Computing Surface

CS-2 Documentation Set

Volume 1

83–MS047

meiko

Acceptance

All Meiko software and associated manuals ("the Software") is provided by the Meiko Group of Companies ("Meiko") either directly or via a Meiko distributor and is licensed by Meiko only upon the following terms and conditions which the Licensee will be deemed to have accepted by using the Software. Such terms apply in place of any inconsistent provisions contained in Meiko's standard Terms and Conditions of Sale and shall prevail over any other terms and conditions whatsoever.

Copyright

All copyright and other intellectual property rights in the software are and shall remain the property of Meiko or its Licensor absolutely and no title to the same shall pass to Licensee.

Use

Commencing upon first use of the Software and continuing until any breach of these terms, Meiko hereby grants a non-exclusive licence for Licensee to use the Software.

Copying

Copying the Software is not permitted except to the extent necessary to provide Licensee with back-up. Any copy made by Licensee must include all copyright, trade mark and proprietary information notices appearing on the copy provided by Meiko or its distributor.

Assignment

Licensee shall not transfer or assign all or any part of the licence granted herein nor shall Licensee grant any sub-licence thereunder without prior written consent of Meiko.

Rights

Meiko warrants that it has the right to grant the licence contained under "Use" above.

Warranty

Meiko warrants that its software products, when properly installed on a hardware product, will not fail to execute their programming instructions due to defects in materials and workmanship. If Meiko receives notice of such defects within ninety (90) days from the date of purchase, Meiko will replace the software. Meiko does not warrant that the operation of the software shall be uninterrupted or error free.

Unless expressly stated in writing, Meiko gives no other warranty or guarantee on products. All warranties, express or implied, whether statutory or otherwise [except the warranty hereinbefore referred to], including warranties of merchantability or fitness for a particular purpose, are hereby excluded and under no circumstances will Meiko be liable for any consequential or contingent loss or damage other than aforesaid except liability arising from the due course of law.

Notification of Changes

Meiko's policy is one of continuous product development. This manual and associated products may change without notice. The information supplied in this manual is believed to be true but no liability is assumed for its use or for the infringements of the rights of others resulting from its use. No licence or other rights are granted in respect of any rights owned by any of the organisations mentioned herein.

Nuclear and Avionic Applications

Meiko's products are not to be used in the planning, construction, maintenance, operation or use of any nuclear facility nor for the flight, navigation or communication of aircraft or ground support equipment. Meiko shall not be liable, in whole or in part, for any claims or damages arising from such use.

Termination

Upon termination of this licence for whatever reason, Licensee shall immediately return the Software and all copies in his or her possession to Meiko or its distributor.

Important Notice

FEDERAL COMMUNICATIONS COMMISSION (FCC) NOTICE

Meiko hardware products ("the Hardware") generate, use and can radiate radio frequency energy and, if not installed and used in accordance with the product manuals, may cause interference to radio communications. The Hardware has been tested and found to comply with the limits for a Class A computing device pursuant to Subpart J of Part 15 of FCC Rules which are designed to provide reasonable protection against such interference when operated in a commercial environment. Operation of the Hardware in a residential area is likely to cause interference in which case the user at his or her own expense will be required to take whatever measures may be required to correct the interference.

Computing Surface

Documentation Guide

S1002–00C101.10

meiko

Meiko's address in the US is:

Meiko
130 Baker Avenue
Concord MA01742

508 371 0088
Fax: 508 371 7516

Meiko's address in the UK is:

Meiko Limited
650 Aztec West
Bristol
BS12 4SD

Tel: 01454 616171
Fax: 01454 618188

Issue Status:

Draft	
Preliminary	
Release	x
Obsolete	

Circulation Control: *External*

Documentation Guide

The following documentation is supplied by Meiko for users of the Meiko CS-2 system. This list includes documentation that is published by Meiko and documentation that is supplied by third parties for their own products.

The following summaries are arranged so that the highest level descriptions occur at the head of the list, and the lowest level at the bottom. The intended audience for each document is shown in the margin and is either Manager, Application Programmer, System Programmer, or Programmer (either systems or applications).

This documentation is supplied on paper, as postscript source in the directory `/opt/MEIKOcs2/docs`, and in a Meiko AnswerBook. Meiko's own documentation is released in all three formats, whereas third party documentation is generally not compatible with the AnswerBook and its distribution may also be restricted by licence.

Many of the commands, library functions, and file formats described in the following manuals are also described by manual pages that are distributed in the `/opt/MEIKOcs2/man` directory; you should ensure that this directory is referred to by your MANPATH environment variable, and that the Meiko directory is listed first. Use the Solaris `man` command to view these manual pages.

Overview Documentation

Managers / Programmers

Communications Processor Overview

Overview of the Elan communications processor, listing design objectives and implementation decisions.

Managers / Programmers

Vector Processing Element Overview

Overview of the Meiko vector processing boards describing the hardware architecture, the Fujitsu μVP and SPARC processors, and compiler technology.

Managers / Programmers

Communications Network Overview

Overview of the CS-2 data network. Compares the CS-2 network with other network types (logarithmic, ring etc.), and describes the benefits of Meiko's implementation.

Software Documentation

Managers / Systems

CS–2 System Administration Guide

Describes the main software components in the CS-2 and their installation and configuration.

Managers / Programmers

Getting Started — Users Guide

A user's introductory guide to the CS-2. Describes the main features of the CS-2, the programming libraries and utilities, and shows how they are used.

Managers / Programmers

Pandora Users' Guide

Pandora is the user interface to the CS-2 resource management system. It allows users to query resource availability. Pandora allows the System Administrator to partition resources, to restrict user access to them, and also provides a diagnostic capability.

Managers / Systems

Group Routing

Describes the group routing facility. The kernel network routing tables have been extended to include user groups that are permitted to use each route. This documentation describes the implementation and usage of this facility.

Programmers	**Elan Widget Library** Library documentation describing low level functions that are used to implement higher level message passing systems, such as PVM and Meiko's CSN.
Applications	**CSN Communications Library for C** Library documentation describing the implementation of Meiko's CSN communications library in the CS-2 environment. This documentation describes the C interface to this library.
Applications	**CSN Communications Library for Fortran** Library documentation describing the implementation of Meiko's CSN communications library in the CS-2 environment. This documentation describes the Fortran interface to this library.
Applications	**Tagged Message Passing and Global Reduction** Library documentation describing the configuration and use of the CS-2 for execution of parallel applications that have been imported from machines with a hypercube topology. This library also defines global reduction operations.
Programmers	**Resource Management User Interface Library** Describes the programmers interface to the resource management system allowing user programs to query the machine configuration and to start parallel applications.
Applications	**PVM Users Guide and Reference Manual** PVM (Parallel Virtual Machine) is a software package that allows a heterogeneous network of parallel and serial computers to appear as a single concurrent computational resource. This documentation describes the CS-2 implementation of PVM.
Programmers	**The Elan Library** Describes the lowest level library interface to the Elan Communications Processor. This library offers direct access to the Elan's DMA and event functionality.

Hardware Documentation

Managers

Processor Module Users Guide

Describes the CS-2 Processor Module and the boards that can be fitted into it. Lists handling requirements, power specifications, and field serviceable components.

Managers

Switch Module Users Guide

Describes the CS-2 Switch Module and the boards that can be fitted into it. Lists handling requirements, power specifications, and field serviceable components.

Third Party Parallel Programming Tools

The following documentation describes software ports or Meiko implementations of programming systems developed or conceived by third parties. References to the originators are included in the documentation that accompanies each product.

Applications

ScaLAPACK — Optional

Defines routines for LU factorization, QR factorization, Cholesky factorization, Hessenberg reduction, tridiagonal reduction, and Bidiagonal reduction. Documentation is provided for these routines and the BLAS and BLACS libraries that form their foundation. Accompanied by Meiko release notes.

Applications

Basic Linear Algebra Subprograms — Optional

Two documents are provided describing the BLAS 2 and BLAS 3 libraries (both prepared at the Argonne National Laboratory). This documentation is supplemented by release notes and usage information that has been prepared by Meiko.

Managers / Programmers

Solaris Documentation

The Solaris operating system is fully documented on the SunSoft AnswerBook CD-ROM.

Applications

TotalView — Optional

Third party debugging software distributed under licence and with extensive documentation. Published by BBN.

Applications

Adaptor — Optional

Adaptor (automatic data parallelism translator) is a tool for transforming data parallel programs written in Fortran with array extensions, parallel loops, and layout directives into parallel programs with explicit message passing. Distributed with User's Guide and Language Reference Manual (published by GMD).

Applications

Paragraph — Optional

A graphical display system for visualising the performance of parallel programs. The documentation is published by the University of Illinois and the Oak Ridge National Laboratory.

Programmers

Portland Group Compilers — Optional

The Portland compilers generate code for the CS-2 vector and scalar processors, and are extensively documented by the Portland Group's own documentation set.

Computing Surface

Communications Processor Overview

S1002–10M100.04

meiko

Meiko's address in the US is:

Meiko
130 Baker Avenue
Concord MA01742

508 371 0088
Fax: 508 371 7516

Meiko's full address in the UK is:

Meiko Limited
650 Aztec West
Bristol
BS12 4SD

Tel: 01454 616171
Fax: 01454 618188

Issue Status:

Draft	
Preliminary	
Release	x
Obsolete	

Circulation Control: *External*

Contents

Overview 1

Effective cooperation between processing elements is a crucial factor in determining the overall performance of an MPP system. Maintaining effective interprocessor communication as a system scales in size is a vital aspect of preserving balance.

In designing the CS-2 architecture Meiko has concentrated on minimising the impact of sharing work between processors. The effect of this is to increase the number of processors that can be used effectively to solve a problem, improving the performance of existing parallel programs and making parallel processing efficient for a wider range of applications.

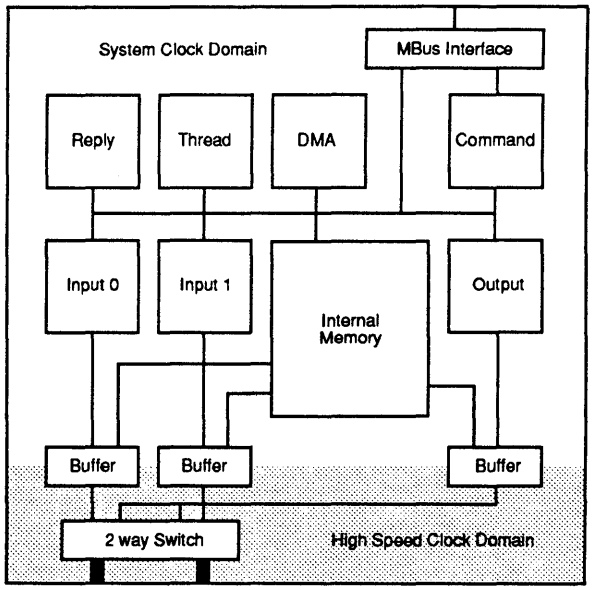
Every processing element in a CS-2 system has its own, dedicated interface to the communications network: a Meiko designed communications processor. The communications processor has a SPARC shared memory interface and two data links. Data links are connected by Meiko designed 8×8 cross-point switches. Data links are byte wide in each direction and operate at 70MHz, providing 50Mbytes/s of user bandwidth in each direction.

