire transfer should be repeated. The second result is optional and indicates how much of the data received was actually written to the device. If this does not agree with the **Got** field, then the discrepancy should be reported to higher levels. If this field is absent then all the data sent was written.

It should be noted that the **Read** and **Write** protocols are designed to neatly dovetail together with minimum interaction from a server. An example of how this works is given in section 15.3.

13.7.3 GetSize

Request *Message Header:*

Flags	preserve
Dest	direct port
Reply	local reply port
FnRc	FG_GetSize

Reply *Message Header:*

Flags	NONE
Dest	Request.Reply
Reply	NullPort
FnRc	Return Code

Control Vector:

word	Size	Object size

This operation requests the size of the object. The object may have been opened with any permission. The request simply delivers the function code to the server. The reply contains the current size of the object. In the case of files this will be the file's upper bound. In the case of simple data source objects such as pipes and serial lines, this is the quantity of data which is immediately available to a **Read** operation.

13.7.4 SetSize

Request *Message Header:*

Flags	preserve
Dest	direct port
Reply	local reply port
FnRc	FG_SetSize

Control Vector:

word	Size	Size to set

Reply *Message Header:*

Flags	NONE
Dest	Request.Reply
Reply	NullPort
FnRc	Return Code

This operation requests that the size of an object be set. The object must have been opened with **Write** permission. The **Size** field in the request indicates the new size. The interpretation placed on this request is server or object specific. For files the given size must be less than its upper bound and effects a truncate operation. For pipes this request sets the amount of data which may be written into the pipe before the writer is blocked. This may only be set at the read end of the pipe.

13.7.5 Close

Request *Message Header:*

Flags	NONE
Dest	direct port
Reply	NullPort OR local reply port
FnRc	FG_Close | mode

Reply *Message Header:*

Flags	NONE
Dest	Request.Reply
Reply	NullPort
FnRc	Return Code

This operation informs the server that the client has finished with the direct operation port for the object. Since this operation is only a hint to the server that the client is finished, the request need not contain a reply port. If it does then the server should return a success reply to the client. The **F** field of the request function code, if non-zero, indicates which mode of the object to close:

ReadOnly Close the object for reading only.

WriteOnly Close the object for writing only.

If the object is open for both read and write, setting a mode in the close will close it only for the given direction. Operation for the other direction will be allowed to continue. If the mode is zero, it is closed for both directions.

13.7.6 Seek

Request *Message Header:*

Flags	preserve
Dest	direct port
Reply	local reply port
FnRc	FG_Seek

Control Vector:

word	CurPos	Current position
word	Mode	Seek mode
word	NewPos	New position relative to mode

Reply *Message Header:*

Flags	NONE
Dest	Request.Reply
Reply	NullPort
FnRc	Return Code

Control Vector:

word	NewPos	New position

This function requests that the server calculate a new position in the object and return it to the client. Although the server does not keep an object position pointer for the client, it is given the job of calculating updates to it. This is to ensure that the new position is calculated and validated using the most up-to-date values for the object size. It can also act as a hint to the server that the client's point of interest in the object has moved. This would allow a file server, for example, to dispose of any cached blocks at the old position and to start pre-reads at the new.

The request **CurPos** field is the client's current absolute position pointer and the **NewPos** field is the new relative position pointer. The **Mode** field indicates how these are to be combined to generate the new absolute position:

Beginning Position relative to start of file, **NewPos** is the new absolute position.

Relative Position relative to original: **CurPos**+**NewPos** is the new absolute position.

End Position relative to end of file: the new absolute position is **FileSize**+**NewPos**.

The final absolute position is compared against the object bounds and trimmed appropriately before being returned in the reply message.

13.7.7 GetInfo

Request *Message Header:*

Flags	preserve
Dest	direct port
Reply	local reply port
FnRc	FG_GetInfo | subtype

Ioctl request *Control Vector:*

word	Type	Ioctl type code
Struct	Value	(Optional) parameter structure

SocketInfo request *Control Vector:*

word	Level	Protocol level code
word	Option	Parameter option code
Struct	Value	(Optional) Parameter value

Reply *Message Header:*

Flags	NONE
Dest	Request.Reply
Reply	NullPort
FnRc	Return Code

Ioctl Reply *Control Vector:*

word	Type	Ioctl type code
Struct	Value	result structure

SocketInfo Reply *Control Vector:*

word	Level	Protocol level code
word	Option	Parameter option code
Struct	Value	Result value

This operation requests the server for control information from the object. The request identifies the information type in the **F** field of the function code. The types are as follows:

Attributes Indicates that the data vector contains an **Attrib** structure.

Ioctl Indicates that the message is an **Ioctl** request.

SocketInfo Indicates that this is a **SocketInfo** request.

Additionally, if the **SendInfo** bit is bitwise ORed into the type value, then the client can send a parameter value in the request. This has been introduced to support certain Unix-compatible **ioctl()** operations. It must not be used as a substitute for **SetInfo** because the server may not handle this in a fully generic way.

The **Attributes** subtype is defined to be compatible with past practice, and defines no format for the control vector. The **Ioctl** subtype is included specifically to support servers which are derived from Unix code, and to provide a fully Unix compatible interface to other servers in future systems. The **SocketInfo** subtype is included to support the TCP/IP Internet server, this currently piggybacks the ioctl() functions it supports onto this subtype by means of a special protocol level.

13.7.8 SetInfo

Request *Message Header:*

Flags	preserve
Dest	IOC port
Reply	local reply port
FnRc	FG_SetInfo | subtype

Ioctl request *Control Vector:*

word	Type	Ioctl type code
Struct	Value	Parameter value

SocketInfo request *Control Vector:*

word	Level	Protocol level code
word	Option	Parameter option code
Struct	Value	Parameter value

Reply *Message Header:*

Flags	NONE
Dest	Request.Reply
Reply	NullPort
FnRc	Return Code

This operation requests that control information be set for the object. The same set of subtypes and data structures are defined for this function as for **GetInfo**.

13.7.9 EnableEvents

Request *Message Header:*

Flags	preserve
Dest	IOC port
Reply	local event port
FnRc	FG_EnableEvents

Control Vector:

word	Mask	Event select mask

Reply *Message Header:*

Flags	NONE
Dest	Request.Reply
Reply	NullPort
FnRc	Return Code

Control Vector:

word	Mask	Event select mask

Event messages *Message Header:*

Flags	NONE
Dest	Request.Reply
Reply	NullPort
FnRc	Event return code

Data Vector:

IOEvent	Event[...]	Event descriptions

Acknowledge message *Message Header:*

Flags	preserve
Dest	direct port
Reply	NullPort
FnRc	FG_Acknowledge

Control Vector:

word	Counter	Counter of last IOEvent received.

NegAcknowledge message *Message Header:*

Flags	preserve
Dest	direct port
Reply	NullPort
FnRc	FG_NegAcknowledge

Control Vector:

word	Counter	Counter of last IOEvent received.

This operation establishes a client port as a recipient of event messages from the object. As and when the selected events occur in the object, a message will be sent to the client's port. This is used to obtain raw mouse and keyboard events as well as by shells to capture special key handling (such as CTRL-C).

The request contains the event port as its message header **Reply** field, and a mask of the events to be notified in the **Mask** field. The reply will return a mask of the events which will actually be sent. This may differ from the requested set if the server does not support the given event type.

Once the event port has been registered the server will send event messages to it using a specific protocol. Each event message may contain one or more **IOEvent** structures in its data vector. The exact size of these structures is server defined, but a server will never send structures of differing sizes. It will pad smaller event structures out to the size of the largest it could send. Each **IOEvent** defines the following standard field:

Type The event type bit from the request mask field.

Counter An increasing, cyclic, sequence counter.

Stamp The timestamp of when the event occurred.

Device A variable sized field containing any event-specific data.

The **FnRc** field of each message contains a return code:

Acknowledge The client should return an **Acknowledge** message.

IgnoreLost The client need not return an **Acknowledge** message.

An **Acknowledge** message contains the **Counter** of the last event received. The server should not throw any event messages away until they have been acknowledged. As each event occurs it should be both buffered and sent to the client. When the buffer becomes close to full the server should set the **Acknowledge** return code in all events it sends until an **Acknowledge** message is received.

If the client ever received an event message with a greater **Counter** than it expected, then it should send a **NegAcknowledge** message to the server containing the **Counter** of the last message it received in sequence. This will have the dual purpose of acknowledging all events up to that point, and informing the server to re-send all subsequent events.

If the nature of the events is such that it does not matter that some messages are lost, then the server can set the **FnRc** of all messages to **IgnoreLost**, in which case the client need never acknowledge receipt of events.

13.7.10 Select

Request *Message Header:*

Flags	preserve
Dest	direct port
Reply	local reply port
FnRc	FG_Select | condition

Reply *Message Header:*

Flags	NONE
Dest	Request.Reply
Reply	NullPort
FnRc	condition

This operation allows the client to be sent a message when an object satisfies a given condition. This is present primarily to support the System library **SelectStream** and Unix compatible **select** functions, and should not normally be used outside of these.

The request encodes the condition in the **F** field of the function code using the same bits as defined in the open mode:

ReadOnly Return the reply when the object has data available for reading.

WriteOnly Return the reply when the object has space to accept a write operation.

Exception Return the reply when the object meets some device specific exception condition.

The reply is not returned immediately, but only when the given condition is met. If more than one condition was given, the exact condition or conditions met are encoded in the least significant 4 bits of the reply.

On receiving a select operation the server should check whether any of the conditions are already met, if so then a reply must be generated immediately. Otherwise the server should save the condition and the reply port with the object. It is acceptable for the server to allow only one outstanding select per object. If one or more of the selected conditions becomes true, the reply should be generated and the pending select cancelled. A select may also be cancelled if the object is ever closed, or an operation for which the select is waiting is performed, or if another select request is received. When a select is cancelled in this way the server must execute a **FreePort()** Kernel call on the reply port to dismantle any port trail.

The client should use this operation to perform selects on a number of objects simultaneously, and then wait for one or more to complete using **MultiWait**. Since it is possible for more than one object to be ready at the time that the request is issued, the client should be ready to receive several replies. It may also receive some error responses if servers are unreachable, or as selects are cancelled by the action of other clients. The timeout facility of **MultiWait** should be used, and if the timeout expires, the full set of select operations should be repeated. The client has no need to cancel pending selects, but to prevent replies interfering with later operations, the reply port should be freed.

13.7.11 Abort

Request *Message Header:*

Flags	preserve
Dest	direct port
Reply	local reply port
FnRc	FG_Abort | mode

Reply *Message Header:*

Flags	NONE
Dest	Request.Reply
Reply	NullPort
FnRc	Return Code

This operation allows the client to abort any pending read or write operations on the object. It is optional whether the server supports this operation, and at present is only used in the pipe protocol. The mode in the request is either **ReadOnly** or **WriteOnly** and causes the appropriate pending operation to be aborted.

13.8 Task control messages

The GSP operations described so far are generic to all objects. The following operations are specific to tasks in the Processor Manager and task forces in the Task Force Manager. This interface should also be presented by any server which acts as a controller of active programs.

These operations fall into two groups, those that are extensions or modifications of the standard protocols, and those that are specific to tasks and task forces. The operations which have been extended are **Create** and **Delete**, while all other indirect operations may be applied with the normal semantics except **Rename** which is not allowed. Once a task (or task force) has been created it may be opened in the normal way. However, only the set of direct operations defined here, along with **Close**, may be applied to it.

13.8.1 Create

The standard **Create** operation is used to execute a task or task force. This is modified, first, by ORing the **FF_Execute** into the function code **F** field. Second, the optional **Info** field is a **TaskInfo** structure:

Name A self-relative pointer to the canonical name of the file containing the program to be executed.

Cap A capability for the program file which allows at least read permission.

Matrix An initial access matrix for the new task, this should be ANDed with a default matrix in the server.

The timeout on the message exchange should always be large for this operation since the creation time of a task force may be long.

13.8.2 Delete

The server's response to a **Delete** operation on a running task or task force should be to attempt to force its termination. The severity with which it does this can be controlled by setting the **F** field in the function code. The exact interpretation of these values is server specific, but in general the higher the value, the more severe the termination will be. Also, the severity should be cumulative, so repeated delete attempts at one level will act with increasing severity. See chapter 11, *The System servers*, for details of how the Processor Manager handles this function.

13.8.3 SendEnv

Request *Message Header:*

Flags	preserve
Dest	direct port
Reply	local reply port
FnRc	FG_SendEnv | 1

Control Vector:

word	Items	Number of items to be sent in environment
word	Data	Total data size of items.

First Reply *Message Header:*

Flags	NONE
Dest	Request.Reply
Reply	data port
FnRc	Return Code OR 1

Environment Message *Message Header:*

Flags	NONE
Dest	data port
Reply	acknowledge port
FnRc	Err_Null

Data Vector:

String	Argv[...]	Argument array
String	Envv[...]	Environment array
Offset	Objv[...]	Object array
Offset	Strv[...]	Stream array
byte	Data[...]	Data referenced by the arrays

Acknowledge Reply *Message Header:*

Flags	NONE
Dest	acknowledge port
Reply	NullPort
FnRc	Return Code

This operation transfers a task environment, normally between a parent task and a newly created child. Both the sender and receiver must implement the complete protocol, although a server may intervene for part or all of the interaction.

The first, request message defines the size of the environment. The **Items** field counts the total number of entries in the four offset vectors in the environment message. It should count 1 for each entry in each of the four arrays, plus 1 for each of the arrays. For example, if there are six arguments, ten environment strings, ten objects and four streams, the number of items is $(6 + 1) + (10 + 1) + (10 + 1) + (4 + 1) = 34$. The **Data** field counts the total number of bytes to be allocated for the **Data** array in the environment message. For each string in the argument and environment arrays, it should count the size of the string, plus the zero terminator, rounded up to a multiple of four. For the objects it should count the size of a capability, plus the size of the pathname string, again counting the zero and rounded up to a multiple of four. For entries in the stream array it should count the same space as for an object array entry, plus space for two words.

The receiver, on getting the request message, allocates sufficient space for the environment. If this is not possible, or some other error occurs, it can return an error,

otherwise the returned message indicates that it is ready for the environment. The receiver must also return a port descriptor in the **Reply** field on which the environment will be received.

The environment itself is sent in response to the second message, to the port returned in that message. Its data vector contains four arrays of offsets into a **Data** array. the offsets are relative to the start of the **Data** array[2]. Each array is terminated by an entry containing -1. This message should also contain a reply port for the acknowledge message.

Entries in the object and stream arrays, in addition to containing offsets in to the **Data** array, may contain special values. If the entry contains **MinInt** then it is empty, and is simply a placeholder for a potential entry. If the entry has the most significant 16 bits set, then the least significant 16 bits contain an offset to some other entry in the same array. This allows two entries in the array to refer to the same object or stream without duplicating it.

Once the message has been received, the receiver must return an acknowledgment message, terminating the protocol.

Normally this protocol is not exchanged directly between parent and child, but is mediated by a server. The Processor Manager is the recipient of the initial request message, which it passes directly to the child on its own port. The remainder of the interaction then proceeds between parent and child without the Processor Manager's intervention. The Task Force Manager takes a more active role, and receives the entire environment itself. It then alters some aspects of this (runtime parameter substitution, access rights to the task force, and so on) before passing it on to each member of the new task force. To do this it sends the environment through the Processor Manager. Hence in this case the parent is actually the Task Force Manager and not the true parent.

13.8.4 Signal

Request *Message Header:*

Flags	preserve
Dest	IOC port
Reply	local reply port
FnRc	FG_Signal

Control Vector:

word	Signal	Signal to raise

Reply *Message Header:*

Flags	NONE
Dest	Request.Reply
Reply	NullPort
FnRc	Return Code

Signal Message *Message Header:*

[2]It should be noted that an earlier version of this protocol put the offset arrays in the control vector and the data in the data vector, which is where this arrangement is derived from. The maximum control vector size of 255 words proved too limiting.

Flags	NONE
Dest	signal port
Reply	NullPort
FnRc	EC_Recover—SS_ProcMan—EG_Exception—EE_Signal—signal

This operation requests that a given signal be raised in the task. The request specifies the signal to be raised, the set of valid signals may be found in **signal.h**. The reply is returned when the signal has been delivered to the task. However, it may not have actually been raised at this point.