The communications processor supports remote read, write and synchronisation operations specified by virtual processor number and virtual address — both are checked in hardware. Latency hiding is supported by non-blocking instructions, instruction sequences and completion tests.

1

This document provides an overview of the design of the communications processor and its usage. For more information about the architecture of the data network see the *Communications Network Overview.*

Figure 1-1 The Elan Communications Processor



Inter-processor Communications 2

In a distributed memory system work is shared between processors by exchanging data over a communications network. The efficiency of data exchange controls the effectiveness of work sharing and hence the number of processors that can be used on a given problem.

Rather than design a new processor with built in communications capability Meiko chose to separate the issues in the design of the CS-2. Processing elements consist of a high performance RISC CPU (with optional vector processing capabilities) and a dedicated communications processor.

The interface between the communications processor and the rest of the processing element is central to the efficiency of the CS-2 network. It provides the following essential features:

- Low communication start-up latency.
- High bandwidth inter-processor communication.
- Security against corruption.
- Operation in a network-wide virtual addressing, virtual process environment.

Latency and Bandwidth

Efficient inter-processor communication requires both low latency and high bandwidth. While solutions to the bandwidth problem can be addressed by ever improving hardware technology, these improvements only exacerbate underlying latency problems.

To show that this is the case consider a system with a communications start-up latency of 10μs. To transfer a 100 byte message via a 1Mbyte/s network we will get an achieved bandwidth of 0.9Mbytes/s (90% efficiency). For the same transfer over a 50Mbytes/s network, the achieved bandwidth is just 8.3Mbytes/s (16% efficiency). Clearly the improvements in bandwidth for this example system have been severely limited by the start-up latency and the size of the data transfer.

By using a dedicated communications processor Meiko have reduced start-up latency by implementing in hardware the communications code that would normally execute on the main processor.

The data links joining communications processors and network switches are byte wide in each direction. Links are clocked at 70MHz. Their bandwidth after protocol is 50MBytes/s in each direction. The CS-2 data network is a fat tree with constant bandwidth between stages. It is capable of supporting full bandwidth transfers between all pairs of processors (see the *Communication Network Overview* for more details).

Moving communications code from the main processor to a communications engine does not in itself reduce latency. Performance improvements come from running the right code in the right places. In particular there are significant benefits to be had from moving the lightweight interrupt intensive operations associated with inter-process communication off a conventional microprocessor and onto a communications processor designed specifically for this purpose.

Network Security

The CS-2 communications network is shared by both user and system level communications so it is vital that a security mechanism is used to prevent unrelated communications from interacting. To relieve the burden of checking from the main processor and to reduce start up latency, the main processor is-

sues unchecked communication instructions to the communications processor, the communications processor then implements the security strategy in hardware. This mechanism is preferable to the more conventional use of kernel mapped devices, which use checked system calls to access the device, often with a significant performance impact (a checked system call in a 40MHz SPARC takes approximately 50μs). The CS-2 network protects processes from communications errors that occur within other unrelated processes, but does not protect a process from errors within itself. This is the same model as that employed for memory protection by the UNIX operating system — processes are protected from each other, but not from themselves.

Virtual Addressing

The communications processor uses separate page tables from the main processor. This means that a user process need not make its entire address space visible when it communicates, only the portion that contains the data need be mapped for communication. Secondly, separate page tables may be used to reduce the amount of cache flushing in non cache-coherent systems; in a write through cache only those pages that are mapped with write permission need be flushed.

The two sets of page tables are kept in step by a modified page out daemon and new page in code in the operating system. The modified page out daemon modifies both sets of tables, whereas the new page in code handles the asynchronous page faults from the communications processor.

2

Elan Functionality 3

The functionality of the communications processor was decided by drawing on experience from Meiko's CSTools/CSN communication software, used to create a programming environment over Transputer networks, and other message passing systems such as the Chorus Nucleus. This analysis showed that the start-up process consists of four components:

- Checking.
- Translation.
- Copying.
- Device control.

Each of which is important if start-up latency is to be minimised.

Checking

The CS-2 supports virtual memory addressing on each processing element, allowing it to implement a fully distributed store for operating system use, and permit it to implement the applications binary interface (ABI) for the base microprocessors. The communications processor therefore has two types of parameters to check: memory addresses and process addresses.

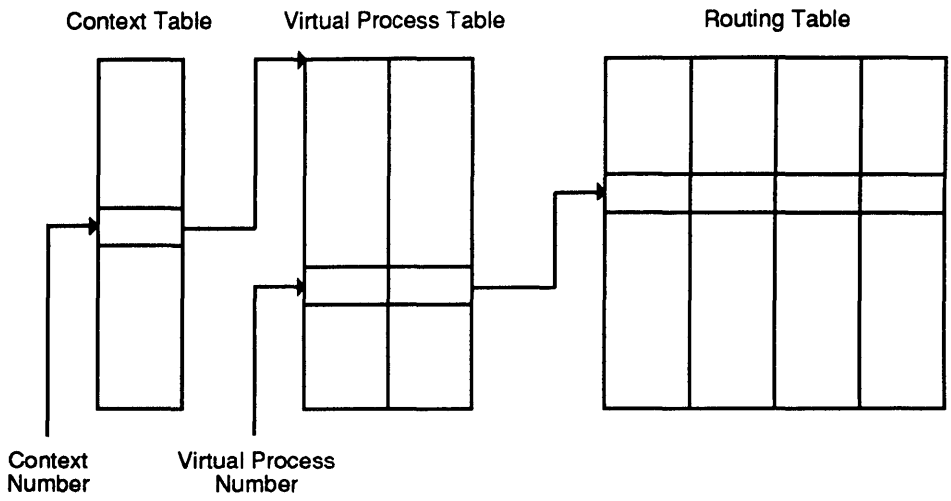
The communications processor receives unchecked virtual memory addresses from the main processor so it must incorporate a memory management unit (MMU). The MMU used within the Elan supports multiple simultaneous contexts allowing I/O to continue for suspended processes.

The checking of process addresses is analogous to the checking of memory addresses. It is implemented by a simple table look-up and exception mechanism. The communications processor is designed to handle the common case where a user is trying to communicate with other processes for which it has permission; an exception is generated whenever there is no permission. As checking is performed independently on each of the communications processors, failed processing elements can be removed from service by removing them from each communications processor's list of valid destinations.

Translation

Process and memory translation within the communications processor is implemented through the same mechanism as the checking, that is, by table look-ups. Memory address translation yields the same results as the main processor's translation mechanism. Dynamic process translation yields two components: a destination processor and a destination context. There are no physical processor or memory addresses in user space

Figure 3-1 Elan Process Translation



Virtual process IDs are translated through a per context virtual to physical processor translation which points at the route bytes needed to direct a message to this processor.

Copying

The communications processor supports a number of features to remove the requirement for copying of data. By using network wide virtual addressing there is no need to copy data into physically mapped output buffers, a common technique in distributed systems to overcome the problems of virtual address translation and page locking during communication. Furthermore, because the main processor and the communications processor share a common memory bus (a SPARC MBus) and the same cache coherency protocols, the problems associated with cache coherency are also avoided.

Clearly the avoidance of unnecessary copying contributes greatly to reduced start-up latency and efficient use of memory bandwidth. For messages that are copied once on sending, this adds $(messagesize \times 2) / (memorybandwidth)$ to the start-up latency, and consumes three times as much store bandwidth.

Device Control

The final requirement of message start-up code is in device control. This is setting up the communications parameters in store, signalling to the communication device, and responding to interrupts returned by the communications processor.

Control of the communications processor is via a command port which is normally mapped into the user address space. The command port consists of a range of memory addresses. The communications processor command is determined by extracting 5 bits from the address that is used. The data that is used by the communications processor command corresponds to the 32 bits of data that are written to that memory address. Commands sent to the command port are written in a single read-modify-write cycle and are acknowledged with the value that is read back (which will be non-negative if the command is accepted). The kernel can prevent the user issuing certain commands by mapping limited portions of the command port address space in to the user address space.

Exceptions generated by the communications processor may be handled by the communications processor's own thread processor, without direct intervention by the main processor.

Thread Processor

One of the objectives of the Elan communications processor is to reduce the number of interrupts and system calls that must be executed to perform message passing. As we have seen the combination of the user mapped command port and the Elan communication processor's security mechanisms allows user level code to initiate remote memory accesses without making a system call. In many cases, however, message protocols require higher level functions than simply the transfer of data. Other common requirements are for synchronisation between processes executing on separate processors, and allocation of global resources. To support these requirements the Elan communications processor includes a RISC processor which can execute user level code independently of the main node processor, and also create additional network transactions.

The hardware and microcode of the thread processor support an extremely lightweight scheduling mechanism. This allows lightweight processes (threads) running on the thread processor to be suspended and then rapidly rescheduled by the hardware when the relevant event has occurred.

The user level code in the main node processor can directly request the execution of a thread process through access to the appropriate command port. The thread code has no more privileges than the user code which initiated it. The Elan communications processor uses its page tables for the relevant user context whenever it makes a store access from the thread.

Thread code

Thread code can be written in ANSI C. An inlined library provides access to the Elan communication processor I/O instructions without the overhead even of a subroutine call.

Events

Events provide a general mechanism by which synchronisation may be achieved between lightweight threads running either in the same, or different, Elan communication processors. In addition an event can be used to cause an interrupt to the main node processor. An event is represented by a double word in store.

A thread can perform the following operations on either local or remote events:

Wait	If the event has already been set, then execution continues and the event is unset. Otherwise the thread is suspended on the event until the event is set, when it will be rescheduled.
Set	The event is set. If there was an action already present on the event then it is performed.
Clear	If the event was set it is cleared.
Test	Poll the status of an event without modifying or suspending on it.

There are various possible actions which can occur when an event is triggered, these depend on what has been suspended in the event structure:

A local thread	The thread is placed back on the thread run queue, so will resume execution.
A remote thread	The remote thread is rescheduled on its own processor.
A local interrupt	The main processor is interrupted.

Events also support queues of outstanding requests. When a queued event is set, the first action on the queue is executed, and the queue updated to point to the next action.

Other Forms of Remote Access

In addition to events, the Elan also supports other forms of remote store access. In particular thread code can generate network transactions to perform:

Atomic Swap	The word at the given remote address is returned, and overwritten with the word sent in the message.
Atomic Add	The word sent in the message is atomically added to the data at the remote address. The original remote data may optionally be returned.
Atomic test and store	The word at the remote address is compared with a test value sent in the message. If equal then a new value sent in the message is written to the remote store, otherwise the remote store is unchanged. The original remote value may optionally be returned.
Remote compares	The word at the remote address is compared with the given data using one of the operations ==, =, >= or <. The result of the comparison is returned as an acknowledge or negative acknowledge.

The broadcast capabilities of the Elite switch can be used to combine the results of a broadcast remote compare operation into a single result.

Using the Communications Processor 4

In this section we show in outline how the communications processor is used to communicate with other processes via the data network. The example shows how to initiate a DMA transfer to remote store.

DMA Transfers

In the previous sections we have seen that a key factor in the design of the communications processor is that it offers low communication start-up latencies, and that communication start-up requires minimal intervention by the main processor. For a typical DMA transfer of data to a remote processor, the actions required by the main processor are as follows:

- User program creates a DMA structure in store identifying the characteristics of the transfer (source and destination addresses, amount of data, etc). This could be done in advance if the same access is to be made repeatedly.
- User program issues DMA command with RmW to command port. The address of the DMA structure is written to the appropriate address in the command port.
- User program checks command accepted; a value of greater than or equal to 0 in the command port indicates that the command was accepted.

The main processor is now free to continue with its work leaving the communications processor to transfer the data, and to ensure its integrity. The actions now required by the communications processor are:

- Command processor reads the 32bit data from the command port and uses this to locate the DMA descriptor. The descriptor is read into the communications processors DMA queue.
- DMA processor reads the queue item in.
- DMA processor performs destination process translation.
- DMA processor reads route information.
- DMA processor reads source data in and starts to send. The route information is prepended to the data, and is stripped off as it passes through the switch network.

If the main processor wanted confirmation that a DMA had completed it would include a pointer to an event in the DMA description. Polling this event (when there is no more useful work to do) would confirm completion of the transfer.

Conclusions 5

Efficient inter-processor communications requires the right balance of latency and bandwidth. CS-2 uses Meiko's own communication hardware, developed from many years experience in the massively parallel processing field, to create a network with both high bandwidth and low start-up latency.

The Elan communications processor is key to minimising the network latency. It serves not just as a communications co-processor, but aims to minimise the amount of message start up code, and therefore minimise start-up latency. For simple communications the overhead on the main processor can be reduced to a single read modify write. More complex protocols require small fragments of code to be run on the communications processor. The requirement for copying of messages is removed by the ability of the communications processor to operate in virtual store. Protection is implemented by hardware table look ups of translation tables which impose low overhead on valid operations, and generate exceptions in the much less frequent error cases.

Computing Surface

Communications Network Overview

S1002–10M105.05

meiko

Meiko's address in the US is:

Meiko
130 Baker Avenue
Concord MA01742

508 371 0088
Fax: 508 371 7516

Meiko's full address in the UK is:

Meiko Limited
650 Aztec West
Bristol
BS12 4SD

Tel: 01454 616171
Fax: 01454 618188

Issue Status:

Draft	
Preliminary	
Release	x
Obsolete	

Circulation Control: *External*

Contents

General Description 1

Effective cooperation between processing elements (PEs) is a crucial factor in determining the overall sustained performance of a Massively Parallel Processing (MPP) system.

In designing the CS-2 architecture, Meiko has concentrated on minimizing the impact of sharing work between processors. The effect of this is to increase the number of processors that can be effectively used to solve a problem, improving the performance of existing parallel programs, and making parallel processing effective for a significantly wider range of applications.

Every processing element in a CS-2 system has its own, dedicated interface to the communications network: a Meiko designed communications processor. The communications processor has a SPARC shared memory interface and two data links, these links connect the communications processors to Meiko designed cross-point switches.

This document provides an overview of the design of the communications network. For more information about the architecture of the communications processor see the *Communications Processor* Overview.

Network Characteristics

The design of the CS-2 data network builds on Meiko's considerable expertise in the field of MPP systems. From the outset the communications network was designed with several key characteristics in mind:

- Full connectivity.
- Low latency.
- High Bandwidth.
- Fault tolerance.
- Deadlock freedom.
- Scalability.

Full Connectivity

Every processing element (PE) has the ability to access memory on any other PE. Messages pass from the source to destination PEs via a dynamically switched network of active switch components. The network is fully connected, allowing a machine with *n* PEs to sustain *n* simultaneous transfers between arbitrarily selected pairs of PEs at full bandwidth.

The communication network does not use the PEs as part of the network, only as gateways on to it. This ensures that node resources (such as CPU and memory bandwidth) are not affected by unrelated network traffic.

Low Latency

Inter-process communications latency has two components, start-up latency (which is covered in the *Communications Processor Overview*) and network latency. The CS-2 communication network is designed to minimize and hide network latency. Wormhole routing is used to reduce the latency through each switch stage, and the overall network topology is designed to minimize the number of stages through which a message passes. The low level communication protocols allow overlapped message acknowledgments, and the message packet size is dynamically adjusted so that it is always sufficient for full overlapping to occur.

CS-2 communications start-up latency are less than 10μs, network latencies are less than 200ηs per switch.

High Bandwidth

The communication bandwidth in an MPP system should be chosen to give an appropriate compute communications ratio for current PE technology. The network design should ensure that additional bandwidth can be added to maintain the compute/communication ratio as the performance of the PEs improves with time. Although the actual required compute/communications ratio is application specific, the higher the network bandwidth the more generally applicable the MPP system will be.

CS-2 data links are byte wide in each direction and operate at 70 MHz. Usable bandwidth (after protocol overheads) is 50 Mbytes/s/link in each direction. Bisectional bandwidth of the CS-2 network increases linearly with the number of PEs. A 1024 PE machine has a bisectional bandwidth of over 50 Gbytes/s.

Fault Tolerance

The network for a very large MPP system will of necessity consist of a very large number of components. Moreover for large systems a significant number of cables and connectors will be required. Under these circumstances reliability becomes a major issue. Tolerance to occasional failures by the provision of multiple routes through the network is desirable for small systems, and essential for very large systems.

CS-2 systems have two fully independent network layers and each PE is connected to both layers. In addition each layer provides multiple routes between each arbitrarily selected pair of PEs. The hardware link protocol uses Cyclic Redundancy Checks (CRCs) to detect errors on each link; failed transmissions are not committed to memory, but cause the data to be resent. All network errors are flagged to the System Administrator; permanently defective links can be removed from service.

Deadlock Freedom

Routing through multistage networks is essentially a dynamic resource allocation problem and, because multiple PEs are attempting to acquire sets of route hops simultaneously, there is the potential for deadlock. The most common deadlock avoidance strategy is always to allocate resources in a fixed order. With.wormhole routing, since the resources are allocated as the message wormholes through a network, this affects routing strategy for a given topolo-

gy. For example in a hypercube or a grid, deadlock free routing is possible by ensuring that a PE routes by resolving the address one dimension at a time in ascending order. Note: that this actually removes the fault tolerance of the network; between PEs that differ by more than one dimension there are many possible routes, but only one can be used without risk of deadlock.

Scalability

The requirement for scalability within a network is one of the most difficult to achieve in actual systems. The three factors that need to be considered are, growth in network latency with scaling, growth in network cost, and growth in bisectional bandwidth.

The scalability properties of various network topologies are:

Type	Number of Switches	Number of Links	Latency	Bisectional Bandwidth
Ring	N	N	$N-1$	2
d dimensional grid	N	dN	$d\sqrt[d]{N}$	$\sqrt[d]{N}$
Arity d Omega net	$N\log_d N$	$(dN\log_d N)/2$	$\log_d N$	N
Arity d benes net	$2N\log_d N$	$dN\log_d N$	$2\log_d N$	N
Crosspoint	N^2	N^2	1	N

Where N is the number of processors in the machine, *Number of Links* is the total number of connections between switches, *Latency* is the worst case number of switches which must be passed through, and *Bisectional Bandwidth* is the worst case bandwidth between two halves of the machine.

For scalability it is essential that the bisectional bandwidth of the machine increases linearly with the number of processors. This is necessary because many important problems cannot be parallelised without requiring long distance communication (for example, FFT, and matrix transposition).

The cost (both in switches and wires) of a full crosspoint switch increases as the square of the number of processors. Adoption of this network therefore leads to a machine in which switch and wire costs rapidly dominate when significant numbers of processors are used. For the logarithmic networks the switch and wire costs increase only logarithmically faster than the number of

processors. It is therefore possible to build machines which contain significantly more processors before the switch costs dominate and the machine ceases to be cost effective.

The crosspoint has the advantages of contention freedom and constant network latency for all routes. However, although the worst case latency in a logarithmic network increases slowly with the number of processors, they can be arranged so as to ensure that this increase only occurs when long distance communication is required—performance is not dependent upon exploiting locality of reference, but doing so is beneficial.

The arity of the logarithmic network is the size of the crosspoint switch from which the network is built. So if the crosspoint is built from 2×2 switches it will have arity of 2. The choice of switch arity is highly influenced by the available packaging technology, since given a limited number of pins to connect into a switch there is a reciprocal relationship between the arity of the switch and the number of wires in each link. As the bandwidth of a link is directly related to the number of wires over which it is carried, this translates into a choice between a high arity switch which can switch many low bandwidth links, or a low arity switch for few high bandwidth links.

Logarithmic Networks

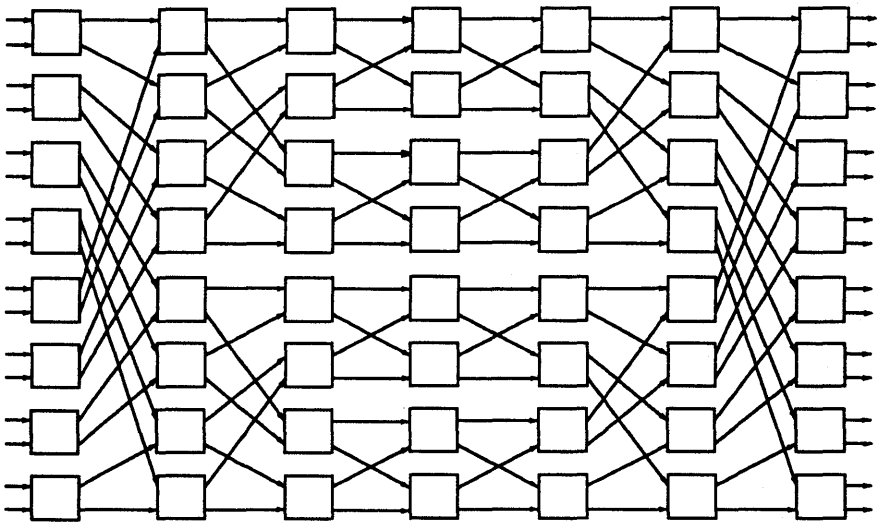
In order to analyze the CS-2 network it is useful to understand the characteristics of the Benes and Omega networks.