The signal is delivered to the task in one of two ways. The first is in the form of a message sent to the task's **Signal Port**, which is established by the **SetSignalPort** private operation between the task and its IOC. If no signal port has been set, then the Exception routine in the task structure is called. This last interface is only present as a compatibility feature. It will eventually be discontinued.

13.8.5 ProgramInfo

Request *Message Header:*

Flags	preserve
Dest	direct port
Reply	info port
FnRc	FG_ProgramInfo

Control Vector:

word	Mask	Mask of information wanted

Reply *Message Header:*

Flags	preserve
Dest	Request.Reply
Reply	NullPort
FnRc	Return Code

Control Vector:

word	Mask	Mask of information which will be reported
word	Size	Number of components

Program Info Message *Message Header:*

Flags	preserve
Dest	info port
Reply	NullPort
FnRc	Return Code

Control Vector:

word	Status[...]	Status of each component

This operation requests that information on task, or task force, state changes be returned to the caller. The request specifies under what conditions the information is to be returned by setting flag bits in the **Mask** field. The only flag currently supported

by either the Task Force Manager or the Processor Manager is **Terminate**, which is reported when the task (or task force) exits.

The initial reply confirms in the **Mask** field the conditions which will cause reports. The **Size** field indicates the number of components in a task force for which status will be returned. This is an obsolete feature and the value returned will always be 1.

When a state change occurs, a second reply will be returned to the original reply port containing the information required. The **FnRc** field will contain a return code for the entire task force, or task. The **Status** vector will contain the exit status of each component (only one such result will ever be returned).

Chapter 14

Protection

The Helios protection mechanism is one of great power and flexibility. In consequence it is slightly more complex than the simpler mechanisms used by Unix or MS-DOS.

This chapter first considers protection mechanisms in general and explains why the particular mechanism chosen has been used. It goes on to describe the Helios protection mechanism in detail, and concludes with some examples of its use.

14.1 Protection mechanisms

A good protection mechanism is one which does not make itself felt until the client violates its rules. It must allow the implementation of the principle of minimum privilege, where no client has more access rights than are absolutely necessary. Finally the protection system should implement the **mechanism** but not the **policy** of any security system; it should be possible to construct several different security systems upon the same protection mechanism.

All protection mechanisms may be placed into one of two classes: access control lists and capabilities. In access control list mechanisms each protected object has associated with it a list of clients with their access rights. Any attempt to access the object is checked against the list to see whether the client is allowed to perform the operation requested. An explicit list of clients can be expensive to check, so most operating systems group clients into categories and only store the access rights for each category. For example, UNIX has three categories: owner, group and public and stores just three rights (read, write and execute) for each.

The capability mechanism is essentially the inverse of the access control list mechanism. Instead of storing a list of clients with each object, a list of objects is stored with each client. When an access is made the client's rights to the object are passed with the request and checked. In most capability based operating systems the mechanisms for identifying the object and for specifying the access rights are combined. The resulting descriptor or identifier is called a capability.

The integrity of both these mechanisms relies upon being able to prevent forgery of either the client identifiers or the capabilities. Where the underlying hardware implements memory protection it is a straightforward matter for the operating system to store these items outside the client's address space and provide some closely controlled functions to manipulate them. Where no hardware protection is present, as on

the Transputer, other mechanisms must be used, specifically sparseness and encryption.

If client identifiers are chosen at random from a large (>=64bit) number space it is unlikely that an intruder will be able to guess a valid client id. However, client identifiers need to be public knowledge, if only so that object access control lists can be manipulated. Therefore the client identifier needs to be accompanied by proof that the client is indeed who it claims to be. This in turn requires an authentication authority to check the identity and issue the proof, which must itself be chosen from a sparse number space to prevent it being guessed. Both the client identifier and the proof must accompany any request, and the pair checked for validity with the authenticator before the operation is allowed. An additional problem now presents itself: the client has passed its proof of identity over to some other program, which could now masquerade as the client. To avoid giving such hostages to fortune, the client actually must generate a special temporary proof for the server, and destroy that as soon as the operation is finished.

From the above brief discussion it should be clear that access lists leave a lot to be desired in the context of a distributed operating system. They require large amounts of data to be transferred with each request, and both clients and servers must continuously interact with the authentication authority to check the validity of clients. The presence of such a central authority violates one of the basic design goals of Helios. For these reasons Helios implements a capability based protection mechanism; how this overcomes, or avoids, the problems raised above will be described in the following sections.

14.2 Helios capabilities

A Helios capability is a 64-bit value divided into an 8-bit cleartext access mask and a 56-bit encrypted validation field. Bits are set in the access mask corresponding to the rights a client may have over the object.

When a capability is issued the access mask is combined with a known value and encrypted to form the validation field. Thus even if the cleartext mask is altered the original access rights are still present in the encrypted validation field. The true access to the object is always determined by ANDing the cleartext mask with the decrypted mask from the validation field. This allows the access rights of a capability to be restricted, but never amplified, by changing the access mask.

Whenever a new object is created a new encryption key is created and stored with the object. Capabilities in requests to access this object are decrypted with the object key and if the known value is correct the access mask is extracted and used. Protection of capabilities against forgery derives from the sparseness of the keys within their number space and not from the encryption itself.

A Helios capability only encodes the client's access rights to an object, identification of the object is by means of an absolute name string. Helios capabilities therefore differ from conventional capabilities which combine these two parts into one item. A Helios capability is virtually useless on its own, although in theory the entire object space could be searched for an object whose key decrypts the capability. While this 'glass slipper' approach may be useful for failure recovery it is clearly impractical

for normal use. For simplicity in the rest of this note the term capability should be interpreted as referring to a name plus capability pair.

Because the encryption key is stored with the object, the protection mechanism is operated only by the server containing the object and does not need any external authenticator. The key selection and encryption algorithms are entirely at the discretion of the server, as is the interpretation of the access mask bits although a system convention exists for the interpretation of these bits.

The conventional access mask for all objects contains at least four access rights bits: **r**, **w**, **d** and **a**. The bits **r** and **w** are conventional read and write permission for files, or list and create permission for directories. **d** allows the client to delete the object, or at least its directory entry. **a** allows the client to alter the protection status of the object, this will be explained in more detail later. In addition to these standard access bits, there are up to four type-specific bits. Files use one such additional bit which is the **e** or execute bit to mark executable files. Directories use all four extra bits which are used by the protection mechanism itself.

14.3 Access matrices

So far it has been assumed that to access an object a client must possess a capability for that object. It is clearly impractical for a client to keep a capability for all the objects in the system it might ever need to access (even if such a list could be made). Instead a client will initially possess just a few capabilities, for example, one for its current directory and one for the user's home directory. All subsequent object accesses are either relative to these initial capabilities or relative to no capability (a non-contextual access). Therefore a GSP message to access an object contains a name/capability pair for its context object, and a relative pathname to the target object. Either, but not both, of these may be absent: no context capability results in an absolute, non-contextual access; no pathname results in access to the context object itself.

The capability in a GSP message only specifies the access rights to the context object, some means of obtaining the client's access rights to the target object is needed. To achieve this a highly modified form of access control list is re-introduced. The extra four access bits associated with a directory place the client in one or more access **categories**. These are given the letters **v**, **x**, **y** and **z** for historical reasons. Each directory entry lists for each of these categories the access mask allowed to a client with that category of access. If the categories are viewed as rows and the access masks as columns then this forms a 4-by-8-bit **access matrix**, which is actually stored as a single 32-bit word. For example the entry for a particular file may be as follows:

v:	r	-	e	-	-	-	-	-
x:	-	w	-	-	-	-	-	-
y:	r	-	-	-	-	-	-	-
z:	-	-	-	-	-	-	-	-

The actual access rights a client has to an object in a directory depends on the access rights it has to the directory itself. Thus in the example, if the client has just the **v** category bit set in its access mask for the directory, its access mask for this file will

be **re**. However if it has both the **x** and **y** bits set, its access mask for the file will be **rw** — **w** for being in category **x** and **r** for being in category **y**.

The convention for writing access masks and access matrices is simple. An access mask is written by giving the letters of the bits which are set. Programs and procedures which accept such mask descriptions should accept the letters in any order. Sometimes it is useful to show the positions where bits are not set, these may be represented in the mask by a `'-'` character. Thus the following access masks are all equivalent:

```
rwd == rw----d- == dwr
```

An access matrix is written by listing the access masks for each category in the order **v**, **x**, **y** and **z**, separating each category with a colon. The access matrix shown above would therefore be written *re:w:r:*.

Since the access mask for a directory contains **v**, **x**, **y** and **z** bits, these may also appear in the matrix rows for a directory entry. These then indicate which access categories a client has for entries in that directory. For example, if a subdirectory entry has the matrix *wv:ry:z:z* and the client has **vx** access to the parent directory, its access mask for the subdirectory will be **rwvy** and hence will have **vy** access rights to its entries. If the file described previously were in this subdirectory, the client would end up with an access mask of **re**.

More formally, to calculate the client's access mask for an object within a directory: the rows of the object's access matrix which correspond to the access categories present in the client's access mask for the directory are extracted and ORed together to form the access mask for the object.[1] By repeated application of this transformation on the client's access mask as a path is followed through a directory network the correct access rights to a target object can be determined.

The access rights a client is allowed to a particular target object depend on two things: the access mask in the context capability, and the route used to reach it. Therefore two clients starting from the same context object and accessing the same target object may have different rights over that object. This is because the access mask in the context capability may give them different initial access categories which will result in different access matrix rows being selected from intermediate directories. Similarly two clients accessing the same target from different context directories may also have different access rights since their paths will pass through different intermediate directories whose matrices will have different effects on the final access mask. It is even possible for a client to get different access rights to a single object by following different paths to it from the same starting point.

There is no explicit meaning or hierarchy inherent in the categories themselves. However by convention **v** access should be reserved for the object's creator or owner and **z** access for public access, **x** and **y** rights may be used to provide similar functionality to the UNIX group mechanisms. Most servers will restrict non-contextual accesses (those without a context capability) to just the **z** category. However a user can alter the access matrices of directories to either shut out all public access (for example *rwva:rwx:ry:*) or to move it to another category (for example *rwva:rwx:ry:x* makes **x** the public category for a directory) or to enhance it. (For example, *rwva:rwva:rwva:rwva* gives all categories the same rights as the owner.)

[1] Even more formally this can be viewed as a boolean arithmetic matrix multiplication of the 8x4 access matrix by the 4 element category vector to yield an 8 element access vector

14.4 Capabilities in programs

Under normal circumstances the programmer does not need to concern himself with the manipulation of capabilities, these are handled for him by the System library. Capabilities are present in the **Object** and **Stream** data structures returned by **Locate**, **Create** and **Open**. An Object structure is in fact no more than a name+capability pair and may be viewed as the programmer's representation of a capability. Similarly the **Locate** function does no more than obtain a capability for a named object although it has the additional function of canonicalising the pathname. The capability stored in a Stream structure is used only if the stream has to be reopened as a result of fault recovery or when the stream is transferred to another task.

14.5 Saving capabilities

The capabilities in Object and Stream structures exist only for the lifetime of the task or program that obtained them. To allow users to retain their privileges from one session to the next capabilities must be preserved. There are two mechanisms for doing this.

The first preservation method allows a capability to be converted into an ASCII string which can then be saved in any text file. This is implemented by the System library calls **EncodeCapability** and **DecodeCapability**. An extension of this is a text format for name+capability pairs in which any name beginning with the **'@'** character is followed by a 16-character text capability and the full pathname of the object (for example, **@abcdefghijklmnop/net/IO/helios/bin**). This format is recognised and decoded by the System library so such names may be used anywhere that a pathname is expected. Hence, they may be placed in shell or environment variables, aliases and shell scripts or compiled into programs.

The second capability storage mechanism is implemented only by certain file systems and is in the form of symbolic links. A symbolic link is no more than a name/capability pair stored in the filing system. When the file server encounters a link while following a pathname it builds a new GSP request using the name/capability pair in the link as the context object and the remainder of the pathname and forwards it to the IOC. Here it is treated like any other request and forwarded to the server which supports the context object (which may or may not be the original server). In this way symbolic links may be stored for any objects in the system, they do not need to be in the same server, and they need not even be files or directories. At present only the RAM file server and the Helios Filing System support full symbolic links.

Capabilities are only worth saving if they are for long-lived objects such as files and directories. The encryption keys, and hence valid capabilities, of transient objects such as tasks and pipes are chosen at creation time and destroyed with the object or on a server or processor crash. Even files and directories are destroyed, so the saved capabilities for them may become invalid. The fact that a capability has been saved for an object is unknown to the object's server, and has no effect upon the object itself.

14.6 File system protection

The foundation of the protection mechanism is the file system where users store their long-lived capabilities. If these can be protected from other users then all else will follow. The following description, which only applies to the Helios File System, is an example of how a secure system might be constructed.

The main access point of the file system is its root directory, within this are only system directories. The access matrix on the root directory is *rz:rz:rz:rz* which restricts all non-contextual accesses to just being able to list the directory and receive **z** category to all subdirectories.

User directories are placed in a special subdirectory of the root, the user master directory. The access matrix of this directory is *rz:rz:rz:rz*, just like the root, and simply propagates the same protection policy. The matrix on user directories is *rz:rzd:rz:rz* which again restricts all accesses to **rz** except that any user with **x** access to the user master directory can delete it. Somewhat arbitrarily we have decided that the **x** category is to be used by the user administrator. He will possess a capability for the user master directory which contains **rwx** access. This gives him the right to create and delete user directories but because the **x** access is not propagated by the user directory matrices he has no more rights over the contents of a users directory than any other user. There may be several user master directories on different file systems, and different users may be the administrators for them. Each will be able to control the users in his own directory, but will not have any privileged access to the other user master directories.

When a file or directory is first created, it is given a default access matrix. For files this matrix is *rwda:rwd:rw:r*, which gives the creator all rights except execute, and the public just read access with intermediate rights going to the other two categories. The default matrix for directories is *rwvda:rwxd:rwy:rz*, ignoring the category bits, the creator gets all rights while the public can just list the directory's contents, again the other two categories get intermediate rights. The category bits present simply propagate the existing category rights down to the objects within the directory. Once created only the possessor of the **v** category can change this matrix. However, the capability returned by the **Create** operation contains full access rights, and allows the creator of an object to have complete control over it regardless of the matrix.

14.7 Processor protection

It is not only files which may be protected by capabilities, but any entity in the system. In particular processors have their own access matrices and may be protected like any other object. The **tasks** and **loader** directories have standard matrices which may be modified to block execute or load request as desired.

A special feature of processors is the way in which the access matrix attached to a processor's own name node in the name table is used. This matrix is used to modify the access rights on all non-contextual GSP requests which pass through the Processor Manager, whether originated locally or remotely.

A non-contextual request does not have a capability in it, but the access mask is set to all ones indicating an effort to get the maximum possible access. If this was simply

passed to the server the client might get more rights than he was entitled to. To prevent this the access mask in such requests is replaced by just the category bits from one of the rows in the processor's access matrix.

The selection of the row is based on the distance the message has travelled to reach the server. If the message has come from a local task the **v** row is selected. If it has come from a processor in the same cluster, then the **x** row is selected. Similarly processors in the same super-cluster will select the **y** row and increasingly more distant processors will all select the **z** row. Note that this is a **logical** distance, not physical; it is possible for two adjacent processors to be distant from one-another in the naming hierarchy.

The default matrix attached to a processor is *rwva:rx:ry:rz*. This indicates that only local tasks may create new servers within the name table or alter this matrix, these operations may not be performed remotely. With this matrix local non-contextual requests to local servers will be restricted to the **v** category while external requests will be restricted to the appropriate category depending on the source's distance. If the matrix were changed to *rwva:rv:ry:rz* then tasks in processors in the same cluster would have the same rights as local tasks to any servers.

It should be emphasised that this modification of the request's access mask is only carried out on non-contextual requests, any request which contains a context object with a valid capability is untouched since these access rights will be verified by the server which supports that object.

Chapter 15

Sockets and pipes

This chapter describes the higher-level facilities provided by Helios for interprocess communication. These consist of a **socket** interface and interprocess **pipes**.