The main attraction of the Benes network is that it can be proved to have equivalent functionality to a full crosspoint (see Hockney and Jesshope[1] for a review)—any permutation of inputs can be connected to any permutation of outputs without contention. There are also multiple routes between any input-output pair. Calculating the routing to ensure that the routes are allocated without congestion for any given permutation is, however, a non-trivial problem.

1. R.W.Hockney & C.R.Jesshope. Parallel Computers 2. Pub. Adam Hilger.

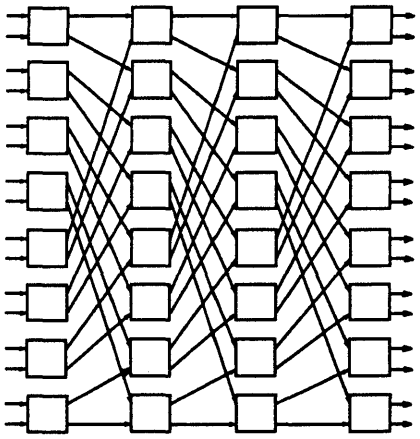
This problem has been solved for a number of interesting special cases communication patterns: rings, grids, hypercubes etc. There has also been extensive simulation of these networks under a wide variety of loadings.

Figure 1-1 16 Processor Benes Network



In an Omega network there is only one possible route for each input-output pair. Not all possible permutations are possible without blocking, although common geometric patterns such as shifts and FFT butterflies can be shown to be contention free.

Figure 1-2 16 Processor Omega Network



1

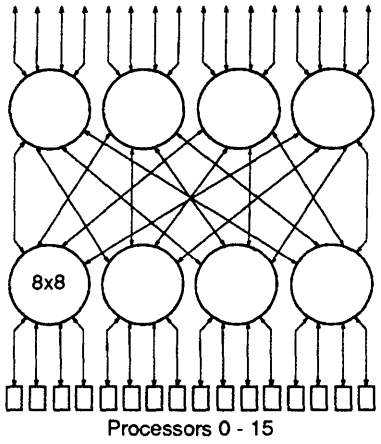
The CS–2 Communications Network 2

CS–2 uses a logarithmic network constructed from 8 way crosspoint switches (see Chapter 3 for details of their implementation) and bidirectional links.

For the purposes of this analysis it can be considered to be a Benes network folded about its centre line, with each switch chip rolling up the functionality of eight of the unidirectional two way switches.

Bandwidth is constant at each stage of the network, and there are as many links out (for expansion) as there are processors. Larger networks are constructed by taking four networks and connecting them with a higher stage of switches. A 16 processor network is illustrated in Figure 2-1.

Figure 2-1 One layer of a 2–stage CS –2 network. 16 processors are connected to stage 1, 16 links connect stage 1 to stage 2, and 16 links are available for expansion.



The scaling characteristics of the CS–2 network are shown in the table below; note that the latency is measured in switch stages for a route which has to go to the highest stage in the network.

Processors	Stages	Total Switches	Latency
4	1	1	1
16	2	8	3
64	3	48	5
256	4	256	7
1024	5	1280	9
4096	6	6168	11

One aspect of implementing the network using bidirectional switches is that routes which are relatively local do not need to go to the high stages of the switch hierarchy. So, for example, a communication to a PE which is in the same cluster of 16 processors only needs to pass through 3 switches irrespective of the total network size.

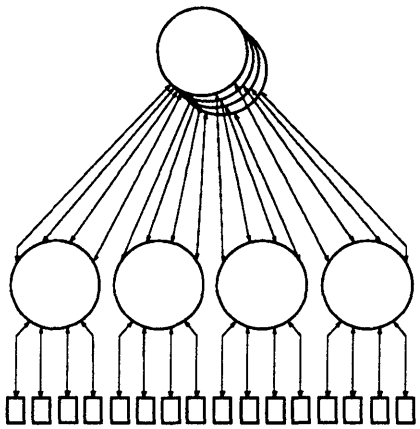
To broadcast to a range of outputs it is necessary to ascend the switch hierarchy to a point from which all the target PEs can be reached. From this point the broadcast then fans out to the target range of processors.

Comparison With Fat-Tree Networks

The multi-stage network used in the CS–2 machine can also be considered as a ``fat tree''. In Figure 2-1 we see that for each of the higher layer switches has identical connections to the lower stages. If this is simply redrawn as shown in Figure 2-2 we get the ``fat tree'' structure.

In fat trees packets do not always have to go to the top of the tree; packets are routed back down at the first node possible. This means that for problems which have locality of reference in communications, bandwidth at higher levels of the tree can be reduced. Exploiting the benefits of locality by reducing upper level network bandwidth has the effect of making process placement more significant. Although the CS–2 network permits this local packet routing, the bandwidth is not reduced in the higher level. This preserves the properties of Benes and Omega networks.

Figure 2-2 One layer of a 16 processor CS-2 network drawn as a fat tree.



Further properties of ``fat trees'' are described by Leiserson[1]

Characterising a CS-2 Network

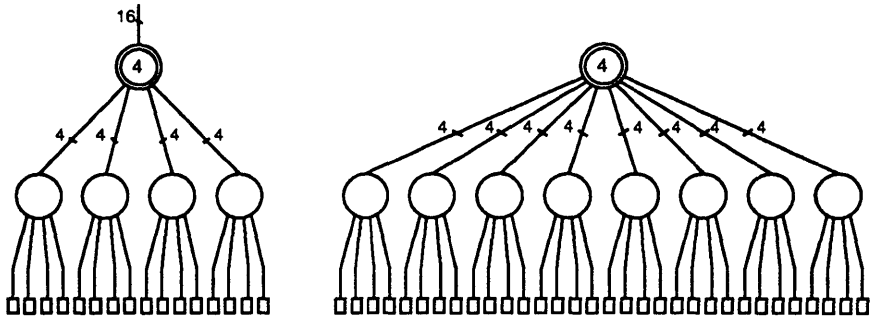
Logarithmic, or multi-stage, switch networks are described in a variety of ways by different people. The scheme used by Meiko is outlined below.

For a machine with *N* processors the size of its network is defined by one parameter: *size*. The position of a processing element is defined by two parameters: *level* and network *identifier*. The position of a switch in the network is defined by four parameters: *layer*, *level*, *network identifier*, and *plane*.

Every processor in a (complete) network is connected via a data link to a switch in the lowest stage, these switches are then connected to higher stages, etc and *N* links emerge from the top of the network. These links can be used to connect to further stages, or if we forgo the ability to expand they can be used to double the size of the network without introducing an extra stage (see Figure 2-3).

1. C.E.Leiserson. Fat-Trees: Universal Networks for hardware-Efficient Supercomputing. IEEE Transactions on Computers, Volume C-34 number 10 (Oct. 1985). pp 892-901.

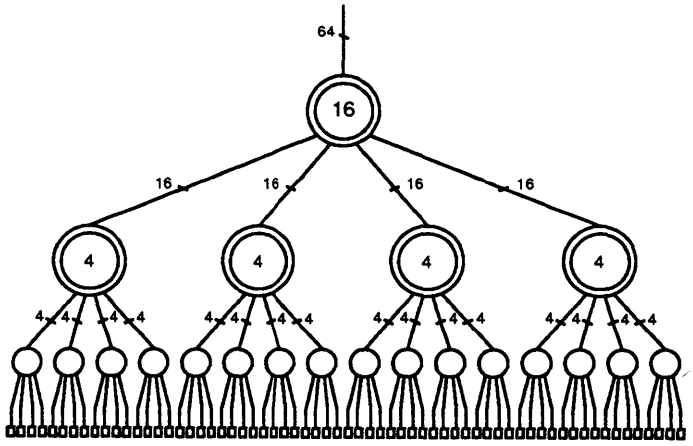
Figure 2-3 Doubling the size of a CS-2 network.



We use a binary form for network size, equal to the number of bits in the network identifier of the lowest processor in the network. This is used because the top stage of the network can use either 4 or 8 links.

A network has [*size*/2] stages, indexed by the parameter level. The top stage is 0. The deepest processors in the network have *level=size*. A network supports between $2^{(size-2)}+1$ and 2^{size} processors. Note: it is not necessary for the switch network to be complete. Figure 2-4 illustrates a network of size 6.

Figure 2-4 One layer of 64 processor (size 6) CS-2 network.



There are a variety of ways of drawing these networks (see the *CS–2 Product Description* for two other examples). To draw (or manufacture!) them without crossing data links you need one more dimension than there are stages in the network.

A CS–2 machine has 2 completely independent identical switch networks. These networks are indexed by the parameter layer. Processors are connected to both layers, switches are in one layer or the other.

The position of each processing element is uniquely determined by its network identifier and level, which describe the route to it from all points at the top of the network (*level*=0). Routes down are written <0-7>.<0-3>.<0-3> ... working down from the top of the network. Each digit represents the output link used on a network switch. For example, in Figure 2-4 processor 0 has route 0.0.0, and processor 17 has route 1.0.1. Note that the route is the same for all starting points at the top of the network. Network identifiers of communications processors (leaves of the network) are sometimes called Elan Identifiers.

Each stage of the switch network has $2^{(size-2)}$ switches, and 2^{level} distinct routes from the top of the network. The network identifier of a switch indexes the distinct routes within each level. Within each stage there are $2^{(size-level-2)}$ switches with the same route from the top of the network.

Network Implementation 3

The CS–2 communications network is constructed from a VLSI packet switch ASIC — the Elite Network Switch. Interfacing between the network and the processors is performed by a second device, the Elan Communications Processor. Switches are connected to each other and to communications processors by byte wide bidirectional links.

The Link Protocols

The choice of a byte wide link protocol is dictated by a number of factors. The link must be wide enough to meet the bandwidth requirements of the processor, but must not be so large that the number of I/O pins on the devices becomes prohibitively large. The implementation that Meiko selected uses 20 wires for each bidirectional link, 10 in each direction. When clocked at 70MHz this yields a bandwidth of 50Mbytes/s (after allowing for protocol overheads) in each direction. This level of performance and the underlying protocol format is appropriate for optic fibre communication over long distances (the link can be converted to a 630MHz data stream).

The use of bidirectional links permits flow control and acknowledge tokens to be multiplexed onto the return link. The low level flow control allows buffering of the data at the line level so that communications clock frequencies in excess of the round trip delay can be used. The interface is asynchronous and is tolerant to a 200ppm frequency difference between the ends. This means that each end can have its own clock, substantially simplifying construction of large systems.