15.1 Sockets

The Helios socket interface is intended to conform to the ARPA specification for sockets (essentially BSD4.3). The mechanism supports connection to both external networks such as the Internet and a network domain functioning within Helios.

Since the word **network** can be ambiguous, the term **external network** will be used to refer to systems such as Internet (an ethernet ring, for example) and the term **internal network** will be used to refer to a Helios processor network.

15.1.1 Posix-level calls

The support in the Posix library is intended to match the set of routines available under BSD or System V. Thus calls such as **socket**, **bind**, **accept** and **connect** function exactly as expected. Similarly **read** and **write** may be used on the appropriate sockets along with the **send...** and **recv...** functions.

The only departures from the specification are in the omission of **sethostid, sethostname, getdomainname** and **setdomainname** which on Unix are superuser only administrative calls and are supported by other mechanisms. At present **gethostid** and **gethostname** assume that the Internet domain server is functioning.

The **socket** function translates the supplied domain identifier (usually a small integer) into a server name. The domains AF_HELIOS and AF_INET can be recognised by the Posix library directly. Any other domain to server mappings are held in the file **/helios/etc/socket.conf**, which will be searched for unknown domains. This file is identical in format to its namesake under System V.

The reader is referred to the appropriate Unix[1] manuals for a full description of the socket mechanism.

[1] Unix is a trademark of AT&T

15.1.2 System library support

The Posix routines are implemented in terms of a small set of support routines in the System library, application programs should not usually need to use these directly.

The routines are described in the following sections, the reader is referred to <**syslib.h**> for the exact function prototypes.

Socket

This routine creates a new socket and returns a Helios Stream structure. The stream is not opened or connected to a server at this time, the domain server name is copied into the stream structure and the type and protocol values encoded into the stream's **Pos** field as follows:

```
stream->Pos = (protocol<<8)|type;
```

Bind

This routine contacts the server specified by the domain and opens a stream to it. This should result in the creation of a socket within the server and its binding to the supplied external network address. If no external network address is supplied then the server should bind the socket to an address of its own invention.

All the remaining calls may only be directed to a bound socket, although **Connect** and **SendMessage** will implicitly bind the socket to an anonymous address if an unbound socket is provided. It is possible to apply **Bind** several times to the same socket.

Listen

This function marks the socket as available for connections and gives an upper bound for the number of pending connections allowed. Once this call has been made, the client may use **Select** to await incoming connections.

Accept

This function waits for a connection to arrive. No timeout is given so this is a potentially infinite wait.

The result of **Accept** is a totally new stream, possibly to a different server, which represents the opened connection. The original stream remains unchanged and may be used for further **select**s or **accept**s. The new stream may be used for data transfer according to the rules of the underlying protocol.

Connect

This function creates a connection to a given address. In connection oriented sockets such as a STREAM socket this must match with an appropriate **Accept** but in DATAGRAM sockets it merely serves to bind an implicit destination address for all transmissions.

Once connected the original stream may be used for data transfer according to the rules of the underlying protocol. Note that the Helios server involved in this may differ from that in which the socket was originally created.

SendMessage

This function supports all the **send...** routines at the Posix level. It is used both to send datagrams and to send out-of-band data through stream sockets.

RecvMessage

This function supports all the **recv...** routines at the Posix level. It is used both to receive datagrams and to receive out-of-band data through stream sockets.

SetSocketInfo

This function supports the **setsockopt** Posix function, its arguments are identical. It is also used to submit **ioctl** operations by defining a new level, SOL_IOCTL, passing the **ioctl** number as the option and the parameter structure as the option value.

15.1.3 GetSocketInfo

This function supports both the Posix **getsockopt** function and **hostid, hostname, peername** and **sockname** functions. This is achieved by adding extra option codes and an extra level – SOL_SYSTEM. Like **SetSocketInfo** this function is also used to support **ioctl** calls. An extra feature is that if the **ioctl** requires that parameters be passed to the server as well as received, then these must appear in the option value buffer. The requirement for this is encoded in the **ioctl** number, so this function depends on the encoding scheme for these.

15.1.4 Message formats

This section details the Helios messages which are exchanged between the System library calls above and the server which handles the given external network domain. It is intended primarily for the authors of external network interface servers. Messages are described in the same format as in the chapter entitled *General Server Protocol*.

The following sections describe each set of message interactions under the heading of the function code which initiates it. The function codes are defined in <**codes.h**>. Most of the data structures referenced here may be found in <**gsp.h**>.

Bind

Request *Message Header:*

Flags	preserve
Dest	IOC port
Reply	local reply port
FnRc	FG_Bind

Control Vector:

IOCCommon	Common	Common part of GSP request
word	Protocol	Protocol type
Struct	Addr	Address to bind to

Reply *Message Header:*

Flags	NONE
Dest	Request.Reply
Reply	direct port
FnRc	Return Code

Control Vector:

word	Type	Object type code
word	Flags	Object Flag bits
Capability	Access	Access rights to object
String	Pathname	Canonical pathname of object

As stated above, the **Socket** call does not generate any message traffic and the first message any server will see is a **Bind** message.

The **Protocol** field is simply the **Pos** field encoded by the **Socket** call. If no address is given the server is expected to invent a name or generate an anonymous socket. It is domain-specific whether more than one client may bind to an address simultaneously.

The reply is a standard open reply. The fields in this reply are used to re-initialise the original stream structure. The **Type** should be **Socket**, the **Flags** and **Access** fields should contain valid values. The Stream structure will have been allocated by **Socket** such that it can contain a pathname of up to 100 bytes, the server must ensure that it does not return a longer name. In the message header, the reply port will be used for all subsequent operations on this socket and as with all open-style functions the return code in the message header is preserved to be ORed with the function codes of all subsequent requests.

Listen

Request *Message Header:*

Flags	preserve
Dest	direct port
Reply	local reply port
FnRc	FG_Listen

Control Vector:

word	Pending	Number of pending connects allowed

Reply *Message Header:*

Flags	NONE
Dest	Request.Reply
Reply	NullPort
FnRc	Return Code

After the **Bind** message a client which intends to accept connections must enable the socket to do this by sending a listen message. The control vector of this message consists of just a single word: the number of pending connects allowed. The reply message merely indicates the success of this in the header return code. Once the socket has been enabled for accepting connections the client should be able to select the socket for read operations to determine whether an **accept** operation is possible.

Accept

Request *Message Header:*

Flags	preserve
Dest	direct port
Reply	local reply port
FnRc	FG_Accept

Reply *Message Header:*

Flags	NONE
Dest	Request.Reply
Reply	direct port
FnRc	Return Code

Control Vector:

word	Type	Object type code
word	Flags	Object Flag bits
Capability	Access	Access rights to object
String	Pathname	Canonical pathname of object
Struct	Addr	Address of connector

Once the socket has been enabled, the client may send an **Accept** message to make a connection. The **Accept** request contains no data and consists solely of a message header containing the appropriate function code.

The first four fields of the reply consist of a standard open reply and should be used, along with the reply port and return code to initialise a new stream as if it had just been opened. The **Type** should be **Type_Socket—Type_Stream**. The address of the connector must be provided and will be returned to the application if required. The stream described need not be supported by the same server (for example in the Helios domain stream sockets are supported by creating pipes in the pipe server).

If no connections are available the server should keep the request for a short period of time (about 10 seconds) and return a recoverable error. This will cause the client to retry and thus establish an idle handshake between server and client while the connection is pending. This allows both sides to guard against the other's sudden demise.

Connect

Request *Message Header:*

Flags	preserve
Dest	direct port
Reply	local reply port
FnRc	FG_Accept

Control Vector:

Struct	DestAddr	Address to connect to
Struct	SourceAddr	Address of originator

Reply *Message Header:*

Flags	NONE
Dest	Request.Reply
Reply	direct port
FnRc	Return Code

Control Vector:

word	Type	Object type code
word	Flags	Object Flag bits
Capability	Access	Access rights to object
String	Pathname	Canonical pathname of object

The alternative to listening for and accepting connections is to connect out to a given address. Normally only the **DestAddr** field is filled in by the client, the **SourceAddr** field is only used in the Helios domain.

If the message return code is positive and the control vector is not empty then the reply is a standard open reply which, like Bind, is used to re-initialise the original Stream structure (therefore the same comments apply). The **Type** should be **Type_Socket—Type_Stream**. If there is no control vector in the message then the stream remains as it stands. As in **Accept**, if no connection can be made immediately, the server should establish an idle handshake.

SendMessage

Request *Message Header:*

Flags	preserve
Dest	direct port
Reply	local reply port
FnRc	FG_SendMessage

Control Vector:

word	Flags	Flags
word	DataSize	Size of data to send
word	Timeout	Time to wait for transfer
Struct	AccRights	Access rights
Struct	DestAddr	Destination Address

First Reply *Message Header:*

Flags	NONE
Dest	Request.Reply
Reply	data port
FnRc	Return Code

Control Vector:

word	Flags	Flags
word	DataSize	Size of data to send
word	Timeout	Time to wait for transfer
Struct	AccRights	Access rights
Struct	DestAddr	Destination Address
Struct	SourceAddr	Source Address (added by server)

Data Message *Message Header:*

Flags	NONE
Dest	data port
Reply	[acknowledge port]
FnRc	Err_Null

Control Vector:

word	Flags	Flags
word	DataSize	Size of data to send
word	Timeout	Time to wait for transfer
Struct	AccRights	Access rights
Struct	DestAddr	Destination Address
Struct	SourceAddr	Source Address
Offset	Data	Data (added by client)

Acknowledge Message *Message Header:*

Flags	NONE
Dest	acknowledge port
Reply	NullPort
FnRc	Return Code

This function supports both datagrams and the transmission of out-of-band data on stream sockets where it is supported. The protocol used here is slightly more complex than a request/reply sequence. The mechanism revolves around the client and server progressively building the contents of the control vector to describe the datagram to be sent.

The client notifies its desire to transmit a message by sending an initial request to the server consisting of just the first five fields of the structure, neither of the **SourceAddr** or **Data** fields are transmitted at this point. If the server is prepared to accept it the message is returned with the **SourceAddr** field filled in and the **DataSize** field possibly adjusted to the amount of data the server is prepared to accept. The client should add only this amount of data to the message and then send it to the data port returned in the message header **Reply** field of the reply. If this data message itself contains a reply port, the server should return a simple acknowledgment message. Normally only STREAM socket servers should do this. If the server is not ready to accept the data it should keep the client idling by returning a recoverable error to the first message every ten seconds or so.

RecvMessage

Request *Message Header:*

Flags	preserve
Dest	direct port
Reply	local reply port
FnRc	FG_RecvMessage

Control Vector:

word	Flags	Flags
word	DataSize	Maximum size of data wanted
word	Timeout	Time to wait for transfer

Reply *Message Header:*

Flags	NONE
Dest	Request.Reply
Reply	NullPort
FnRc	Return Code

Control Vector:

word	Flags	Flags
word	DataSize	Size of data
word	Timeout	Time to wait for transfer
Struct	AccRights	Access rights
Struct	DestAddr	Destination Address
Struct	SourceAddr	Source Address
Offset	Data	Data

Like **SendMessage** this function supports both datagrams and special reads on stream sockets. Unlike **SendMessage** this is a simple request/reply interaction, although it uses the same control vector structures.

A client notifies the server that it wants to read a message by sending a request containing just the **Flags**, **DataSize** and **Timeout** fields. If data is available the reply will consist of a completed control vector with all fields filled in. The **DataSize** field will indicate how much data has actually been received, and must never be more than requested. It should be noted that the **SendMessage** and **ReadMessage** protocols are designed to dovetail together in much the same way that the **Read** and **Write** protocols do. Again, if no data is available, the server should keep the client idling.

Open

Normally a server which supports sockets will not see an **Open** operation since this function is performed by **Bind**. However it is possible for the connection between client and server to timeout, in which case the automatic error recovery strategy will cause a re-open to be performed. Also, if a program passes the Stream on to another program, the recipient will connect to the server by means of an **Open** operation.

The server should therefore respond to an Open on an existing socket by returning an open reply as usual. At present servers can reject open attempts for objects which do not exist, but in future systems it may be convenient to use a **Open** request to establish connections.

Close

The standard **Close** operation should be supported. If any mode bits are set in the **F** field of the function code, then the corresponding transfer direction is shutdown.

Other GSP messages

The server is at liberty to interpret other GSP message types (**FG_Delete**, **FG_Rename**, and so on) as it sees fit. As a minimum the server should present the usual directory interface so users may browse the current list of bound sockets. This is most simply achieved by use of the Server library, see chapter 12, *Writing servers*, for further details.

A more sophisticated interface might present all, or some, of the external networks functionality through the normal Helios stream interface. So opens to particular files or directories might result in the acceptance or connection of a stream socket to a given destination. By using some textual convention to represent external network addresses within Helios file names, existing programs may use remote services and resources without change.

15.2 The HELIOS domain

In addition to an interface to external networks, Helios also provides an internal communication domain. This is intended to have, as a minimum, the same functionality as the UNIX domain. It functions only within a single Helios network, but the communicating parties may be placed on any processors within the internal network. The Helios domain supports three socket types: stream, datagram and raw.

A Helios domain address consists of a name string of up to 31 characters. The name domain is shared with the existing Helios server name space and bound sockets are placed in the local name table. Sockets may not be bound with the same name as an existing Helios server, but multiple clients are allowed to bind to the same name. If this is done it is unspecified which client will receive any incoming connection or datagram.

STREAM sockets are implemented by pipes, and may currently be used with normal **read** and **write** calls but not any of the **send... or recv...** calls. **Select** may be used both on open pipes and on sockets pending a connection.

Datagram sockets allow single messages to be delivered to a named socket with high probability. The source socket will be implicitly bound to a name in the local processor if not already bound. Select may be used to wait until datagrams may be sent or received.

Raw Sockets simply exchange Helios message ports, Any further communication is performed by calls to **PutMsg** and **GetMsg**.

15.3 Pipes

Helios pipes are the primary application level interprocess communication mechanism. They are created by the Posix **pipe()** call, by the Task Force Manager to connect task

force components, and by the Processor Manager to support STREAM sockets in the HELIOS domain.

Pipes are supported by a **pipe** server, and by the pipe protocol code in the System library. Pipes are strictly one-to-one and cannot have multiple readers or writers. In this sense, they are closer in nature to Occam channels than Unix pipes.

15.3.1 Pipe server

The pipe server acts only as a rendezvous point for the two pipe ends. It is not involved in the data transfer phase. However, it must be informed when a pipe is finally destroyed. Pipe servers are in the same class as the **ram** file server and the **fifo** server, being loaded on demand into any processor.

A pipe is created by sending a **Create** request to a pipe server. The server chosen should be resident on the same processor as one of the communicating programs. This is not essential, but using a pipe server elsewhere may introduce some extra links into the path between the ends. Once an Object has been returned from the Create, two pseudo streams should be created from it with **PseudoStream**.

These streams can be used directly by the calling program, or passed to other programs through **SendEnv**. This may occur repeatedly, however, since the pipe server sets the **CloseOnSend** and **OpenOnGet** bits in the flags returned by the **Create** operation the streams will be opened and closed as they are passed. This is so that if the pipe in not used (it could be connected to stderr for example) the server will know when the last Stream for it is closed and will be able to free its own data structures. Because the **CloseOnSend** flag is set, once a pipe has been passed on to another program, it cannot be used in the original. This ensures that at any time there are exactly two valid ends for the pipe.

15.3.2 Pipe connection protocol

Once the pipe ends have reached their eventual destinations, the pipe is not actually created until both ends attempt to read or write the pipe. An attempt to do either of these on a new pipe causes entry into the Pipe Connection Protocol. This results in a **Connect** message being sent to the pipe server, containing the descriptor of a local port which will be used for the pipe protocol. The server waits until it has received such a message from both pipe ends, and then replies to them both, passing each the port supplied by the other.

The result of this protocol is that each end of the pipe now has a message port through which it can send messages to the other end, and a local port on which it can receive messages.

15.3.3 Pipe data transfer protocol

The pipe data transfer protocol is operated within the System library, and uses the extended **Read** and **Write** protocols. The exact behaviour of the System library depends on the mode in which the pipe end has been created.

If the mode is **WriteOnly** the ports obtained from the connect protocol are used as the direct and reply ports of the Stream. All subsequent direct operation on that stream

will be sent to the other end of the pipe.

If the mode is **ReadOnly** then a pipe server process is **Fork**ed locally. A local port is created and used as the direct operation port of the Stream. Hence all subsequent direct operations will be directed to the local pipe process.

If the mode is **ReadWrite** then the same initialisation as for **ReadOnly** is performed. However, depending on the operation type, subsequent direct operations are sent either to the local pipe process, or to the other end. Only **Read**, **GetSize**, **SetSize** and read **Select**s are sent to the local process, all others are sent to the remote process.

In all these initialisations it is assumed that the other end of the pipe has a matching mode. The result of this mechanism is that all **Read** operations are sent to a the local process while all **Write** operations are remote. Hence, for a particular direction of data transfer, all requests are handled in one place: the pipe process at the reader's end. For bi-directional pipes the two directions are handled at opposite ends of the pipe.

The main purpose of the pipe process is to match the read and write operations from both ends, and to cause data to be transferred between them. To smooth the flow of data, and to allow the writer to avoid being blocked, the pipe process maintains a pending data buffer. This is initially only 2048 bytes, but its size may be queried with **GetSize** and altered with **SetSize**.

When a **Read** operation arrives at the pipe process it first checks the pending data buffer. If this is not empty then some, or all, of the data in it is returned to satisfy the request. Otherwise the request is kept, and bounced on a timeout to maintain an idle handshake in the usual way. If a **Write** request arrives while there is an outstanding read request then the pipe server returns a **Sizes** reply to the writer containing the size of the read request, and the reader's reply port. The writer will now transfer its data directly to the reader without intervention from the pipe process. Since the reader is using the extended protocol, it will respond with a **ReadAck** operation, to the pipe process, once it has received the data. In response to this the pipe process acknowledges the **ReadAck** and sends a confirming **Done** reply to the writer, completing the data transfer.

If there is no outstanding read when a write request arrives, the pending data buffer is inspected. If there is space, the pipe process uses the mechanisms available in the **Write** protocol to receive just enough data to fill the buffer. If this leaves the write request unsatisfied, it is bounced on its timeout in the usual way. When reads make more space in the buffer, more data is obtained from the writer.

Chapter 16

Program representation and calling conventions

This chapter describes how programs are represented as executable files on disc, in memory and during execution. It describes some advanced programming techniques available under Helios. First it gives a brief history of the linking process, as a prelude to explaining the Helios module table mechanism. Then it describes the actual calling conventions in more detail, explaining the display and the vector stack and how stack checking actually works. This is followed by an example piece of C, showing the assembler file produced by the C compiler and describing how this file is processed by the linker. Next, this section explains how to build Resident libraries, using the information given so far. Similar information is given for device drivers. This section concludes with a description of the Nucleus structure and how users can produce their own Nuclei.

This chapter describes primarily the Transputer version of Helios, and several examples of Transputer assembler are given. However, the basic concepts of the module table and the calling conventions used are common to all Helios systems.

The final two sections contain a more formal summary of the Helios program representation and Nucleus structure, to complement the information provided in the rest of the chapter.

16.1 Module tables

This section gives a brief description of how programs can be linked together, and how conventional linkers have developed over the years. Then it gives a description of the BCPL global vector mechanism, which is an alternative way of solving the linking problem. Finally, it explains module tables which can be thought of as a generalisation of the global vector mechanism.

16.1.1 History

Suppose a program consists of just two modules, 1 and 2. Somewhere inside module 1 there is a call to a routine **A()**, which is held in module 2. The basic problem of linking and loading programs boils down to ensuring that the call in module 1 actually

ends up at the right position in module 2. There are variations on this basic theme. For example, some code in module 1 might access a variable in module 2 instead of a function. These variations have little effect on the linking process. Figure 16.1 shows this.

Figure 16.1: The linking process

Note that the two modules are compiled completely independently, so the compiler does not know anything about routine **A()** when compiling module 1. All it can do is put some information into the intermediate object file for module 1, indicating that some external routine **A()** will be called. Other programs (usually the linker) are responsible for sorting out such external references.

The term 'module' is used here both for pieces of user code and for libraries provided by the system or written by the user. Even a simple **Hello World** program written in C, consisting of just a single user source file, has to be linked with the C, Posix, System, Utility, and Kernel libraries. There is little or no difference between calling **printf()** in the C library and routine **A()** in a separate user module.

In the early days of computing, linking was relatively straightforward. At the time, machines had little or no operating system support, and even a system monitor was a luxury. The computer could run only one program at a time and this program was always loaded at a fixed address in memory. For example, the linker would know that module 1 would reside at location 0x100 in memory. If module 1 was exactly 2048 bytes in size, then module 2 would reside at location 0x900 in memory. Hence routine **A()** in module 2 would be at location 0x950, and calls to routine **A()** could be converted to calls to absolute address 0x950. The linker could produce a binary image to be loaded at a fixed address in memory with all cross-module references resolved. There were some variations on this basic scheme. For example, calls could be made relative to the current program counter instead of to absolute addresses, but this does not affect the basic model.

As far as linking is concerned, the next development in computing involved multitasking. As computers became more powerful it was desirable to have several programs loaded in memory at the same time, with time-slicing and scheduling algorithms to share the available CPU time. Usually one of these programs would be an operating system. A program's location in memory was not known until the program was actually loaded, so the linker could not do all the work it did previously. A separate program, the Loader, was responsible for taking the binary executable with the relocation information, putting it into a free area of memory, and resolving everything. A typical piece of relocation information might contain the following:

1. At offset 0x300 within the program there is a call to a routine **A()**
2. There is a routine **A()** at offset 0x850 within the program.

Given this information and the starting address of the program, the Loader could

patch the program loaded in memory. This patching is equivalent to the work done by the linker previously. Note that this architecture is roughly equivalent to some existing machines running Helios.

The next development came with memory management hardware. Programs now operated in a virtual address space, which was mapped by some hardware onto real addresses. Bits of memory could be swapped to and from disc automatically so that programs could use more memory than was actually attached to the processor. Once again programs could be loaded at a fixed location in memory because this fixed location was inside the program's own address space, independent of the various other programs loaded in memory. Nevertheless, the use of relocation information continued. In particular it was desirable to maintain information about which routines were at which addresses because this could be used by debugging and profiling tools. Essentially this is the mechanism used by most existing Unix systems.

A fairly recent development in Unix systems is shared libraries. Previously, every copy of every program had its own version of the C library and other libraries embedded in the binary executable. This is inefficient in memory and disc usage. Hence some versions of Unix now support shared libraries similar to Helios Resident libraries. A shared library is a separate piece of code, which has to be linked with application programs at load time. The relocation information in the binary executable contains information on the shared library references, allowing it to be patched at load time.

16.1.2 The BCPL global vector

The mechanisms described so far all involve modifying the binary image of a program, either at link time or at load time, to put in absolute addresses for the various routines and bits of data that must be accessed. These techniques work, but they have their disadvantages.

1. In the absence of memory management hardware it is difficult to share bits of code between programs. Every instance of a program needs its own data and the code has to be patched to access this data. Hence the same piece of code cannot be used to access two different data areas, and it cannot be shared by two different programs.

2. If code cannot be shared, every binary executable needs to hold versions of the C library, Posix library, and so on. This is inefficient in disc space. More importantly, it is inefficient in memory usage. Typically, processors running Helios have only limited amounts of memory attached (between one and sixteen megabytes) and it is important to use this limited resource efficiently.

3. Code cannot be moved easily from one processor to another. In particular it is not possible for the system to take copies of a Resident library loaded into a processor and move it to another processor, because the library image was changed at load time.

The BCPL language eliminates the need for relocation information completely. When a BCPL program is running there will a table of words (the global vector) somewhere inside that program's own data space. A pointer to this global vector is

always held at a readily accessible location, usually in a register or at a fixed offset from the current stack pointer. Hence the global vector itself can be accessed without any need for absolute addresses in the code. The global vector holds all information that is shared between modules.

Consider an example.

1. Module 1 contains a call to a routine **A()** in a different module.

2. Module 2 has the code for this routine.

3. Both modules know that this routine has been allocated slot 200 of the global vector. When the program is loaded, some initialisation code inside module 2 will put the current address of the routine in the correct slot of the global vector. Information about the slot allocation comes from a separate header file which must be included in all source files.

4. The code in module 1 can always access the global vector, either through a register or a fixed position on the stack. Hence it can access slot 200 of this vector and get the address of routine **A()**. The actual routine can now be called by indirecting through this address.

For this mechanism to work, various conditions must be met. Firstly, the various modules making up the program must have the same information about the global vector layout. If one module thinks routine **A()** is held in slot 199 and the other module uses slot 200, the program will not work. Secondly, the compiler must produce some module initialisation code together with the real code, so that the slots of the global vector can be filled in, and the system must call all the initialisation code in the different modules before actually starting the program. There must be some way of storing the global vector address where it can always be accessed readily, either in a register or somewhere on the stack. To permit sharing and moving of code the code must not be self-modifying. With most modern processors this can all be achieved fairly easily.

The global vector mechanism makes it relatively easy to use shared libraries. At load time the initialisation code of the shared libraries will be called, just like the initialisation code of user modules. This code can place the address of system routines in the global vector. Hence user applications can call system routines in exactly the same way as other user routines. Figure 16.2 illustrates the use of global vectors in BCPL.

Figure 16.2: The BCPL global vector

There are two programs loaded in memory. Each of these has some code which is used only by itself. Both programs have their own global vector, representing their own private data areas. Some parts of the global vector point at system routines in a shared library. Other parts point at routines internal to each program.

16.1.3 Module tables

The global vector mechanism works for the BCPL language. However it does have some limitations. Most importantly, it relies on the programmer to define the layout of the global vector. To make the mechanism more usable, and in particular to support languages other than BCPL, it must be possible to generate the global vector automatically.

The Helios mechanism is essentially an extension to the global vector. The module table is a vector of pointers to module data areas, also known as static data areas. For example slot 7 in the module table usually refers to the C library. This slot contains a pointer to another piece of memory containing the various bits of information associated with the C library. The fifth word of this piece of memory holds the address of the **putc()** routine. Hence an application can call **putc()** by locating the module table, which is typically held in a register or on the stack; indirecting through the 7^h slot of the module table to get the C library data area; and indirecting again through the 5^h word in this data area. This is illustrated in Figure 16.3.

Figure 16.3: Calling through the module table

The compiler has to do the following work to support module tables.

1. For every routine that might be called from other modules, a slot has to be allocated in this module's data area. In addition it is necessary to produce a name table describing which routine has been allocated which slot in the data area.

2. Every piece of data that can be accessed from other modules needs to have space allocated within this module's data area. This also needs to be put into the name table.

3. Every piece of static data (private to this module) needs to have space allocated within the data area. There is no need to put this information into the name table.

4. A call to an external routine **A()** always involves the following stages:

   (a) Extract the module table pointer

   (b) Index into the module table to get the data area for the module containing routine **A()**. At compile time it is not known which slot of the module table will be used for that module, so this has to be put in by the linker.

   (c) Index into the relevant module's data area to get the address of routine **A()**. Again the required offset will not be known until all the modules are linked together.

   Using all these indirections may seem an expensive way of calling another routine. In practice on Transputers they involve, typically, only five bytes worth of instructions.

5. Accessing external data involves much the same process, with indirections to get to the relevant module's data area.

6. The size of this module's data area has to be stored at a fixed location within the executable so that, when the program is loaded, the system can allocate a sufficiently large area of memory.

7. Some initialisation code must be produced to fill in the module's data area. This code will be called by the system before the actual program is started.

The jobs to be done by the linker are as follows:

1. Read in all the files passed as arguments, including Scanned libraries. Build a global name table of all the names in all the modules. Files may contain a single module, for example the result of compiling a single user source file. Alternatively they may contain many modules, for example a Scanned library.

2. Determine the minimum set of modules needed to link the program. This involves eliminating any parts of Scanned libraries that are not actually used.

3. Assign module table slot numbers for all the modules that will be incorporated in the final executable. Resident libraries have pre-defined slot numbers. Other modules are assigned slot numbers starting from 1, skipping any slots that have been used already for Resident libraries.

4. For every module, for every access to an external routine or piece of data, fill in the module's slot number and the offset within the module's data area corresponding to that routine.

5. On processors with variable length instruction sets, notably Transputers, perform a final code growing phase. Until now neither the compiler nor the linker could know whether access to a particular module slot or data area involved one, two, or more bytes worth of instructions. Hence the final code can be produced only at this time.

6. The end result is a binary executable that can be written to a file.

When the program is loaded the system has to do the following jobs.

1. Work out the size of the module table, in other words how many modules there are in the executable program and hence how many slots there must be in the module table.

2. For every module, work out the size of its data area.

3. Allocate a single piece of memory large enough for the module table and for all the data areas. The program's initial stack and heap are also placed in this data area. The initial stack and heap size are held at a fixed offset within the program.

4. Fill in the module table slots with pointers to the data areas, all within the same piece of memory.

5. Call all the module initialisation routines with a pointer to this module table, so that all the various pointers and bits of data get initialised. In fact to allow for possible circular dependencies the initialisation routines are called twice, the first time with an argument **0**, the second time with a non-zero argument. During the first pass the module initialisation code should initialise data internal only to itself, which is usually most if not all. During the second pass any data relying on other modules. For example, suppose module 1 has a static variable which is automatically initialised to the address of a function in module 2. Until the first pass has reached module 2 there is no way for module 1 to know the address of this function.

6. Call the program's entry point, **c0.o** or **s0.o** depending on how the program was linked, which will be at a fixed position within the executable program.

Note that only a single piece of memory is used to hold the module table and all data areas. The module table for a simple C program might look something like Figure 16.4.

Figure 16.4: A C program's module table

There are problems when building Resident libraries. When, for example, the C library is built to produce a binary object that can be loaded at any time, it must have some way of accessing its own data area. For example, it must have some way of accessing the table of **FILE *** pointers so that it can initialise these. The library's data area can only be accessed through the module table, so when the C library is built it must know which module table slot to use to access its data area. Similarly, if the C library is to be able to call other Resident libraries such as the Posix library through the module table, it must know the slot number and data area layout of these other libraries. The solution here is to pre-allocate certain module table slots to specific Resident libraries. For example, the Kernel library always uses slot 1 of the module table, and the C library always uses slot 7.

16.2 Calling convention

Helios defines a basic calling convention which is assumed by the system libraries. The exact details, in other words which instructions are used, vary from processor type to processor type. This subsection describes mainly the Transputer conventions. Other processor versions will use similar conventions, but processor specific documentation should be consulted for exact details.

When a procedure is entered the stack pointer should point at the return link, in other words the address of the instruction to be executed when the call returns. In the next location there should be a pointer to a display area. The nature of this display area is not defined by Helios, as different languages have different requirements. However,

the first word in the display area should be the module table pointer. Any arguments should be on the stack following the display pointer. This is shown in Figure 16.5.

Figure 16.5: Helios calling convention

For example, on Transputers the following code fragment can be used to call the **putc()** routine which occupies word 5 of the C library's static data area, where the C library uses slot 7 of the module table.

```
ldl      1          -- load display pointer off stack
ldnl     0          -- load module table pointer from display
ldnl     7          -- indirect to C library's static data area
ldnl     5          -- indirect to get putc routine
gcall               -- call the routine
```

The calling convention is designed to be usable from a wide range of languages. Since the system libraries assume no more than this convention these libraries can be called from all the languages. Should a language implementation need to use a different calling convention then some extra work will be required before the system can be called. For example the Meiko Fortran compiler uses its own calling conventions, and the Fortran library has routines **POSIX_WRITE()** and so on which call the actual libraries after a conversion process.

Once a routine has been entered it may do anything it likes with the stack pointer. Usually the position is adjusted to hold variables local to this routine, possibly including temporary ones generated by the compiler. A routine can return by restoring the stack pointer to its initial value, pointing at the return link, and executing the **ret** instruction. For example the following code fragment shows a routine that allocates space for one local variable, sets it to five, and returns immediately.

```
.routine:                -- entry point label
         ajw        -1    -- space for one local variable
         ldc         5
         stl         0    -- assign five to the local variable
          ...             -- do useful work
         ajw         1    -- restore the stack pointer
         ret              -- and return to the calling routine
```

For languages that do not require a display the module table pointer itself can be used as the display, because slot 0 of the module table always points to the module table itself.