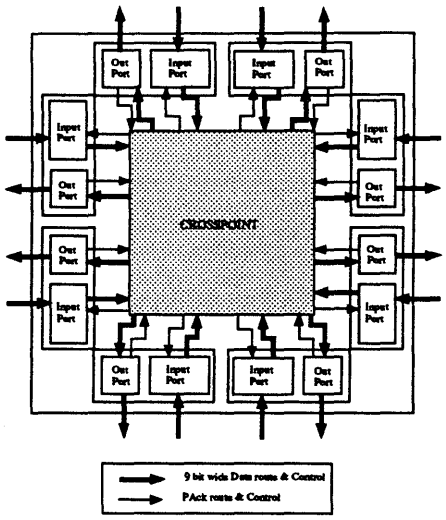
3

The Meiko Elite Network Switch

The Elite switch is capable of switching eight independent links, each byte wide. The switch is a full crosspoint, allowing any permutation of inputs and outputs to be achieved without contention. For each data route through the switch a separate return route exists, ensuring that acknowledgements are never congested by data on the network.

The switch component contains a broadcast function that allows incoming data to be broadcast to any contiguous range of output links. The switch contains logic to recombine the acknowledge or not-acknowledge tokens from each of the broadcast destinations. To allow broadcasts to ranges of outputs over multiple switches the switch topology must be hierarchical.

Figure 3-1 Meiko Elite network switch.



The data passing through a switch is CRC checked at each switch. If a failure is detected the message is aborted, an error count is incremented, and the packet is negatively acknowledged. This ensures that incorrect data is removed from the network as soon as possible.

Routing within the switch is byte steered. On entry into a switch the first byte of any packet is interpreted as the destination output or range of outputs. This byte is stripped off within the switch so that the next byte is used for routing in

the following switch. The latency through each switch device is 7 clock cycles for outgoing data, and 5 cycles for returning acknowledge tokens. The switch contains no routing tables of any sort. The translation between destination processor and route information is performed entirely on the communications processor, where it can be more easily modified or updated.

Although the switch component is an 8×8 crosspoint, the use of bidirectional links means that for the purposes of constructing logarithmic networks the effective radix is 4.

Each switch has a performance monitoring and diagnostic interface connected to the CS–2 control network. This allows collection of statistics on error rates and network loading.

Routing Algorithms

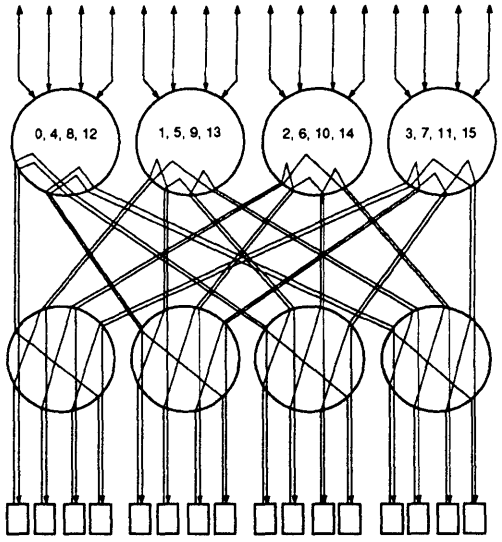
Although the CS–2 data network can have the congestion properties of a full crosspoint, achieving this requires allocation of routes in a non-contending fashion. In the CS–2 network the route is predetermined by the communications processor. By storing the route information in the Elan it becomes easier to change the routing algorithm, due to machine reconfiguration or link failure for example.

The translation from a processor address to network route is handled in the communications processor by a look-up, the table is stored in the memory of the PE and indexed by destination processor. Each table entry contains four alternative routes to the destination processor, one of which is selected. The specification of alternative routes allows the even distribution of traffic throughout the network, although all four routes may be identical when this is undesirable. Each PE maintains its own look-up table which may be different to the others, thus enabling any function of source/destination addressing to be used from.

One simple routing function is to direct all data for the same destination processor through a single switch node at the top of the hierarchy. This allows the network to perform two functions: data distribution, and distributed arbitration for use where many senders wish to communicate with the same processor simultaneously. By adopting this strategy we ensure that if blocking does occur, it does so as soon as possible, and consumes little of the network resource. Using this simple algorithm has the effect of reducing the network to an Omega

network — essentially the second, return part, of the network is guaranteed non blocking, and performs a simple data ordering operation. By virtue of its similarity to an Omega network, this network will be non-blocking for arbitrary shifts and FFT style permutations.

Figure 3-2 Shift by 5 on a 16 processor CS-2 network.



The programmable nature of the CS–2 communication network allows users (who are so inclined) to design their own routing algorithms. This permits optimisation of routing for specific traffic patterns or study of the effect of routing strategy on network performance.

Conclusions 4

The CS–2 network provides a flexible solution to the problem of connecting together large numbers of processing elements. The network can provide equivalent performance to a full crosspoint, but can be simplified where this level of interconnect is not required. The combination of Meiko Elan and Elite network technology allows considerable flexibility in the choice of routing algorithm.

The communications co-processor uses a lookup table to map abstract processor addresses to switch network routes. By maintaining the lookup tables within the PE memory they are easier to modify to reflect changing workload or network failures. By maintaining separate lookup tables on each communications processor, any function of address mapping may be implemented. The Elan communications processor acts as a gateway into the CS–2 switch network.

The Elite network switch is a full 8×8 crosspoint switch. It is the fundamental building block of the CS–2 communications network. The route through the switch is determined by the header byte of each incoming message. Headers are added by the communications processor and removed by the switch as the message passes through it. In addition to a direct mapping from input link to output link, the switch supports broadcast and combining operations by mapping a single input to a contiguous range of outputs.

Computing
Surface

Vector Processing Element Overview

S1002–10M101.05

meiko

Meiko's address in the US is:

Meiko
130 Baker Avenue
Concord MA01742

508 371 0088
Fax: 508 371 7516

Meiko's full address in the UK is:

Meiko Limited
650 Aztec West
Bristol
BS12 4SD

Tel: 01454 616171
Fax: 01454 618188

Issue Status:

Draft	
Preliminary	
Release	x
Obsolete	

Circulation Control: *External*

Contents

General Description 1

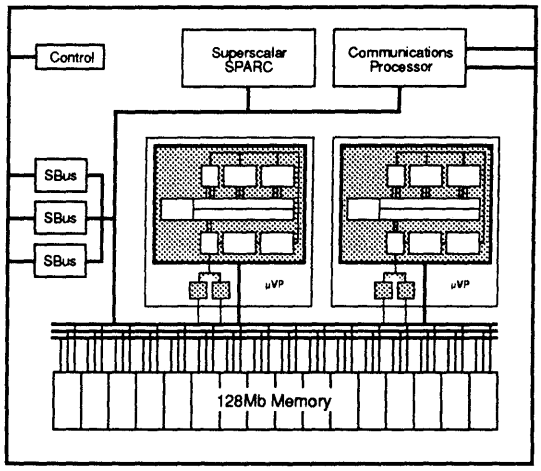
This document describes the architecture of the CS–2 vector element (MK403). It briefly describes the internal architecture of the Fujitsu μVP and the compilation strategy used to exploit the combined resources of the SPARC and multiple μVP processors.

For more details of the workings of the μVP see the *μVP Programmers Reference Manual.*

MK403 Overview

The CS–2 vector element incorporates a 40MHz Superscalar SPARC, a Meiko Elan Communications Processor and 2 Fujitsu μVP vector processors. All processors have access to the memory system via 3 memory ports, two of which are used by the vector processors and the third by the SPARC and Elan (which share an MBus).

Figure 1-1 CS-2 Vector Processing Element.



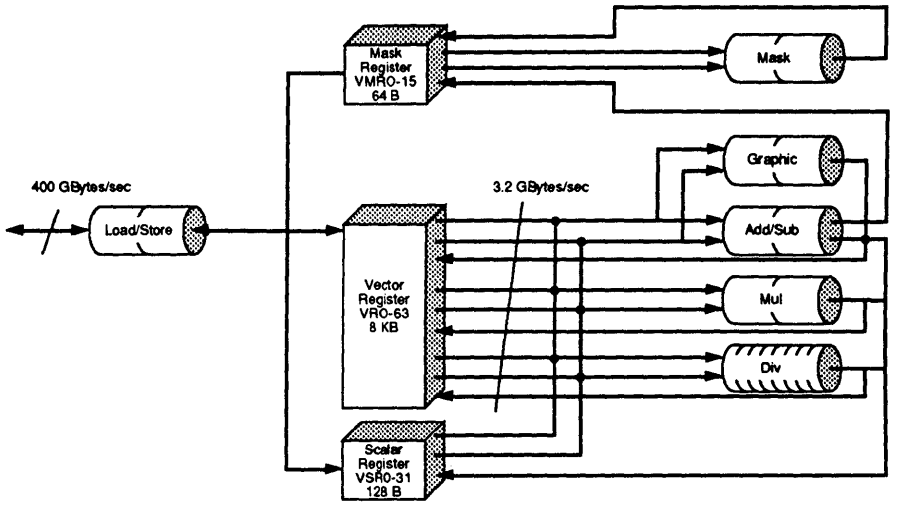
The memory system is implemented as 16 independent banks, with a (current) total capacity of 128 Mbytes. Memory bandwidth for each of the 3 ports is 1.2 Gbytes/s, with a total bandwidth of 3.2 Gbytes/s.

External I/O support is provided through 3 SBus interface slots — primarily used for disk controllers, but capable of supporting network interfaces and graphics cards.

µ*VP Vector Processor*

The µVP operates with a 50 MHz (20ηs) clock. It has a vector register architecture with 8 Kbytes of vector registers, configurable as between 8 and 64 vectors each of 16–128 64-bit registers (see below). In addition there are 32 scalar registers and a set of vector mask registers whose format tracks that of the vector registers.

Figure 1-2 μVP Vector Processor.



Configuration of the VP vector and mask registers:

Precision	Length	Number of registers
Single	32	64
Single	64	32
Single	128	16
Single	256	8
Double	16	64
Double	32	32
Double	64	16
Double	128	8

The μVP has separate pipes for floating point multiply, floating point add, floating point divide, and integer operations. The floating multiply and add pipes can each deliver one double precision (64bit) or two single precision (32bit) IEEE format result(s) on every clock, giving a maximum theoretical performance of 100MFLOPS/s double precision and 200MFLOPS/s single precision; the divide pipe can simultaneously deliver an extra 6MFLOPS/s in either single or double precision. Both the add and multiply pipes have the low latency (pipe depth) of two cycles (40ηs), with one extra cycle being required to read and one to write the vector register file.

The vector register elements are scoreboarded, so that chaining between input and output operands occurs wherever possible without requiring explicit compiler or programmer intervention.

The μVP has a single load/store pipe which is used for accessing the memory system. This is a 64bit interface which can generate four addresses on consecutive clock cycles before stalling for the returned data. Once the data is present a 64bit word can be transferred on each clock cycle, giving a maximum bandwidth of 400Mbytes/s.

The instruction set includes masked vector operations, compressions (*sum, maxval, maxindex, minval, minindex*), vector compress under mask and expand under mask operations, as well as logical operations on integers and mask registers and conditional branches. Vector loads and stores can be performed with strides and under mask, as well as with an index vector (``indirect"). For further information about the μVP instruction set the *μVP Programmers Reference Manual.*

Superscalar SPARC Processor

The MK403 uses SPARC MBus processor modules. It is generally populated with a 36 or 40MHz Viking SPARC, but other standard modules can be used.

The Superscalar SPARC has two independent integer ALUs which can execute separate arithmetic operations or can be cascaded so that the processor can execute two dependent instructions in the same cycle. It has instruction issue logic which can issue up to three instructions on the same cycle. Load and stores operations of all data types to the on chip 16Kbytes data cache occur in a sin-

gle cycle. The floating point unit can execute multiply and add instructions simultaneously, though only one floating point instruction can be issued per cycle.

Memory System

The Superscalar SPARC processors and Elan communication processor are connected to a standard 40MHz MBus. The vector processors and MBus are connected to a 16 bank memory system, each bank providing 64bits of user data (78bits including error checking and correction, implemented using 20 by 4bit DRAMs with two bits unused). Error detection and correction is implemented on each half word (32bits), allowing write access to 32bit (ANSI-IEEE 754–1985 `single`) values to be performed at full speed, without requiring a read modify write cycle.

Each bank of memory maintains a currently open DRAM page within which accesses may be performed at full speed. This corresponds to a size within the bank of 8Kbytes, giving 128Kbytes total for the 16 banks. When an access is required outside the currently open page a penalty of 6 cycles is incurred to close the previous page, and open the new one.

Refresh cycles are performed on all banks within a few clock cycles of each other, thus allowing the cost of re-opening the banks to be pipelined (since the VP can issue four addresses before stalling for the data from the first), and reducing the overhead of refresh to a few percent of memory bandwidth.

The memory system is clocked at the same speed as the μVP processors (50MHz), and accesses from the 40MHz MBus are transferred into the higher speed clock domain. When accessing within an open page each memory bank can accept a new address every two cycles (40ηs), and replies with the data four cycles (80ηs) later, giving a bandwidth of 8bytes every two cycles (40ηs), that is 200Mbytes/s. Since there are 16 banks, the total memory system bandwidth is thus 3.2Gbytes/s.

Each μVP can issue a memory request every cycle (20ηs), and can issue 4 addresses before it requires data to be returned. In the absence of bank contention (which will be discussed below), after a start up latency of four cycles, these requests can be satisfied as fast as they are issued, giving each μVP a steady state bandwidth of 8bytes every 20ηs, that is 400Mbytes/s.

Since each bank can accept a new address every two cycles (40ηs), but the μVP can generate an address every cycle (20ηs) there is the possibility of bank contention if the μVP generated repeated accesses to the same bank. With a simple linear mapping of addresses to banks, this would occur for all strides which are multiples of 16 (for 64bit double precision accesses). Such an access pattern would then see only one half of the normal bandwidth, that is 200Mbytes/s. All other strides achieve full bandwidth.

To ameliorate this problem as well as allowing the straightforward linear mapping of addresses to banks, Meiko also provide the option (through the choice of the physical addresses which are used to map the memory into user space) of scrambling the allocation of addresses to memory banks. The mapping function has been chosen to guarantee that accesses on "important" strides (1, 2, 4, 8, 16, 32) achieve full performance. Access on other strides may see reduced performance, but there are no strides within the open pages which see the pathological reduction to one half of the available bandwidth.

Compilers 2

Overview

The Fortran and C compilers for the vector processing element generate code for all three processors: using the scalar processor to execute scalar code, and the two μVPs to execute vector loops. They incorporate a wide range of standard optimisations:

> constant folding, constant propagation, common subexpression removal, **automatic function inlining,** instruction scheduling, loop invariant removal, induction variable detection, **software loop pipelining, loop splitting, loop interchange, loop vectorisation, vectorisation of intrinsic functions, vector idiom recognition,** dead code removal,

as well as proprietary optimisations for the CS–2.

Languages

Fortran, C, High Performance Fortran (HPF) and Fortran–90 are supported.

FORTRAN and C

The FORTRAN language conforms to ANSI X3.9–1978, with the addition of many extensions including CRAY Pointers, ALLOCATABLE arrays and COMMON blocks, VMS structures, END DO statements, and NAMELIST I/O. The compiler also recognises the CRAY vectorisation directives (for example, CDIR$IVDEP).

The C compiler accepts the ANSI C language, and incorporates the same vectoriser and code generator as the FORTRAN compiler.

High Performance Fortran (HPF)

The High Performance Fortran Forum (HPFF) is a group of industrial and academic organisations which is open to all. The objective of the group is to standardise annotations and extensions to ISO 1539:1991 (Fortran–90) to allow a Fortran program to be efficiently executed under a data parallel execution model. HPFF have published the final draft specification for public comment. A HPF compiler for the CS–2 is currently under development.

Fortran–90 Binding

The HPFF has chosen Fortran–90 as the language for extension. The new dynamic storage allocation and array calculation features make it a natural base for HPF. The HPF language features fall into 3 categories with respect to Fortran–90:

- New directives.
- New language syntax.
- Language restrictions.

The new directives are structured comments which suggest implementation strategies or assert facts about a program to the compiler. They may affect the efficiency of the computation performed, but do not change the value computed by the program. The form of the HPF directives has been chosen so that a future Fortran standard may chose to include these features as full statements in the language.

A few new language features, namely the FORALL statement and certain intrinsics, are also defined. They were made first-class language constructs rather than comments because they can affect the interpretation of a program, for example by returning a value used in an expression. These are proposed as direct extensions to the Fortran–90 syntax and interpretation.

Full support of Fortran sequence and storage association is not compatible with the data distribution features of HPF. Some restrictions on use of sequence and storage association are defined. These restrictions may in turn require insertion of directives into standard Fortran programs in order to preserve correct semantics.

New Features in High Performance Fortran

High Performance Fortran extends Fortran in several areas. These areas include: data distribution features, parallel statements, extended intrinsic functions, foreign procedures and changes in sequence and storage association.

Code Generation

The compilers for the vector processing elements produce code that executes on the SPARC, and, dynamically if appropriate, on the two attached vector processors. Scalar code executes on the Superscalar SPARC processor, vector code is compiled to execute on either the SPARC processor or the μVPs, or both.

Where the vector length is not known at compile time, the compiler generates both vector code (for the μVPs) and scalar code: the choice of which code to execute being made at run time based on the actual vector length.

The vectoriser exploits the multiple μVPs in two different ways. Where there is a loop around a vector loop, as shown below, the compiler will generate code which executes alternative iterations of the outer loop on each of the μVPs; each instance of the inner loop (and its strip-mine loop) will execute entirely on a single μVP:

```
DO I = 1,N
   DO J = 1,M
      X(J,I) = A*X(J,I) + Y(J)
   END DO
END DO
```

The generated code is analogous to the following (pseudo) source code:

In parallel on μVP 1:

```
DO I = 1,N,2
   DO J = 1,M
      X(J,I) = A*X(J,I) + Y(J)
   END DO
END DO
```

and on μVP 2:

```
DO I'=2,N,2
   DO J' = 1,M
      X(J',I') = A*X(J',I') + Y(J')
   END DO
END DO
```

Where there is no outer level independent loop which can be exploited, then the compiler will split the individual strips of the inner loop across the two μVPs. Consider the following example:

```
DO J = 1,M
   X(J) = A*X(J) + Y
END DO
```

The generated code is analogous to the following (pseudo) source code:

In parallel on μVP 1:

```
   IBASE = 1
   ILEN = MIN(M-IBASE, stripLength)
C  Strip mine loop
   DO WHILE (ILEN .GT. 0)
C     Vector operation
      DO J = IBASE,IBASE+ILEN
         X(J) = A*X(J) + Y
      END DO
C     2 here is number of uVPs involved
      IBASE = IBASE + 2 * stripLength
      ILEN = MIN(M-IBASE, stripLength)
   END DO
```

and on μVP 2:

```
      IBASE' = stripLength
      ILEN' = MIN(M-IBASE', stripLength)
C     Strip mine loop
      DO WHILE (ILEN' .GT. 0)
C        Vector operation
         DO J' = IBASE,IBASE+ILEN'
            X(J') = A*X(J') + Y
         END DO
C        2 here is number of uVPs involved
         IBASE' = IBASE' + 2 * stripLength
         ILEN' = MIN(M'-IBASE', stripLength)
      END DO
```

All of this code executes on the μVP.

The code generator schedules vector instructions to ensure that chaining of vector operations happens as often as possible (by ensuring that there are no scalar operations scheduled between dependent vector operations).

If the operation is a vector sum, then each μVP will produce the sum of the elements it processes, and the final accumulation of the two partial sums will be performed by the scalar processor.

2

Conclusions 3

Each CS–2 vector processing element consists of a Superscalar SPARC, a Meiko Elan communications processor, and 2 Fujitsu μVP vector processors sharing a three ported memory system. Cycle time is 20ηs, performance peaks at 200MFLOPS/s per processing element in 64bit arithmetic, or 400MFLOPS/s in 32bit.

To achieve high performance on real world problems you need the correct balance of CPU and memory system performance. The CS–2 vector memory system is organised as 16 independent banks, enabling it to sustain 1.2Gbytes/s on direct, strided, or indirect addressing. Memory capacity is currently 32 or 128Mbytes per processing element.

The CS–2 development environment for the vector processing elements includes compilers for FORTRAN–77, ANSI C, Fortran–90, and High Performance Fortran. The compilation system produces compiled code that executes on either the SPARC processor or, dynamically where appropriate, on the two attached vector processors.

Computing Surface

Getting Started — Users' Guide

S1002–10M117.02

meiko

Meiko's address in the US is:

Meiko
130 Baker Avenue
Concord MA01742

508 371 0088
Fax: 508 371 7516

Meiko's address in the UK is:

Meiko Limited
650 Aztec West
Bristol
BS12 4SD

Tel: 01454 616171
Fax: 01454 618188

Issue Status:

Draft	
Preliminary	
Release	x
Obsolete	

Circulation Control: *External*

Typographic Conventions *i*

The following typographic conventions are used in this document and all other Meiko documentation:

Library reference.

Italicised text is used for references to other documents, or emphasised expressions that may be expanded later in the text. Also used in example command lines in place of site specific options.

> See the document *Tagged Message Passing and Global Reduction.*

> `Password:` *password*

Do not copy this.

Emboldened text is used to emphasise expressions of particular importance.