16.2.1 C calling convention

By default the C compiler uses a vector stack. Consider the following piece of code.

```
void a_routine(void)
{ int     x;
```

```
    char    big_array[256];
    int     y;
}
```

One way of allocating these local variables is simply in order of declaration. Hence there would be a single word on the stack for integer **x**, then 256 bytes for the array, and another word on the stack for the second integer. This is shown in Figure 16.6.

Figure 16.6: Inefficient stack usage

Using a stack laid out like this is inefficient, because the local variable **y** cannot be accessed easily. Many processors have instructions specifically for accessing local variables close to the current stack pointer. Having local variables far away from the current stack pointer increases code size and slows down the program. The Helios C compiler uses a vector stack instead. All arrays and data structures larger than eight bytes, in other words larger than the size of a double precision number, are placed in a separate piece of memory, the vector stack. Typically this is placed at the opposite piece of memory from the stack, in other words the normal stack grows in one direction and the vector stack grows in the other direction. This is shown in Figure 16.7

Figure 16.7: The C vector stack

Using a vector stack makes it easy to implement stack checking. When a routine is entered it can check whether or not the normal stack and the vector stack now overlap. If so then there has been a stack overflow. If the vector stack is disabled then the compiler cannot generate code to perform stack checking. Note that the vector stack is not part of the standard Helios calling convention, and hence it must be disabled when compiling Resident libraries or device drivers which might be called from languages other than C. The C compiler accepts pragmas to control whether or not stack checking is enabled.

Job	Compiler Driver **c**	Compiler **cc**	Source code
Enable stack checking	default	default	`#pragma -s0`
Disable stack checking	-Fs	-ps1	`#pragma -s1`
Enable vector stack	default	default	`#pragma -f1`
Disable vector stack	-Ff	-pf0	`#pragma -f0`

The vector stack works as follows:

1. The display used by the C compiler consists of two words. The first word is the module table pointer, as per the Helios calling convention. The second word is the current vector stack pointer.

2. If a routine does not allocate space on the vector stack then the display is passed unchanged to other routines.

3. Otherwise the routine must allocate a new display on its local stack. This new display will inherit the module table pointer from the old one, and it will have a new vector stack pointer consisting of the old one plus the space allocated by this routine. Any routines called from here are passed the new display instead of the old one.

4. No work is required when returning from a routine because previous stack frames hold displays with the correct information about vector stack usage.

16.2.2 An example

This subsection illustrates some of the mechanisms described so far, using a small piece of C and describing the code produced by the compiler and the linker. The C program is simple.

```
#pragma -s1

extern int global;
static int local;

int main()
{
  local = global;
  printf("Hello world.\n");
}
```

Note that stack checking is disabled to keep the code produced by the compiler somewhat simpler.

The compiler

The first code produced by the C compiler looks something like this:

```
        align
        module  -1
.ModStart:
        word    #60f160f1
        word    .ModEnd-.ModStart
        blkb    31,"main.c" byte 0
        word    modnum
        word    1
        word    .MaxData
        init
```

This information is a Module header as defined in the header file **module.h**. There is a magic number 0x60f160f1 used to identify the type of the module, in this case an ordinary program. This is followed by the size of the code in this module, which will be evaluated automatically by the linker. The name of the module is part of the code, and this information is used by programs such as **objed**. There is a word to hold

this module's slot number in the module table, which will be filled in by the linker. The next word holds version numbers for Resident libraries and can be ignored. The MaxData field will be the size of this module's static data area.

The code for routine **main()** looks something like this.

```
        align
        word #60f360f3,.main byte "main",0 align
.main:
        ajw     -1          -- space on stack for temporary variable
        ldc     ..1-2       -- get address of "Hello world."
        ldpi
        stl     0           -- store it in the temporary variable
-- Line 7 (main.c)
-- Line 8 (main.c)
        ldl     2           -- load display pointer off stack
        ldnl    0           -- get module table
        ldnl    @_global    -- get data area for required module
        ldnl    _global     -- access the actual variable
        ldl     2           -- load display pointer again
        ldnl    0
        ldnl    modnum      -- move to own static data area
        stnl    ..dataseg+0 -- and write to variable local
-- Line 9 (main.c)
        ldl     0           -- fetch "hello world." string
        ldl     2           -- extract display pointer
        call    .printf     -- call from stub below
        ajw     1           -- clean up stack
        ret                 -- return to parent routine
```

There is some header information at the start of the routine, consisting of the magic number 0x60f360f3 and the name of the routine. This will be embedded in the final code. It can be used by, for example, the stack overflow handler to report the name of the routine that caused the stack error. However, it will use up space, in this case a total of 12 bytes. To keep programs as small as possible it is possible to suppress these names. The relevant options are: **-Fg** for the compiler driver; **-pg0** for the C compiler; and #**pragma -g0** for source code.

The program contains a single literal value, the string "Hello world.", that is read-only. Hence this string can be left in the code part of the binary executable, and it does not need to be copied into the static data area.

```
-- Literals
        align
..1: -- 1 refs
        byte    "Hello world.\n"
        byte    0
        align
```

There is a call to an external routine **printf()**. All such calls go through calling stubs, which are responsible for indirecting through the module table and so on. Using calling stubs has several advantages. Most important, if the external routine is called from more than one place in the code then using calling stubs reduces code size. Also it makes life easier for the compiler: when there is a call to routine **A()**, which may or

may not be in the same module, the compiler can always generate the instructions **call .A**. If the routine is declared in the same module no extra work is required; otherwise the compiler simply generates a calling stub; in theory this permits all programs to be compiled in just one pass of the compiler.

```
-- Stubs
        align
.printf:
        ldl     1          -- get display pointer off stack
        ldnl    0          -- get module table pointer
        ldnl    @_printf   -- find static data area
        ldnl    _printf    -- get the address of routine printf
        gcall              -- and call it
```

The next piece of code produced by the compiler reflects the static data area of this module. This module defines a single static variable **local** requiring one word of static data area. In addition it defines one routine **main()** that is accessible from other modules. The **global** statement declares **_main** as a global name, so that the linker can resolve cross references. The **data** statement allocates one word of store in the static data area, which will be filled in by the module initialisation code with the address of routine **main()**. Hence the total size of the static data area is two words.

```
-- Data Area
        data    ..dataseg 1
        global  _main
        data    _main 1
```

The final piece of code is responsible for initialising the static data area for this module. Its main job is to get the address of routine **main()** and put it into the second slot of this module's static data area. The final line specifies the end of this module. **.ModEnd** is referred at the start of the code as a way of determining the code size.

```
-- Data Initialisation
        align
        init
        ajw     -2
        ldl     3          -- get display pointer
        ldnl    0          -- get module table pointer
        ldnl    modnum     -- get this module's data area
        stl     1          -- store in temporary variable
        ldl     1
        ldnlp   ..dataseg
        stl     0
        ldl     4          -- get argument to initialisation code
        cj      ..3        -- if 0 then first pass of initialisation
                           -- second pass initialisation code
                           -- would go here
        j       ..4
..3: -- 1 refs
        ldc     .main-2    -- get address of main
        ldpi
```

```
        ldl     0
        stnl    1           -- store in slot 1 of the static data area
..4: -- 1 refs
        ajw     2           -- initialisation code finished
        ret
        data    .MaxData 0
        align
.ModEnd:
```

Note that the code shown here is the assembler source produced by the compiler, in other words the **.s** file. Usually this is passed through the assembler to tokenise it, replacing the text by binary opcodes and special linker directives, to produce a much smaller **.o** file. The information contained in the **.o** file is the same as the information in the **.s** file.

The linker

The linker starts by reading in all the files specified: **c0.o**, the program's own object file and also **helios.lib**, and **c.lib**. From this it can work out which modules are required to build the executable binary, and allocate module table slot numbers.

Slot	Contents	Type
3	c0.o	Program header
1	Kernel	Resident library reference
2	System	Resident library reference
4	Utility	Resident library reference
5	Floating point	Resident library reference
6	Posix	Resident library reference
7	C library	Resident library reference
8	main.c	user module

Note that the final program layout does not necessarily reflect the slot number allocation. In particular the program start-up code **c0.o** is the first part of the executable file, but has been given slot 3 in the module table because this is the first free slot. The various Resident libraries have fixed slots in the module table. Finally user modules and Scanned library modules are assigned any free slots that are available.

Once all the modules have been assigned slot numbers the linker can go through the code and fill in suitable information. For example, a calling stub to access routine **putc()** in the C library would be changed from:

```
ldl     1           -- get display pointer
ldnl    0           -- load module table pointer
ldnl    @_putc      -- module containing putc
ldnl    _putc       -- offset of putc within data area
gcall
```

to:

```
ldl     1
ldnl    0
ldnl    7
ldnl    5
gcall
```

or, more correctly, to the binary sequence `0x71 0x30 0x35 0x37 0xf6`.

Once all the module table and static data areas have been resolved it is possible to produce the final code. The results can now be written to a file which looks something like this:

1. An image header. This contains a magic number 0x12345678 to specify that the program is an ordinary executable rather than a compiled task force or some other strange file. There is a flags field to hold information such as the processor type. Then there is a size field giving the total size of the whole executable program.

2. A **Program** structure. This contains a module header for the start-up code **c0.o**, the initial stack and heap sizes, and the program's entry point as an offset.

3. The required modules. These may be Resident library references or real code. The system can distinguish between these by examining the type field at the start, since ordinary modules and Resident library references have different numbers to represent their types.

The Loader

When it is time to execute the program it has to be loaded into memory. This is the shared responsibility of two Helios servers, the Loader and the Processor Manager, which are present on every Helios processor. For convenience the term Loader is used here for both these servers.

The first step is to read the image header, which allows the Loader to check that the file is really an executable program and to determine its size. A suitable area of memory can now be allocated and the program can be loaded off disc.

Once in memory the Loader examines every module. The first module comes immediately after the fixed-size system header. Every module has its size built into the binary. Hence the Loader can access every module. If a module is actually a Resident library reference then the Loader ensures that this is in memory, getting it off disc or from a neighbouring processor if necessary, before continuing. As every module is examined the Loader keeps track of the static data areas required and the total number of modules. With this information, plus the stack and heap sizes held at the start of the program, the Loader can allocate another suitable area of memory. The positions of the stack and initial heap within this area are readily determined, as is the start of the module table. The various slots in the module table can now be filled in with pointers to every module's static data area. The module initialisation routines can now be called twice, and the program is fully initialised and ready to run.

16.3 Resident libraries

Helios Resident libraries are separately compiled and linked pieces of code which can be used by applications without embedding all the code in the application's binary image. They have the following characteristics:

1. Resident libraries always reside in the directory **/helios/lib**. Hence they can be installed only by the system administrator, and ordinary users do not normally produce their own.

2. Resident libraries usually have an associated **.def** file used by the linker. This file defines the library's module table slot number and its static data area, giving the linker the information it needs. This **.def** file is in addition to the main file containing the actual code. For example, **Clib** contains the code for the C library and **Clib.def** contains the library definition.

3. Application programs can contain Resident library references. When a program is loaded the system checks for such Resident library references and will automatically fetch libraries off disc or from neighbouring processors.

4. Resident libraries always have fixed slot numbers within the module table. If this were not the case then the library could not access its own private data, and other libraries would be unable to call its routines.

5. As far as Helios is concerned a Resident library is a single module with just one slot in the module table. In practice any non-trivial library has to be built from more than one source file. Hence some special work is required to build Resident libraries.

This section describes how to build simple Resident libraries. Please note that application programmers should normally use Scanned libraries instead. Resident libraries are used mainly by system programmers, and for most user applications there is little or no benefit in producing Resident rather than Scanned libraries.

16.3.1 Slot numbers

Every Resident library needs a slot number. The first 25 slots have been allocated already. The table below shows slot number allocations for Transputer versions of Helios.

Slot	Purpose
1	Kernel
2	System library
3	Server library
4	Utility library
5	Floating point library
6	Posix library
7	C library
8	Floating point part of C library
9	X library
13	Modula 2 library
15	BCPL library
18	Profiling library
19	Debugging library
21	Fortran library
24	Resource Management library

To allow for future expansion Helios reserves slots 25 to 50. Slots 51 to 100 are reserved for third parties developing general purpose libraries, and Perihelion Software should be consulted for details of how to obtain such a slot. Slots 101 onwards may be used by all programmers.

Problems can arise only if a program needs two Resident libraries that use the same slot. For example, suppose both the C library and the Fortran library had been given slot 7. If a program had to be linked with both these libraries for some reason then both libraries would attempt to use the same piece of memory for their data areas, and the program would crash almost immediately. Because all the main libraries have been allocated separate slots such problems are unlikely to arise for the foreseeable future.

16.3.2 Compiling the sources

Some of the code produced by the C compiler is inappropriate for Resident libraries.[1] In particular, the compiler must not generate code for the following:

1. There should not be a module header at the start of the file. The final library must have a single module header, typically produced from an assembler file, instead of a separate module header for every source file making up the library.

2. Similarly the module terminator **.modend** should not be produced. Shipped with Helios is the file **/helios/lib/modend.o** which should be linked at the end of the Resident library.

3. Most of the static data area layout should be determined separately. In particular the offsets of the various functions and variables that can be accessed by application programs and other libraries should be specified in the assembler file and not generated automatically by the C compiler. Otherwise small changes to the C code could cause the compiler to produce a different and incompatible layout and the library would no longer work with existing programs.

4. Calling stubs to routines outside the Resident library must also be put into the assembler file, and not produced by the compiler.

Note that the C source code is no different for Resident libraries. All that has to change is the intermediate object files generated by the compiler.

The C compiler has an option specifically for building Resident libraries, which takes into account the above requirements. Using the compiler driver this can be achieved as follows:

```
c -m -c matrix.c -o matrix.p
```

Note that code produced for Resident libraries is usually given a **.p** suffix instead of a **.o** suffix, to make it easier to keep track of the library's purpose. If the compiler is invoked explicitly instead of through the compiler driver the following commands can be used:

[1] This subsection describes only Resident libraries written in C. Other languages may support similar features to the ones described here, permitting Resident libraries to be written in those languages. The reader is referred to the appropriate language manual for more details.

```
cc -l -d__HELIOS -d__TRAN -d__HELIOSTRAN matrix.c -s matrix.s
asm -p -o matrix.p matrix.s
```

Code compiled for Resident libraries should be compiled without stack checking and without the vector stack, using the compiler options or pragmas.

16.3.3 The library assembler file

The first component of a Resident library is normally written in assembler. In practice this assembler file uses macros throughout, to ensure that the same source file can be used for many different processors. A typical file might look like this:

```
include basic.m
include library.m
include sem.m

Resident
[
        name    Maths
        slot    105
        version 1000

        static
        [
                extern  func    invert_matrix
                extern  func    fft
                extern  func    integrate
                extern  word    maths_errno
                extern  func    solve_eqn

                -- additions must go here

                -- initialise statics
                code
                [
                        initword maths_errno 0

                        -- call into C source to initialise statics
                        libcall mathslib_init
                ]
        ]

        uses    Kernel
        uses    SysLib

        stubs
        [
                -- kernel
                stub    Wait
                stub    Signal
                stub    InitSemaphore
                stub    TestSemaphore
```

```
                -- syslib
                stub    Malloc
                stub    Free
        ]
]
-- end of mathslib.a
```

This assembler file will be passed through the Assembler Macro Pre-Processor **AMPP**, a copy of which will be required for building Resident libraries. The syntax of the pre-processor is described in the **AMPP** manual, as are the various macros supplied in the macro header files. The first three lines simply include some macro header files that are required. Then the **Resident** macro is invoked to start building a Resident library. This should contain the following fields:

1. The library name. This must be the final name of the library inside the directory called **/helios/lib**. Programs can be linked with the definition file **Maths.def**, and the Loader will automatically fetch the file **Maths** when this library is required.

2. A **version** field. This is currently unused. In future the Loader may exploit it to ensure that programs linked with a later version of a library are not run with an old version, which might be lacking some routines.