> It is important that you **do not try this yourself.**

`integer name`

Courier is used for variable names, command names, filenames, and other text that might be entered into the computer system, or for the computer's response to a user's request. See also the use of bold courier below.

> The function `csn_init()` must be called before others in the library.

cat file

Bold courier is used when illustrating a dialogue between the computer and a user. Text entered by the user is shown in this font, text displayed by the machine is shown in courier.

```
user@cs2-1: ls /opt/MEIKOcs2
bin/ docs/ example/ include/ lib/ man/
```

Warning – Used to draw the reader's attention to an important note.

Contents

General Introduction 1

This document is a users' guide to the CS-2. It describes how to login to the CS-2, how to familiarise yourself with the operating system, and how to create parallel programs. This document also provides a user view of the resource management system — the software that controls user access to the processing resources.

The following chapters are included:

Chapter 2, *Using a CS-2 for the First Time*

Describes logging in to the CS-2, basic Unix commands, on-line documentation systems (manual pages and the AnswerBook system), and how to run parallel programs.

Chapter 3, *Resource Management*

Provides an overview of the Resource Management System and describes the user interface to it.

Chapter 4, *Resource Management Commands*

Describes resource management commands, with examples of typical usage.

Chapter 5, *Parallel Programming*

Describes the message passing libraries, and some simple parallel programs.

Chapter 6, *The Meiko Parallel Filesystem*

Describes the user interface to the Meiko parallel filesystem.

1

Chapter A, *Glossary*

A glossary of terms used throughout this document.

Using a CS-2 for the First Time 2

UNIX on a CS-2 System

CS-2 runs the Solaris operating system, the same Unix operating system as a Sun Microsystems SPARC workstation or server. In fact it is designed to be binary compatible with such machines, any program that runs on your SPARC workstation will run on a CS-2. As well as having the same operating system, CS-2 has the same command shells, editors, compilers, linker, and libraries and it runs the same applications packages. If a Solaris application doesn't run immediately it is likely to be a licensing issue.

If you are not familiar with Solaris, or have never used a Unix system before then refer to the *SunOS Users Guide* (version 5.0 or later). The *Solaris Roadmap to Documentation* outlines the full documentation set[1]. The standard textbook on Unix is *The UNIX Programming Environment; Kernighan/Pike*, it provides a general introduction to the standard Unix utilities and commands shells. For information on the windowing system you should refer to the *OpenWindows Users's Guide* (version 3.0.1 or later) or the help systems (see below).

CS-2 differs from a conventional SPARC system in two important areas: CS-2 runs parallel applications as well as sequential, and processes running on a CS-2 system are controlled by its Resource Management System (see page 15 for introductory details).

1. The Solaris documentation is viewable on-line with the AnswerBook documentation system.

2

In the following sections we describe how you login in to your CS-2, how to set-up your login environment, and some common Unix and CS-2 commands.

Logging In

CS-2 systems are generally connected to a local area network, and used from workstations or network connected terminals. To login from a workstation you should use `rlogin` or `telnet`, typing:

```
workstation: telnet cs2
```

This document uses `cs2` as the name of the CS-2 system and `workstation` as the name of your local workstation. Your system is likely to have different names, if you don't know them then ask your System Administrator.

If you are using the CS-2 system from a directly connected terminal hit return and the machine should respond immediately. Network connected terminals have a variety of connection commands (we have to type `open cs2` on ours). Again your System Administrator or Network Manager will know all about this.

Once a connection to the machine is established you will get a `login` prompt (a banner giving the machine name may also appear); you should respond with your account name. The system will then display a `password` prompt, enter your password, it will not appear on the screen.

```
UNIX(r) System V Release 4.0 (cs2)

login: user
Password: password
```

The system will verify your password, and if it is correct you will be logged in. The message of the day file will be output, initialisation files run and a prompt will appear.

```
Last login: Fri Jul 9 20:29:38 from workstation
SunOS 5.3 MEIKO FCS 08 June 94
user@cs2:
```

A word of warning at this point. Unix security is based on keeping passwords secret. If other people know your password then they can interfere with your work. The operating system provides a controlled means of sharing work and data, keep your password secret. You can change your password with the `passwd` command; the documentation for this command also gives useful guidelines for choosing a secure password.

You can now use the machine; to run a program simply type its name at the command prompt. To find out the name of your home directory type `pwd` and to list your files type `ls`:

```
user@cs2: pwd
/home/cs2/user
user@cs2: ls
bin/ example/ include/ lib/
```

The Command Shell

Whenever you login to a Unix system a command shell is started for you. The command shell is the interpreter that parses commands and executes them. A number of different types of shell are available, the most common being either the C shell and the Bourne shell. The principal difference between the shells is command syntax.

The type of command shell that you use is specified by your System Administrator. It is important that you understand which shell you are using because some commands described in this manual are shell-specific.

Use the following command to check the shell type:

```
user@cs2: echo $SHELL
/bin/csh
```

`csh` is the C-shell, `sh` is the Bourne shell (there are others).

You can find out more information about your shell by referring to the Solaris documentation, or by using the on-line manual pages (see below). Most shells offer command aliasing (allowing you to define a simple alternative name for a command line), variables, job control (executing several jobs concurrently as background processes), and flow control (`if then else` type constructs). If you don't like your default shell you can ask your System Administrator to change it, or you can simply start a new shell from your command line.

Shell Scripts

All command shells read a shell script when they are started: the C-shell reads a file called `.cshrc`, and the Bourne shell reads `.profile` (both from your home directory). In addition when you first login the system reads the `.login` file, and when you logout it reads the `.logout` file (if these exist). These files are shell scripts: they contains shell commands that are executed as if you had entered then at the command line. Default start-up shell scripts will have been defined by your System Administrator; they will define common command aliases and environment variables for use at your site.

You can create shell scripts yourself to describe commonly used command sequences. In most cases you add the commands into a file and pass the filename argument to the shell program[1]. The following simple shell script is created with `cat` and executed by the Bourne shell.

```
user@cs2: cat > cleanup
rm *.o *.s
rm *.ps *.dvi
^d
user@cs2: sh -c cleanup
```

Shell scripts that make full use of the shell's command syntax can perform quite complex tasks. Most of Meiko's software installation procedures are implemented with shell scripts.

Checking your Environment

All commands that you execute are passed a list of environment variables that describe the command's environment. A number of these variables will have been defined by your System Administrator in your shell's start-up files. In particular the PATH variable, which is used by your shell and other programs to locate executable programs, must include the Meiko `bin` directory, and the MANPATH variable, which is used by the `man`(1) command to locate on-line manual pages, must include the Meiko `man` directory. If you are using a workstation with a graphics display you must also set the DISPLAY environment variable to identify the display's name to graphics applications.

To check the current setting of these variable use the `echo` command:

1. Each shell offers different mechanisms for reading commands from shell scripts; consult the documentation for your shell.

```
user@cs2: echo $PATH
.:/opt/MEIKOcs2/bin:/usr/bin:/bin
user@cs2: echo $MANPATH
/opt/MEIKOcs2/man:/usr/man
user@cs2: echo $DISPLAY
workstation:0
```

The way in which you set (or change) an environment variable depends on the shell you are using. The following examples are for the C-shell and Bourne shell respectively. Note that the PATH and MANPATH variables take a list of directories; the following examples prefix the Meiko directories onto the existing (possibly empty) definition of the variable:

```
user@cs2: setenv DISPLAY workstation:0
user@cs2: setenv PATH /opt/MEIKOcs2/bin:$PATH
user@cs2: setenv MANPATH /opt/MEIKOcs2/man:$MANPATH
```

```
user@cs2: DISPLAY=workstation:0
user@cs2: export DISPLAY
user@cs2:
user@cs2: PATH=/opt/MEIKOcs2/bin:$PATH
user@cs2: export PATH
user@cs2:
user@cs2: MANPATH=/opt/MEIKOcs2/man:$MANPATH
user@cs2: export MANPATH
```

You can add the appropriate commands into your shell's start-up shell script so that they are executed automatically whenever you login.

Starting a Windowing Systems

If you are using a workstation with a graphics display you will want to start a windowing system. This will allow you to start several command shells, each in a separate window, and will allow you to run graphics applications such as the AnswerBook (the Solaris documentation system) — described later.

To start the Open Windows system use the `openwin` command:

```
user@cs2: openwin
```

The appearance of your windows can be tailored by a number of configuration files in your home directory which your System Administrator may have defined for you (their filenames will begin with either `.x` or `.openwin`). Consult the Open Windows documentation for more information about these files.

Getting Help

Meiko's documentation (and where possible that for third party products) is distributed in four formats:

- Printed documentation — at least one copy will be supplied with all systems.
- PostScript — all Meiko's documentation will be supplied in this format; third party documentation is also included when product licences permit.
- Manual pages — included as part of each software release.
- AnswerBook — a hypertext system containing all of Meiko's documentation.

Manual Pages

Manual pages provide concise summaries of commands and files. They are useful if you already know something about the command/file that you are querying, but rather less useful if you don't.

You can use the `man` command to provide information about itself by typing:

```
user@cs2: man man
```

The following example will tell you about the C-shell:

```
user@cs2: man csh
```

At the end of each page of information `man` pauses; press the space bar to read the next page, or `q` to quit (i.e. `man` uses the `more` command to display the information).