3. A static section. This defines the layout of the public part of the static data area associated with this library. All functions in the library that can be accessed from applications or from other libraries should be declared here as **extern func** items. In this example four such routines are made accessible. The code for these routines is, of course, still in the C source files. Similarly all variables that must be accessible should be listed here. Integer-sized data items should be declared as **word**. Alternatively vectors or data structures could be specified. The main Helios data structures are already defined in the macro header files shipped with **AMPP**.

   ```
   extern vec 256     byte_array
   extern struct Sem  maths_lock
   ```

   It is **ESSENTIAL** that the ordering of this static section is not changed. Changing this section, for example putting the routines into alphabetical order or adding new routines in the middle, changes the static data area layout. If an application has been linked with this library already then the code will expect to find the **fft()** routine at offset 1 within the data area, so if the library programmer rearranges this area and puts the **integrate()** routine at offset 1 instead then the application will almost certainly crash.

   If new routines or variables are added to the library then they must be placed at the end of the static data area. Routines and variables cannot be removed from Resident libraries without, potentially, causing existing applications to stop working. If a particular routine is no longer needed then it should be replaced with a dummy routine of the same name, which typically only returns an error code.

4. After the function and variable declarations there should be a small piece of code. This will be called during the first pass of the module table initialisation. The code can contain macros such as **initword**, **initptr**, and **inittab** to initialise variables.

   More generally it is possible to call an ordinary C routine to initialise variables. However, since this code is called during the first pass of the initialisation the C routine must be entirely self-contained and cannot access routines or data in other modules.

5. After the initialisation code the various other libraries called from inside this one must be listed, with the **uses** statement.

6. Finally all the routines called from inside this library must be given calling stubs. If a calling stub is missing then when the library is linked the linker will produce a warning message about an undefined function.

Resident libraries can usually be built simply by taking the above example, changing some names and the slot number, and adding to the list of externally accessible functions and calling stubs.

16.3.4 makefile

A makefile for a simple Resident library would look something like this:

```
.suffixes:
.suffixes: .a .c .p

.c.p:
        c -m -c $*.c -o $*.p

default: Maths Maths.def
        echo Maths library built.

Maths:  mathslib.p matrix.p fourier.p simuleqn.p integrate.p
        asm -f -n$@ -o$@ $^ /helios/lib/modend.o \
                /helios/lib/kernel.def \
                /helios/lib/syslib.def

mathslib.p: mathslib.a
        ampp -o$*.s -dhelios.TRAN 1 -i/helios/include/ampp/ \
          /helios/include/ampp/basic.m $<
        asm -p -o$@ $*.s
        rm $*.s

Maths.def: mathslib.a
        ampp -o$*.s -dmake.def 1 -dhelios.TRAN 1 \
          -i/helios/include/ampp/ /helios/include/basic.m $<
        asm -p -o$@ $*.s
        rm $*.s
```

Note how the assembler file is processed twice, once to generate the initial part of the library and once to generate the **.def** library definition file. The Resident library macros in **library.m** check for the flags **make.def** and produce code accordingly.

16.4 Device drivers

Device drivers are another special kind of binary object. A device driver is a piece of code that is loaded from inside an application while the application is running. This is different from Resident libraries which are loaded before the application starts up. The main use of device drivers is when writing Helios servers for a variety of different hardware. For example the Helios file server is independent of the underlying file system hardware, and loads a hardware specific device driver to interact with SCSI discs, M212 disc controllers, and so on.

The linker does not know which device drivers will be loaded by a program. This causes a special problem: because the linker has no advance knowledge of the device driver it cannot allocate a module table slot to that driver, nor can it allow for another static data area. This has the following effects on the device driver:

1. A device driver can have **no** static data at all. In Helios static data is always handled using the module table mechanism, and device drivers do not have any space within the module table.

2. Device driver routines cannot be accessed directly. The addresses of these routines would normally be placed in the module table, but the device driver has no space within the module table. Instead the device driver has a specific entry point **DevOpen()** whose offset within the code is held within a header structure.

3. The device driver can only make calls to Resident libraries that have been linked to the application. Otherwise, drivers expecting the relevant library to be bound into the module table will erroneously indirect through a non-existent or re-used slot in the table. This will cause the application to crash.

Building device drivers is in many ways similar to building Resident libraries, involving the same options to the C compiler and a separate assembler file. A typical C source file for a device driver to control, for example, a model railway would look like this:

```
#include <device.h>
#include "railway.h"   /* define hardware specific structures */

static void DeviceOperate(DCB *device);
static void DeviceClose(DCB *device);

RailwayDCB *DevOpen(Device *dev, RailwayInfo *info)
{ RailwayDCB *dcb = Malloc(sizeof(RailwayDCB));
  dcb->DCB.Device   = dev;
  dcb->DCB.Operate  = &DeviceOperate;
  dcb->DCB.Close    = &DeviceClose;
  return(dcb);
}

static word DeviceClose(RailwayDCB *dcb)
{
    /* shut down the railway */
```

```
  return(Err_Null);
}

static word DeviceOperate(RailwayDCB *dcb, RailwayReq *req)
{
    /* perform the work specified by the request */

  return(Err_Null);
}
```

To load a device driver Helios provides the routine **OpenDevice()** which takes two arguments: the name of the device driver, for example **railway.d**; and some hardware specific information, possibly NULL. The specified piece of code is loaded off disc, usually from the **/helios/lib** directory, and the **DevOpen()** routine in the device driver is called. This routine can be accessed because its offset within the driver code is held in some header information at the start of the code. The **DevOpen()** routine allocates a device control block of some sort, which contains as its initial part the **DCB** structure defined in the header file **device.h**, usually followed by some additional hardware specific data. Any variables that would normally be allocated statically must be put into this control block. The **DevOpen()** routines must also fill in the **Operate** and **Close** fields with addresses of suitable routines. The close routine is called when the server is shutting down. The operate routine is called when the server needs to perform some hardware specific job, details of which can be encoded in the request argument.

Apart from the restrictions that there can be no static variables within the device driver and that its routines are called indirectly, usually through the operate routine, the device driver contains standard C code. However, a separate assembler file is needed to actually build the device driver. A typical assembler file would look like this:

```
include device.m

        Device Railway.Device 1000

        uses    Kernel
        uses    SysLib
        uses    Util

        stubs
        [       -- kernel
                stub    InitSemaphore
                stub    Wait
                stub    Signal
                stub    TestSemaphore

                -- syslib
                stub    Delay

                -- util
                stub    memcpy
                stub    strcpy
                stub    strlen
```

```
                stub    strcmp
        ]
-- End of raildrvr.a
```

The assembler file for a device driver is rather simpler than for a Resident library. A macro header file is needed to define the **Device** macro. This macro takes just two arguments, a device name and a version number, neither of which are important. The macro expands to a special header structure which must be the first part of the final binary object. There is no need to produce a static data area layout, since device drivers do not have any static data. It is necessary to produce calling stubs and library references in the same way as for Resident libraries.

The makefile for a device driver would look something like this:

```
.suffixes:
.suffixes: .a .c .p

.c.p:
c -m -c $*.c -o $*.p

default: railway.d
echo Railway device driver built.

railway.d: raildrvr.p railway.p
asm -f -n$@ -o$@ $^ /helios/lib/modend.o \
/helios/lib/kernel.def \
/helios/lib/syslib.def \
/helios/lib/util.def

railway.p: railway.c

raildrvr.p: raildrvr.a
        ampp -o$*.s -dhelios.TRAN 1 -i/helios/include/ampp/ \
          /helios/include/ampp/basic.m $<
        asm -p -o$@ $*.s
        rm $*.s
```

16.5 The Nucleus

The Nucleus is the core component of Helios, running on every Helios processor in a network. Application programmers rarely need to worry about the Nucleus. System programmers will occasionally need to produce their own versions of the Nucleus to meet special requirements. For example, in a standalone system it is usual to boot a small monitor program from ROM which then fetches a special Nucleus from the initial sectors of a hard disc. This special Nucleus needs to incorporate the file server, the hard disc device driver, and the **devinfo** configuration file as well as the standard Nucleus components. Otherwise the newly-booted processor would be unable to start up the rest of the system by using the **init** program and the **initrc** file.

The default Nucleus for Transputers includes the following components:

1. The Kernel is responsible for maintaining the processor hardware. This part of Helios is the first to start up in a newly booted processor. It initialises the

hardware, for example determining the amount of memory on the processor, and then transfers control to the higher-level Processor Manager. The Kernel is unusual in that, to some extent, it is a program as well as a Resident library: it starts off its own threads to control specific bits of hardware, it has its own memory pool, and so on; it also contains library routines for low-level message passing, semaphore synchronisation, and similar low-level operations. However, the Kernel is not a full program in the sense that it does not have its own module table.

2. The system library is a Resident library, providing the interface between Helios applications and servers. The system library is not usually invoked directly, but it is used by higher levels of the system such as the Posix library.

3. The server library facilitates the job of writing Helios servers.

4. The utility library contains various miscellaneous routines which are needed by other parts of the Nucleus (routines such as **memcpy()** and **Fork()**).

5. **nboot.i** is a little utility used for booting other processors.

6. The Processor Manager is a program, not a Resident library. Once the hardware has been initialised the Kernel transfers control to this program. It is responsible for maintaining the Helios world, for example keeping an accurate clock, storing the name table and performing distributed searches, and similar administrative chores. In addition it provides a **/tasks** server to allow programs to be run on that processor. When the Processor Manager has done its initialisation it will examine the rest of the Nucleus and start up any other programs, one by one.

7. The Loader is another program responsible for maintaining the code loaded into that processor. To achieve this it provides a **/loader** server. Note that the Loader does not perform all of the jobs of starting up a program, it is responsible only for fetching code and required Resident libraries off disc. The Processor Manager does the rest. When the Loader is started up by the Processor Manager it examines the Nucleus and creates **/loader** directory entries for all programs, Resident libraries, and other objects.

The above components are fixed and must be present in every Nucleus. However it is possible to build a Nucleus with more programs or Resident libraries embedded. For example, it is possible to build a Nucleus with the Posix, C, and Floating Point libraries embedded to reduce file I/O. Helios comes with two programs for maintaining nuclei: **sysbuild** and **sysbreak**.

To build a Nucleus it is necessary to obtain separate copies of the Kernel, system library, and so on. These are not shipped with Helios as separate files. Instead there is a **sysbreak** program which takes an existing Nucleus and extracts the required files. The following command line can be used:

```
sysbreak /helios/lib/nucleus kernel.i syslib.i servlib.i util.i \
         nboot.i procman.i loader.i
```

The first argument is a file containing an existing Nucleus. The remaining arguments specify the various output files, with a **.i** suffix indicating that the object can be embedded in the Nucleus. The **sysbuild** command is similar.

```
sysbuild nucleus.C kernel.i syslib.i servlib.i util.i nboot.i \
         procman.i loader.i /helios/lib/FpLib.t8 \
         /helios/lib/Posix /helios/lib/Clib
```

The first argument now specifies the output file instead of the source file, and the remaining arguments specify the Nucleus components. Alternatively, the following command line builds a Nucleus with the Helios file server embedded, which could be used in a standalone Helios system.

```
sysbuild nucleus.fs kernel.i syslib.i servlib.i util.i nboot.i \
         procman.i loader.i /helios/lib/scsi.d \
         /helios/etc/devinfo /helios/lib/fs
```

Only four types of binary objects may be embedded in a Nucleus: programs, Resident libraries, device drivers, and the **devinfo** file. Programs embedded in the Nucleus need a special startup. The Processor Manager will call them without an environment, so they must be linked with **s0.o** and not **c0.o**. Furthermore they must send a message to the Processor Manager once any necessary initialisation has been done, before the Processor Manager will continue running other programs in the Nucleus or the **init** program. For example a file server should normally delay this message until the file system is ready, to stop the Processor Manager loading **init** off hard disc too early. The following code fragment shows what is needed.

```
#include <task.h>
#include <message.h>

int main(void)
{
      /* do any essential initialisation work */

      /* let the Processor Manager continue */
   { MCB        m;
     InitMCB(&m, 0, MyTask->Parent, NullPort, 0x456);
     (void) PutMsg(&m);
   }
      /* do whatever the program is supposed to do */
}
```

16.6 Program representation

This section describes how the text (code) of a program is stored. It provides a formal summary of the the information provided earlier in this chapter.

A program in Helios is composed of a number of **modules**. Each module is composed of a body of text, plus a static data area. The text is position independent and not self modifying, so it may be shared between several programs consecutively and

concurrently. The module's static data area is referenced by a pointer in a per-task **module table**. Each module occupies a single, unique, slot in the module table. All intermodule references are indirected through the module's static data area which contains pointers to the module's external procedures as well as data. This allows the interfaces to be fixed while allowing the implementations of the modules to be varied.

The modules which comprise a program are of two basic types, those which are private to the program, and shared library modules. On disc and in memory the private modules of a program are simply concatenated together. Shared library modules are represented in this concatenation only by a reference to the name of the module and are only bound to the program at load time.

An important feature of the Helios program representation is that all information regarding a program is embedded into its text as data. Nothing is lost by loading a program into memory and may be read out again as if it were in a file. This is significantly different from other program formats (for example, **a.out**) which will strip off relocation and symbol table information as a program is loaded.

Continuing the theme of making all code self-describing, the static data area is initialised by special **initialisation procedures** added to the code by the compiler. This allows the data initialisation to be as complex or as simple as the language requires. It avoids the problem of the load format not supporting some special initialisation operation. It also greatly simplifies the loading process since the initialisation is all in executable form and does not need to be interpreted by the Loader.

16.6.1 Type codes

All embedded data items are tagged by an unique type code. The values of these are chosen to match illegal or unimplemented instructions, or instruction sequences, on the selected processor. The type codes currently supported are:

Module A module header structure.

Program A Program header structure.

ResRef A reference to a shared Resident library.

Proc A procedure header.

ProcInfo Extra procedure information for debugging purposes.

Device A Helios device driver.

FileName An embedded source file name.

DevInfo A **devinfo** file.

16.6.2 Modules

A module consists of a header structure followed by the text of the module itself. The header structure consists of the following fields:

Type **Module** type code.

Size Size of module in bytes including the header.

Name The module's name. Compilers should set this to the source file name.

Id The module table slot number of this module.

Version The version number of this module.

MaxData The size of this module's static data requirements.

Init The offset, from this word, of the first initialisation routine.

16.6.3 Resident library references

A shared Resident library reference is a module which contains no text, just the name of a library. It is essentially a place-holder for the actual module which is elsewhere in memory. A **ResRef** structure consists of the following fields:

Type **ResRef** type code.

Size Size of **ResRef** structure in bytes.

Name The name of the Resident library being referenced.

Id The module table slot number of this module.

Version The version number of the module required.

Module Pointer to actual module installed by Loader.

The combination of the **Name**, **Id** and **Version** uniquely identify the module required. Hence modules in different slots may have the same name, and different libraries may occupy the same slot(but not in the same program). The version number will match if the version on the real module is greater than or equal to the version number in the reference. [2]

The module table slot number of the real module must match that of the reference exactly. This is because intermodule references in the rest of the program will have been linked assuming the slot supplied in the reference.

It should be noted that the **ResRef** structure does not define the static data area size of the module, this is supplied from the real module. This allows different implementations of a module, using different quantities of static data, to be used by a program without recompilation.

16.6.4 Programs

In Helios a program is a simple concatenation of modules and Resident library references in module table slot order.

The first module is special and has an extended header. This is a **Program** structure:

[2] At present Helios only matches modules by name, the slot and version numbers are ignored. This will be changed in future systems.

Module A standard **Module** header, but with type **Program**.

Stacksize Size of initial thread stack.

Heapsize Size of initial program heap.

Main Offset, from this word, of the entry point to the program.

Normally this first module is programming language dependent and is responsible for causing the runtime system to be initialised and for the program proper to be entered. The Kernel function **TaskInit** starts the program running at the instruction referenced by **Main**.

At the end of the concatenation, a single word containing zero marks the end of the program. This occupies the place where the **Type** field of a following module would appear. Hence a program's modules can be scanned easily by adding a modules size to its header address until a module with type code zero is encountered.

When stored in a file, an additional **ImageHdr** header is prepended to the program:

Magic A 'magic number' indicating what type of program image this is. Current numbers are as follows:

- **Image** A normal Helios program image.
- **TaskForce** A compiled CDL object file.
- **RmLib** A Network Object.

Flags Unused.

Size Program size in bytes, including this header.

This header is primarily present to supply the program image size. This allows a program to be embedded in a file containing other data.

16.6.5 Resident libraries

A shared Resident library is a single module which implements a fixed, well known, interface. In memory it consists of the implementing module, followed by a number of **ResRef**s for the libraries that it invokes, terminated by a zero word.

In a file a library has the same **ImageHdr** as a program.

16.6.6 Embedded information

In addition to the data structures defined so far for describing the structure of a program, the text may also contain a number of embedded information structures.

The most common information structure is the **Proc** structure which describes a procedure:

Type **Proc** type code.

Proc Offset from this word to the start of the procedure.

Name Procedure name.

This normally appears immediately before the procedure it describes. Hence, by searching backwards from the procedure entry point (or a return address to it) for the **Proc** type code, the procedure's name can be found.

When a program has been compiled for debugging, the compiler can sometimes also generate a **ProcInfo** structure just before the **Proc** structure:

Type **ProcInfo** type code.

Size Size of the procedure in bytes.

StackUse Calling stack usage in words.

VstackUse Vector stack usage in words.

Modnum Module table slot number for this module.

Offset Offset of this procedure's pointer in module data area.

It is also possible for the source file name of a module to be inserted using a **FileName** structure:

Type **FileName** type.

Modnum Module number of this file.

Name Source file name.

16.7 Nucleus structure

The Helios Nucleus contains those Resident libraries and programs which are common to all processors, or which must be loaded before a file system is available (for example, the file server itself). Each of these is compiled, linked and written out to a file in standard program or library format. To build the Nucleus image file, the necessary files are simply concatenated together and preceded by a header.

The Nucleus header is an array of self-relative offsets (RPTRs) to the various components. In the standard Helios Nucleus these are:

0 The total size of the Nucleus, this is also an RPTR to the first word after the Nucleus when in memory.

1 RPTR to the Kernel.

2 RPTR to the System Library.

3 RPTR to the Server Library.

4 RPTR to the Utility Library.

5 RPTR to a copy of the bootstrap (nboot.i).

6 RPTR to the Processor Manager.

7 RPTR to the Loader.

An important point to note is that each of the libraries is in the slot which corresponds exactly to its module table slot number. The remaining programs in the array should only use libraries which are included in the system image. Unless told otherwise by the configuration, the Kernel will always start the program in slot 6, this is then responsible for starting all other programs/servers.

Like programs, the Nucleus loses no information on being loaded into a processor, and is not self-modifying. Hence it may be used from memory, to boot further processors.

Appendix A

Options: debugging and configuration file

The debugging options: summary

`A` **All**. Either enable all debugging, or disable any debugging which is currently active.

`B` **Boot**. Give a progress report while the root processor is being booted.

`C` **Communications**. Monitor transmissions to and from serial lines and similar devices.

`D` **Delete**. List all files and directories being deleted.

`E` **Errors**. Report any error messages generated by the I/O server.

`F` **File I/O**. Give details of miscellaneous file I/O activities such as renaming files.

`G` **Graphics**. Report any graphics transactions.

`H` **Raw disc**. List sector reads and writes on a raw disc device.

`I` **Initialisation**. Give a progress report as the I/O server initialises its various component servers.

`J` **Directory**. Show details of any directory accesses.

`K` **Keyboard**. Report any key presses.

`L` **Logger**. Cycle the error logging destination between screen-only, file-only, and both screen and file.

`M` **Message**. Report all messages sent to and from the I/O server.

`N` **Names**. Show the names of objects Helios is trying to access.

`O` **Open**. List all files that Helios is trying to open.

`P` **Close**. Report any file close requests sent by Helios.

`Q` **Quit**. Give a progress report when the I/O server tries to exit.

`R` **Read**. Monitor any file reads.

`S` **Search**. Report all distributed searches arriving at the I/O server.

`T` **Timeouts**. Report any stream timeouts that may occur.

`U` **Nopop**. In the *Server windows* system, toggle between pop and nopop mode.

V **OpenReply**. Give details of replies to Open, Create, and Locate requests.

W **Write**. Monitor any file writes.

X **Resources**. Produce a snap shot of what the I/O server is currently doing.

Y **List**. Give details of all debugging options.

Z **Reconfigure**. Re-read the configuration file **host.con** .

Configuration file options

The main options used in the configuration file are given below.

bootfile = SYSTEM LEVEL.

This defines the initial bootstrap code file.

bootlink

This defines the link used to boot the root Transputer. (Default = **0**.)

box =

This defines the type of extension card or external rack connected to the I/O processor. It must be one of: **NTP1000**, **MCP1000**, **B011**, **B014**, **IMB**, **VOLVOX**, **remote**.

com*n***_base**

This specifies the base address of the given extra PC serial port.

connection_retries = SUN ONLY:

This gives number of connection attempts to the link daemon.

default_centronics = PC ONLY:

This defines default centronics port.

default_printer = PC ONLY:

This defines default printer port.

default_rs232 = PC ONLY:

This defines default rs232 port.

enable_link

This is used when the I/O server connects directly into the network, rather than via a root
Transputer.

family_name = SUN ONLY:

This defines link daemon socket. Must be one of **AF_INET** or **AF_UNIX**.

floppies = PC ONLY:

This defines the PC floppy drives available.

helios_directory =

This gives the location of the **helios** file server.

hydra_host = SUN ONLY:

This defines the Internet name of the host machine.

host =

This identifies the host computer or I/O processor.

io_processor =

This changes the name of the I/O processor (default = **/IO**).

logfile =

This defines the file used as the destination of log messages. (Default is **logfile** in the current directory.)

logging_destination =

This sets the default destination of log messages to be one of **file**, **screen** or **both**. (Default is **screen**.)

message_limit =

This gives the size of the I/O server message buffer. (Default is 2000.)

mouse_divisor = PC ONLY:

This is the amount of mouse movement required.

mouse_resolution = PC ONLY:

This is the amount of mouse movement before message.

no_bootstrap SYSTEM LEVEL.

This disables the bootstrap of the root processor with the file **nboot.i**.

no_check_processor SYSTEM LEVEL.

This disables the check that the root processor is a Transputer in the bootstrap.

no_config SYSTEM LEVEL.

This disables the sending of the configuration vector in the bootstrap.

no_image SYSTEM LEVEL.

This disables request for accepting the system image in the bootstrap.

no_reset_target SYSTEM LEVEL.

This disables the reset of the root Transputer in the bootstrap.

no_*server*

Disables the given server. For example:

no_clock disables the clock server.

no_helios disables the file server.

no_logger disables the error logger.

no_rawdisk disables the disk server.

no_window disables the window server.

no_sync SYSTEM LEVEL.

This disables the sync messages within the bootstrap.

processor_memory =

This specifies the amount of memory on the root processor.

rawdisk_drive = PC ONLY:

This defines the discs converted to rawdisks.

root_processor =

This changes the name of the root processor (default = **/00**).

rs232_ports = PC ONLY:

This specifies the numbers of PC rs232 ports.

rs232_interrupt = PC ONLY:

This specifies interrupt vector of PC rs232 ports.

Server_windows

Tells the I/O server to provide a window server instead of a simple **/console** device.

site = SUN ONLY:

Remote processor sites accessible via the I/O server.

Server_windows_nopop

This disables the automatic popping to the front of the error and debug window.

socket_name SUN ONLY:

This gives the Unix socket name for accessing the link daemon.

system_image = SYSTEM LEVEL.

This defines the nucleus used to boot the system.

target_processor =

This defines the type of the boot processor. It must be one of: **Arm**, **T414**, **T800**, **T425**, **T400**, **i860**, **68000**, **T9000**.

Xsupport PC ONLY:

This enables X windows servers.

Appendix B

Options: debugging and configuration file

The debugging options: summary

`A` **All**. Either enable all debugging, or disable any debugging which is currently active.

`B` **Boot**. Give a progress report while the root processor is being booted.

`C` **Communications**. Monitor transmissions to and from serial lines and similar devices.

`D` **Delete**. List all files and directories being deleted.

`E` **Errors**. Report any error messages generated by the I/O server.

`F` **File I/O**. Give details of miscellaneous file I/O activities such as renaming files.

`G` **Graphics**. Report any graphics transactions.

`H` **Raw disc**. List sector reads and writes on a raw disc device.

`I` **Initialisation**. Give a progress report as the I/O server initialises its various component servers.

`J` **Directory**. Show details of any directory accesses.

`K` **Keyboard**. Report any key presses.

`L` **Logger**. Cycle the error logging destination between screen-only, file-only, and both screen and file.

`M` **Message**. Report all messages sent to and from the I/O server.

`N` **Names**. Show the names of objects Helios is trying to access.

`O` **Open**. List all files that Helios is trying to open.

`P` **Close**. Report any file close requests sent by Helios.

`Q` **Quit**. Give a progress report when the I/O server tries to exit.

`R` **Read**. Monitor any file reads.

`S` **Search**. Report all distributed searches arriving at the I/O server.

`T` **Timeouts**. Report any stream timeouts that may occur.

`U` **Nopop**. In the *Server windows* system, toggle between pop and nopop mode.

V **OpenReply**. Give details of replies to Open, Create, and Locate requests.

W **Write**. Monitor any file writes.

X **Resources**. Produce a snap shot of what the I/O server is currently doing.

Y **List**. Give details of all debugging options.

Z **Reconfigure**. Re-read the configuration file **host.con** .

Appendix C

Allocation of streams

The way in which streams are allocated for components is summarised by the following rules.

- A parallel constructor ^ defines no communication between its operands.
- A pipe constructor | defines a single communication between its operands. For the task force `A | B`, file descriptor 1 of A is connected to file descriptor 0 of B. Note that A and B may themselves be task forces. File descriptor 0 of `(A | B)` is equivalent to file descriptor 0 of component A, and file descriptor 1 of `(A | B)` is equivalent to file descriptor 1 of component B.
- A subordinate constructor <> defines a pair of communications between its operands. For the task force `A <> B`, file descriptor 5 of A is connected to file descriptor 0 of B, and file descriptor 4 of A is connected to file descriptor 1 of B. Again, A and B may themselves be task forces. The file descriptors 0 and 1 for the task force `(A <> B)` correspond to file descriptors 0 and 1 of component A.
- The interleave constructor ||| can be treated as a special case of the subordinate constructor. Thus `A ||| B` is equivalent to `A <> lb <> B`.
- The order of allocation of streams is defined by the precedence of the constructors. Thus for the task force `A | B <> C`, the subordinate constructor has a higher precedence so the streams for `B <> C` are allocated, and then the streams for `A | (B <> C)`.
- The allocation of streams within an auxiliary list is a special case. Consider the task force
  `A (<> B, | C)`. Here the constructors <> and | share the same left-hand operand A. The stream allocation for A starts at the first free file descriptor starting with number 4. The stream allocation for B and C is the same as for the simple cases above. Thus for the task force `A (<> B, | C) <> D`, the main task force is `(A <> D)` and the stream allocation for this is resolved, using up file descriptors 4 and 5 of component A and file descriptors 0 and 1 of component D. Then the auxiliary list components are dealt with, starting from the left: `<> B` needs two file descriptors for A, and the first two unused

file descriptors starting from 4 are 6 and 7; also, file descriptors 0 and 1 of component B are allocated; `| C` needs an output file descriptor for A, and the first unused odd file descriptor is 9; hence file descriptor 9 of A communicates with file descriptor 0 of C.

- Allocation in an auxiliary list always occurs from left to right, ignoring the precedence of the operators in the auxiliary list. Otherwise the allocation could become much more confusing. If in doubt, the `-i` or `-l` option of the CDL compiler can be used to produce a listing of the streams for all the components.

Appendix D

Measuring performance

The listings of the test programs used to investigate communications performance are given below. Two program listings are given, both of which measure the rate of data transmission between a sender and a receiver process. The first program is used to evaluate the efficiency of the Helios communications mechanisms. The level of communication is defined at compile-time to be one of :

1. Message passing primitives (**GetMsg()** and **PutMsg()**).
2. System library routines (**Read()** and **Write()**).
3. Posix library routines (**read()** and **write()**).

The second program features direct link utilisation (message transmission is implemented via in-line calls to assembler macros).

Both program listings are preceded by a CDL script that would be typically used to launch the processes on explicitly named processors. The network topology should be a simple pipeline. By modifying the **puid** fields in the CDL scripts, the required distance between processors and hence the number of intermediate links, can be specified.

Program 1: Helios communication

```
#! /helios/bin/cdl
# Communication performance measuring cdl script
component sender
{
  code /Cluster/perf/comms;
  puid /Cluster/06;
}
component receiver
{
  code /Cluster/perf/comms;
  puid /Cluster/07;
}
sender <> receiver 0
```

```
/*
- - Communications Performance
- - comms [<any.param>]
- - <any.param> : indicates whether node is the sender or receiver
- -               - if specified, node is the receiver
- -
- - Compile-time define (-D) options :
- - MSG        : Socket based message passing (GetMsg / PutMsg)
- - SYSTEM_LIB : System library routines      (Read   / Write)
- - POSIX_LIB  : Posix library routines       (read   / write)
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <nonansi.h>
#include <syslib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/hel.h>

#define MAX_BUF        65536
#define DATA_SIZES     17
#define ITERATIONS     1000

#if defined (MSG)
#define SOCKET_NAME    "socket"
void socket_error (char *err_msg)
{
  perror (err_msg) ;
  exit (1) ;
}
#elif defined (SYSTEM_LIB)
#define from_receiver  fdstream (4)
#define to_receiver    fdstream (5)
#define from_sender    fdstream (0)
#define to_sender      fdstream (1)
#endif


int main (int argc,  char **argv)
{
  char *buf ;

#if defined (MSG)
  int sock, msg_sock ;
  struct sockaddr_hel socket_addr ;
  int addr_len = sizeof (socket_addr) ;
  Port tx, rx ;
  MCB txmcb, rxmcb ;
```

```
  /* create socket */
  if ((sock = socket (AF_HELIOS, SOCK_RAW, 0)) < 0)
    socket_error ("socket ()") ;

  socket_addr.sh_family = AF_HELIOS ;
  strcpy (socket_addr.sh_path, SOCKET_NAME) ;
#endif

  if (argc > 2)
  {
    fprintf (stderr,"Usage: comms [<any.param>]\n") ;
    exit(1) ;
  }

  unless (buf = (char *) Malloc (MAX_BUF))
  {
    fprintf (stderr, "Insufficient memory to create buffer\n") ;
    exit (1) ;
  }

  if (argc < 2)
  /* -- sender ------------------------------------------------------------ */
  {
#if defined (MSG)
    Delay (OneSec) ;
    /* initiate connection */
    {
      int connect_status = -1 ;
      while ((connect_status =
      connect (sock, (struct sockaddr*) &socket_addr, addr_len)) < 0)
        fprintf (stderr, "Trying to connect ...\n") ;
    }

    tx = fdstream (sock)->Server ;
    rx = fdstream (sock)->Reply ;

    InitMCB (&txmcb, MsgHdr_Flags_preserve, tx, NullPort, 0) ;
    InitMCB (&rxmcb, MsgHdr_Flags_preserve, rx, NullPort, 0) ;

    txmcb.Data =
    rxmcb.Data = buf ;

    rxmcb.MsgHdr.DataSize = 1 ;
    /* synchronise with receiver */
    GetMsg (&rxmcb) ;
#elif defined (SYSTEM_LIB)
    /* synchronise with receiver */
    Read (from_receiver, buf, 1, -1) ;
    SetFileSize (from_receiver, 0) ;
#elif defined (POSIX_LIB)
    /* synchronise with receiver */
    read (4, buf, 1) ;
```

```
    SetFileSize (fdstream (4), 0) ;
#endif

#if defined (MSG)
    printf ("Message Passing (GetMsg, PutMsg)\n") ;
#elif defined (SYSTEM_LIB)
    printf ("System Lib (Read, Write)\n") ;
#elif defined (POSIX_LIB)
    printf ("Posix Lib (read, write)\n") ;
#endif

    printf ("%s %18s %18s %18s\n", "Msg Size (bytes)",
            "Time (usecs)", "Bytes/sec", "Kbytes/sec") ;
    {
      register int i, buf_size ;

for (i = 0, buf_size = 1 ; i < DATA_SIZES ; buf_size = 1 << ++ i)
      {
        register int j ;
        int start ;
        float time, rate ;

        if (i == 16) buf_size - - ; /* 16-bit data size */
#if defined (MSG)
        rxmcb.MsgHdr.DataSize =
        txmcb.MsgHdr.DataSize = buf_size ;
#endif
        start = _cputime () ;
        for (j = 0 ; j < ITERATIONS ; j ++)
        {
#if defined (MSG)
          PutMsg (&txmcb) ;
          GetMsg (&rxmcb) ;
#elif defined (SYSTEM_LIB)
          Write (to_receiver, buf, buf_size, -1) ;
          Read (from_receiver, buf, buf_size, -1) ;
#elif defined (POSIX_LIB)
          write (5, buf, buf_size) ;
          read (4, buf, buf_size) ;
#endif
        }
        time = (_cputime () - start) * 10000 ; /* usecs */
        time /= (2 * ITERATIONS) ;
        rate = (buf_size / time) * OneSec ; /* bytes/sec */
        printf ("%15d %18d %19d %18d\n", buf_size, (int) time,
                (int) rate, (int) (rate / 1024)) ;
      }
    }
#if defined (MSG)
    close (sock) ;
#endif
  }
  else
```

```
  /* -- receiver ------------------------------------------------------- */
  {
#if defined (MSG)
    /* name socket */
    if (bind (sock , (struct sockaddr *) &socket_addr, addr_len) < 0)
      socket_error ("bind ()") ;

    /* listen for connection */
    if (listen (sock, 1) < 0) socket_error ("listen ()") ;

    /* accept connection */
    if ((msg_sock = accept (sock, (struct sockaddr*) &socket_addr,
                            &addr_len)) < 0)
      socket_error ("accept ()") ;

    tx = fdstream (msg_sock)->Server ;
    rx = fdstream (msg_sock)->Reply ;

    InitMCB (&txmcb, MsgHdr_Flags_preserve, tx, NullPort, 0) ;
    InitMCB (&rxmcb, MsgHdr_Flags_preserve, rx, NullPort, 0) ;

    txmcb.Data =
    rxmcb.Data = buf ;

    txmcb.MsgHdr.DataSize = 1 ;
    /* synchronise with sender */
    PutMsg (&txmcb) ;
#elif defined (SYSTEM_LIB)
    /* synchronise with sender */
    Write (to_sender, buf, 1, -1) ;
    SetFileSize (from_sender, 0) ;
#elif defined (POSIX_LIB)
    /* synchronise with sender */
    write (1, buf, 1) ;
    SetFileSize (fdstream (0), 0) ;
#endif

    {
      register int i, buf_size ;

      for (i = 0, buf_size = 1 ; i < DATA_SIZES ; buf_size = 1 << ++ i)
      {
        register int j ;

        if (i == 16) buf_size - - ; /* 16-bit data size */
#if defined (MSG)
        rxmcb.MsgHdr.DataSize =
        txmcb.MsgHdr.DataSize = buf_size ;
#endif
        for (j = 0 ; j < ITERATIONS ; j ++)
        {
#if defined (MSG)
          GetMsg (&rxmcb) ;
```

```
          PutMsg (&txmcb) ;
#elif defined (SYSTEM_LIB)
          Read (from_sender, buf, buf_size, -1) ;
          Write (to_sender, buf, buf_size, -1) ;
#elif defined (POSIX_LIB)
          read (0, buf, buf_size) ;
          write (1, buf, buf_size) ;
#endif
        }
      }
    }
#if defined (MSG)
    close (msg_sock) ;
#endif
  }
  exit (0) ;
}
```

Program 2: In-line assembler macros

```
#! /helios/bin/cdl
# Communication performance measuring cdl script
component p05
{
  code /Cluster/perf/directIO;
  puid /Cluster/05;
}
component p06
{
  code /Cluster/perf/directIO;
  puid /Cluster/06;
}
component p07
{
  code /Cluster/perf/directIO;
  puid /Cluster/07;
}
#       prev    next
# p05   2       1
# p06   3       2
# p07   0       3
# directIO <prev> <next> <position>
# position : 0 = start, 1 = intermediate, 2 = end

p07 0 3 2 ^^ p06 3 2 1 ^^ p05 2 1 0
```

```
/*
- - Direct Link Usage
```

```
- - directIO <prev> <next> <position>
- - <prev>     : link to prev node (0..3)
- - <next>     : link to next node (0..3)
- - <position> : position in pipe (first (0), intermediate (1), last (2))
- -
- -    FIRST_NODE            INTER_NODE             LAST_NODE
- -        __|__                 __|__                 __|__
- -    __|     |_____ ... _____|     |_____ ... _____|     |__
- -      |_____|next       prev|_____|next       prev|_____|
- -         |                    |                    |
- -
- - Note : message pipelining not utilised
*/

#include <stdio.h>
#include <asm.h>
#include <chanio.h>
#include <stdlib.h>
#include <syslib.h>
#include <nonansi.h>
#include <link.h>
#include <config.h>

#define MAX_BUF          65536
#define DATA_SIZES       17
#define ITERATIONS       1000

#define FIRST_NODE       0
#define INTER_NODE       1
#define LAST_NODE        2

#define DUMB             0
#define INTELLIGENT      1

#define ALLOC            0
#define FREE             1

#define invalid_link(c)  (c < 0) || (c > 3)
#define invalid_pos(p)   (p < FIRST_NODE) || (p > LAST_NODE)

void Set_Link_Mode (int, int) ;

void Set_Link_Usage (int, int) ;

int main (int argc, char **argv)
{
  int link_next, link_prev, node_position ;
  char *buf ;

  if (argc != 4)
  {
    fprintf (stderr, "Usage: directIO <prev> <next> <position>\n") ;
    exit (1) ;
```

```
  }

  unless (buf = (char *) Malloc (MAX_BUF))
  {
    fprintf (stderr, "Insufficient memory to create buffer\n") ;
    exit (1) ;
  }

  link_prev     = atoi (argv [1]) ;
  link_next     = atoi (argv [2]) ;
  node_position = atoi (argv [3]) ;

  if ((invalid_link (link_next)) || (invalid_link (link_prev)))
  {
    fprintf (stderr, "Invalid link specification\n") ;
    exit (1) ;
  }

  if (invalid_pos (node_position))
  {
    fprintf (stderr, "Invalid node position\n") ;
    exit (1) ;
  }

  /* reconfigure links to Dumb mode and allocate for direct use */
  if (node_position != FIRST_NODE)
  {
    Set_Link_Mode (link_prev, DUMB) ;
    Set_Link_Usage (link_prev, ALLOC) ;
  }
  if (node_position != LAST_NODE)
  {
    Set_Link_Mode (link_next, DUMB) ;
    Set_Link_Usage (link_next, ALLOC) ;
  }

  switch (node_position)
  {
    /* -- first ------------------------------------------------------------ */
    case FIRST_NODE :
    {
      register int i, buf_size ;

      printf ("Direct Link I/O\n") ;
      printf ("%s %18s %18s %18s\n", "Msg Size (bytes)",
              "Time (usecs)", "Bytes/sec", "Kbytes/sec") ;

      buf_size = 1 ;
      /* synchronise with LAST_NODE */
      link_in_data  (link_next, buf, buf_size) ;

for (i = 0, buf_size = 1 ; i < DATA_SIZES ; buf_size = 1 << ++ i)
      {
```

```
        register int j ;
        int start ;
        float time, rate ;

        if (i == 16) buf_size - - ;
        start = _cputime () ;

        for (j = 0 ; j < ITERATIONS ; j ++)
        {
          /* send to & receive from next node */
          link_out_data (link_next, buf, buf_size) ;
          link_in_data  (link_next, buf, buf_size) ;
        }
        time = (_cputime () - start) * 10000 ; /* usecs */
        time /= (2 * ITERATIONS) ;
        rate = (buf_size / time) * OneSec ; /* bytes/sec */
        printf ("%15d %18d %19d %18d\n",
                buf_size, (int) time,
                (int) rate, (int) (rate / 1024)) ;
      }
    }
    break ;
    /* -- intermediate ------------------------------------------------- */
    case INTER_NODE :
    {
      register int i, buf_size ;

      buf_size = 1 ;
      /* synchronise FIRST_ & LAST_NODE */
      link_in_data  (link_next, buf, buf_size) ;
      link_out_data (link_prev, buf, buf_size) ;

for (i = 0, buf_size = 1 ; i < DATA_SIZES ; buf_size = 1 << ++ i)
      {
        register int j ;

        if (i == 16) buf_size - - ;

        for (j = 0 ; j < ITERATIONS ; j ++)
        {
          /* receive from previous and send to next */
          link_in_data  (link_prev, buf, buf_size) ;
          link_out_data (link_next, buf, buf_size) ;
          /* receive from next and send to previous */
          link_in_data  (link_next, buf, buf_size) ;
          link_out_data (link_prev, buf, buf_size) ;
        }
      }
    }
    break ;
    /* -- last --------------------------------------------------------- */
    case LAST_NODE :
    {
```

```
      register int i, buf_size ;

      buf_size = 1 ;
      /* synchronise with FIRST_NODE */
      link_out_data (link_prev, buf, buf_size) ;

 for (i = 0, buf_size = 1 ; i < DATA_SIZES ; buf_size = 1 << ++ i)
      {
        register int j ;

        if (i == 16) buf_size - - ;

        for (j = 0 ; j < ITERATIONS ; j ++)
        {
          /* receive from and send to previous node */
          link_in_data  (link_prev, buf, buf_size) ;
          link_out_data (link_prev, buf, buf_size) ;
        }
      }
    }
    break ;
  }

  /* free links and reconfigure to Intelligent mode */
  if (node_position != FIRST_NODE)
  {
    Set_Link_Usage (link_prev, FREE) ;
    Set_Link_Mode (link_prev, INTELLIGENT) ;
  }
  if (node_position != LAST_NODE)
  {
    Set_Link_Usage (link_next, FREE) ;
    Set_Link_Mode (link_next, INTELLIGENT) ;
  }
  exit (0) ;
}

/* -- set link mode and usage ------------------------------------------- */

void Set_Link_Mode (int link_id, int mode)
{
  struct LinkConf link_conf ;

  link_conf.Id = link_id ;

  if (mode == DUMB)
  {
    link_conf.Mode = Link_Mode_Dumb ;
    link_conf.State = Link_State_Dumb ;
  }
  else /* INTELLIGENT */
  {
    link_conf.Mode = Link_Mode_Intelligent ;
```

```
    link_conf.State = Link_State_Running ;
  }

  if (Configure (link_conf) != 0)
  {
    fprintf (stderr, "Could not configure link %d\n", link_id) ;
    exit (1) ;
  }
}

/* ---------------------------------------------------------------------- */

void Set_Link_Usage (int link_id, int mode)
{
  int status = 0 ;

  if (mode == ALLOC)
  {
    while (status == 0)
    {
      fprintf (stderr, "Trying to allocate link %d ...\n", link_id) ;
      status = AllocLink (link_id) ;
    }
  }
  else /* FREE */
  {
    if (FreeLink (link_id) != 0)
    {
      fprintf (stderr, "Can't free link %d\n", link_id) ;
      exit (1) ;
    }
  }
}
```

Performance measurements

The rates of data transmission obtained from the test programs used to evaluate communications performance are given in the following tables. The programs were designed to measure the time taken to transmit messages between a sender and a receiver task. These tasks were placed on processors separated by varying numbers of intermediate nodes. The programs were compiled using version 2.01 of the Helios C compiler, and run under Helios version 1.2.1. The hardware environment comprised a directly linked pipeline of 20MHz IMS T800C-G20S Transputers, each having access to 1 Mbyte of 4 cycle (200 ns) external RAM. The links speeds of the processors were set at 20 Mbits/second.

The rates of transmission are expressed in Kbytes/second, and are given with respect to the message size (in bytes), and the number of intermediate links through which the messages were routed. The results are presented separately for each respective level of communication:

1. Direct link usage using assembler macros.
2. Message passing primitives (**GetMsg()** and **PutMsg()**).
3. System library routines (**Read()** and **Write()**).
4. Posix library routines (**read()** and **write()**).

Direct link usage

Size	Number of intermediate links							
(bytes)	0	1	2	3	4	5	6	7
1	-	195	97	65	48	39	32	27
2	-	195	130	130	78	65	55	48
4	-	781	260	195	130	111	86	86
8	-	781	390	312	223	156	142	130
16	-	1562	520	390	284	240	195	156
32	-	1562	694	446	328	271	223	195
64	-	1562	735	520	378	304	255	219
128	-	1666	833	531	403	324	268	229
256	-	1724	847	555	416	333	277	239
512	-	1724	854	568	423	338	283	241
1024	-	1724	858	571	427	341	285	244
2048	-	1724	862	575	429	343	286	245
4096	-	1727	863	575	430	344	287	246
8192	-	1729	864	576	430	344	288	246
16384	-	1730	865	576	430	344	287	246
32768	-	1731	865	576	430	345	287	246
65535	-	1731	865	577	430	345	287	246

Message passing primitives

Size	Number of intermediate links							
(bytes)	0	1	2	3	4	5	6	7
1	11	7	5	3	3	2	2	1
2	21	15	10	7	6	5	4	3
4	45	30	20	15	12	10	8	7
8	91	60	40	30	24	20	17	15
16	173	115	80	60	47	40	34	30
32	328	215	152	115	93	78	67	59
64	657	390	260	204	171	145	126	111
128	1250	641	384	324	274	240	215	192
256	2173	925	500	446	396	359	328	304
512	3571	1204	746	645	581	523	476	436
1024	5263	1418	1015	925	843	778	727	680
2048	6666	1562	1223	1111	1010	932	873	809
4096	7920	1646	1340	1271	1197	1136	1086	1038
8192	8648	1689	1410	1369	1313	1275	1243	1210
16384	9090	1711	1447	1425	1384	1359	1339	1319
32768	9343	1723	1466	1454	1421	1405	1393	1379
65535	9453	1729	1475	1468	1440	1428	1419	1409

System library routines

Size	Number of intermediate links							
(bytes)	0	1	2	3	4	5	6	7
1	0	0	0	0	0	0	0	0
2	1	1	1	1	1	1	1	1
4	2	2	2	2	2	2	2	2
8	4	5	5	5	4	4	4	4
16	9	11	10	10	9	9	9	9
32	19	22	21	20	19	18	18	18
64	37	44	42	40	38	37	36	35
128	75	87	82	78	74	70	68	67
256	148	170	151	140	141	132	125	117
512	288	298	265	250	257	242	229	217
1024	529	457	381	347	319	295	279	264
2048	1005	724	610	557	508	470	444	418
4096	1818	1021	866	810	756	712	679	648
8192	3059	1287	1095	1047	996	955	925	896
16384	4630	1478	1260	1228	1184	1153	1130	1108
32768	6243	1596	1364	1344	1307	1285	1270	1254
65535	7467	1657	1415	1403	1372	1354	1346	1335

Posix library routines

Size	Number of intermediate links							
(bytes)	0	1	2	3	4	5	6	7
1	0	0	0	0	0	0	0	0
2	1	1	1	1	1	1	1	1
4	2	2	2	2	2	2	2	2
8	4	5	5	5	4	4	4	4
16	9	11	10	10	9	9	9	9
32	18	21	20	20	19	18	18	18
64	37	43	41	39	38	36	35	35
128	74	86	81	77	73	69	68	66
256	145	168	149	139	140	131	123	117
512	283	294	263	248	255	240	227	214
1024	520	454	379	346	317	294	277	263
2048	987	720	607	554	507	469	442	416
4096	1793	1019	863	808	753	709	677	647
8192	3018	1284	1092	1047	993	953	924	895
16384	4597	1475	1259	1227	1184	1151	1129	1106
32768	6213	1595	1363	1343	1307	1285	1270	1254
65535	7441	1657	1415	1403	1372	1356	1345	1335