AnswerBook

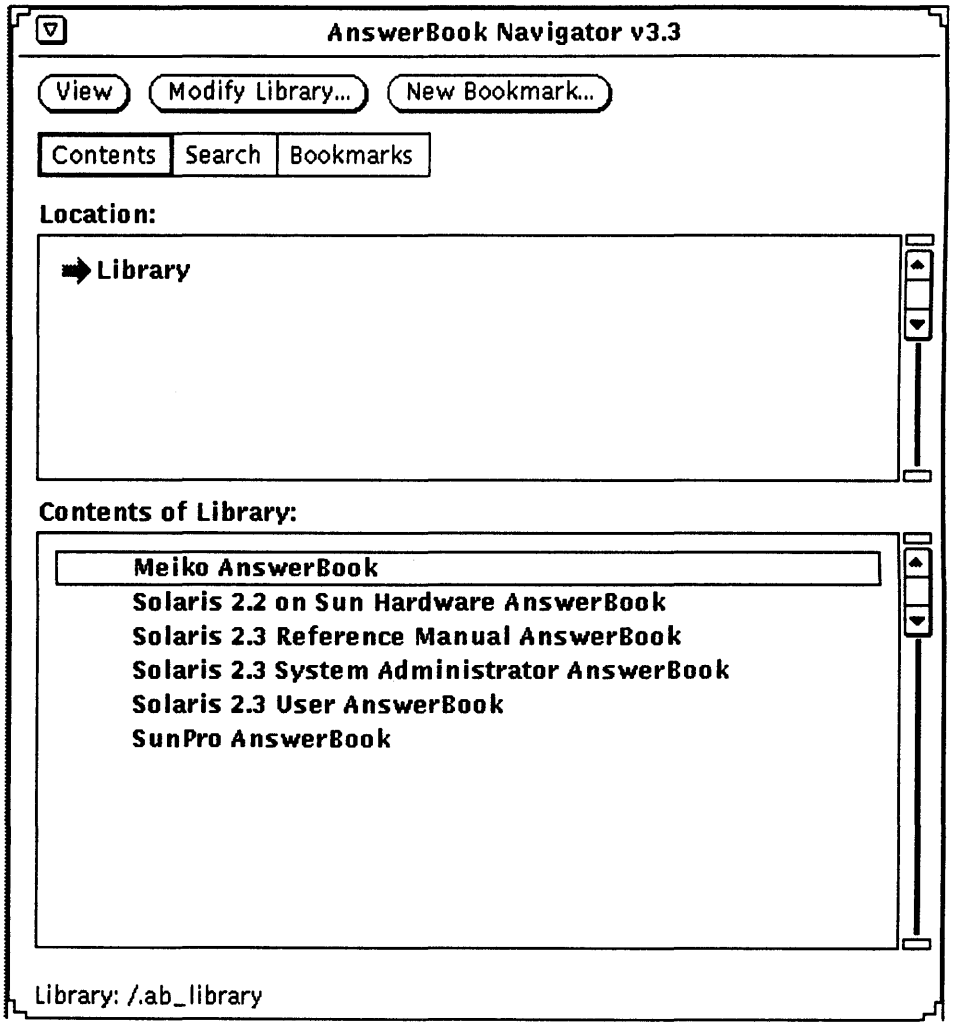
The AnswerBook system is more friendly but you will need to be running OpenWindows Version 3.0.1 (or later) before you can use it. To start the AnswerBook on the console type:

```
user@cs2: answerbook
```

Some time later (it takes a while to start) a window will appear; this is the AnswerBook navigator (see Figure 2-1).

To view a document use your mouse to select entries in the Contents window; a double click on a document chapter or hypertext button will pop-up a Viewer window displaying the requested part of the manual. You can use the search button near the top of the Navigator to perform a keyword search over all the installed AnswerBooks; a double click on an entry in the Documents Found list will take you to the most appropriate section in that manual.

Figure 2-1 The AnswerBook Navigator



Exploring the Meiko Directory Hierarchy

The structure of the CS-2 software system follows that used in a standard Solaris installation, with Meiko specific code under `/opt/MEIKOcs2`. The following directories will be present:

`/opt/MEIKOcs2/bin`	Executables.
`/opt/MEIKOcs2/docs`	PostScript documentation source.
`/opt/MEIKOcs2/example`	Example programs.
`/opt/MEIKOcs2/include`	Header files.
`/opt/MEIKOcs2/lib`	Libraries.
`/opt/MEIKOcs2/man`	Manual pages.

You should be able to access all these directories, read the `man` pages, run the example programs and print out the documentation.

Running A Parallel Program

Having logged in for the first time we suggest that you run a simple parallel program. To do this you need to use `prun`:

```
user@cs2: prun uname -a
SunOS cs2-0 5.3 MEIKO_FCS dinol sparc
SunOS cs2-1 5.3 MEIKO_FCS dinol sparc
SunOS cs2-2 5.3 MEIKO_FCS dinol sparc
SunOS cs2-3 5.3 MEIKO_FCS dinol sparc
```

This is an example of a very simple parallel application, in which a number of copies of a sequential program are executed on your CS-2. There is no inter-process communication.

An example of a communicating parallel application is `dping`, one of the compiled demonstration programs. It executes on two processors and shows you how fast you can move data between processors as a function of the size of the data transfer. A synopsis for `dping` is available by invoking the command with the `-h` option:

```
user@cs2: prun -n2 dping 0 8k
0(4): fn ping, reps 10000, dma normal
  0 pinged   1:    0 bytes 0.000009Sec  0.000Mb/s
  0 pinged   1:    1 bytes 0.000011Sec  0.090Mb/s
  0 pinged   1:    2 bytes 0.000011Sec  0.181Mb/s
  0 pinged   1:    4 bytes 0.000011Sec  0.364Mb/s
  0 pinged   1:    8 bytes 0.000011Sec  0.725Mb/s
  0 pinged   1:   16 bytes 0.000011Sec  1.452Mb/s
  0 pinged   1:   32 bytes 0.000011Sec  2.899Mb/s
  0 pinged   1:   64 bytes 0.000011Sec  5.777Mb/s
  0 pinged   1:  128 bytes 0.000012Sec 10.642Mb/s
  0 pinged   1:  256 bytes 0.000015Sec 17.134Mb/s
  0 pinged   1:  512 bytes 0.000022Sec 23.703Mb/s
  0 pinged   1: 1024 bytes 0.000032Sec 31.612Mb/s
  0 pinged   1: 2048 bytes 0.000056Sec 36.897Mb/s
  0 pinged   1: 4096 bytes 0.000100Sec 40.842Mb/s
  0 pinged   1: 8192 bytes 0.000199Sec 41.223Mb/s
```

In both the above examples `prun` executes the example program on a partition; a group of processors and their I/O devices. The following chapter provides an introduction to the CS-2 resource management system; subsequent chapters describe the command interface the resource management system, including a more detailed description of `prun`, and lists the source code for some simple parallel programs.

Resource Management 3

The role of the resource management system is to allow a System Administrator to optimise the use of the resources in a CS-2 system. It does this by controlling user requests to login and to run parallel applications, by controlling access, accounting usage, and by visualising system performance.

Resource Partitioning

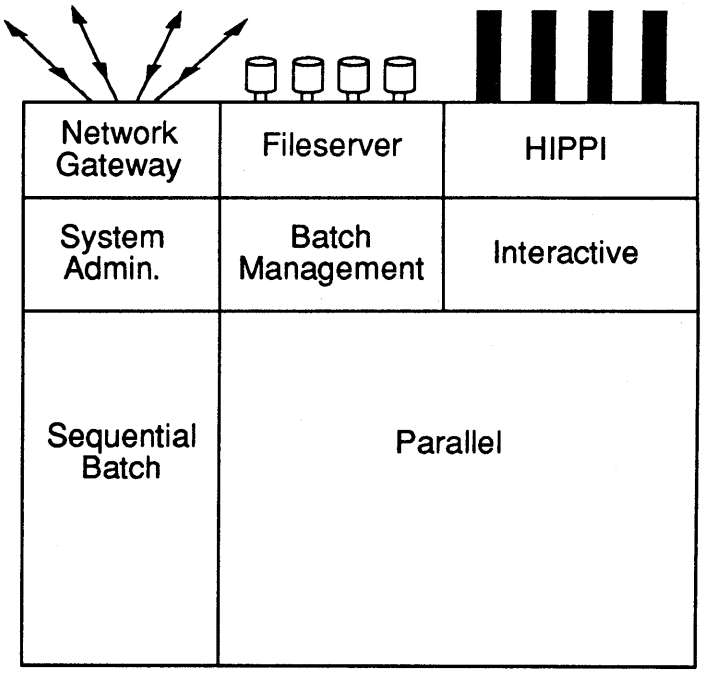
A CS-2 system consists of one or more *partitions*. A partition is simply a collection of resources (processors and their I/O devices) dedicated to a specific task or tasks. Partitions are created because the System Administrator wants to allocate specific resources to different tasks or because it is appropriate to run different scheduling policies on different parts of the machine. Consider the following types of resource usage:

- Development of new parallel programs.
- Production execution of parallel programs.
- Running conventional Unix processes.
- A distributed system service used by conventional Unix processes such as a parallel database server.
- Device management.

A given machine may have to support one or more of these processing loads. By allocating each to a partition and allocating appropriate resources to that partition the System Administrators control the resources used for each type of task.

Figure 3-1 illustrates a partitioning scheme for a large CS-2 system designed to support several hundred users in a production environment, it serves to illustrate a number of further examples of partitions.

Figure 3-1 Partitioning a large CS-2 system



Users login to the system and are connected to processors in the *interactive* partition. The connections are managed by processors in the *network gateway* partition, which are dedicated hosts of the network adaptors and *login load balancers*; these processors do not run user processes.

The login load balancer, `logbal`, is executed on a processor in the network gateway partition whenever a user logs in. `logbal` liaises with the resource manager to identify the least heavily loaded processor within the interactive partition, using criteria specified by the System Administrator. The user's shell is

then executed on the nominated processor with I/O transferred from user to login shell via `logbal`. The actions of `logbal` are largely transparent to users; you may notice that login shells are hosted by a different processor each time you login.

A *fileserver* partition defines the processors that are dedicated to high performance disks. Additional processors are dedicated to managing HiPPI connections and are only used by user processes that access these devices.

Users run applications interactively or by submitting them to the batch system. Specific resources can be dedicated to running particular batch queues. The bulk of the system runs user applications in the *Sequential Batch* and *Parallel* partitions, the balance between them depending upon whether the system is used in *capacity* mode (large numbers of small jobs) or *capability* mode (smaller numbers of large jobs).

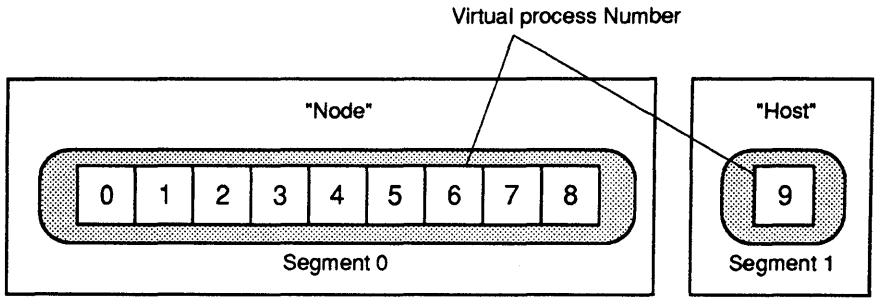
In a large configuration the resource management and accounting systems consumes resource; they can run on one or more dedicated processors.

Parallel Programs

CS-2 supports multi-segment parallel programs in which each segment consists of some number of copies of a single executable. The simplest example is a Unix process, which contains one segment and one process. A parallel application has 2 segments, the first a controlling process in the interactive Unix or batch management partition and the second multiple copies of the same executable running in a parallel processing partition (see Figure 3-2).

The controlling process is the application's interface to the resource management system and is also the process that handles the screen and keyboard I/O requirements from the other processes. The controlling process can be a part of the parallel application, communicating and cooperating with the other processes to complete the task (i.e. a 'hosted' application), or simply the means of starting it (e.g. `prun`).

Figure 3-2 A 2 segment parallel program



Every process in a parallel program has a unique process id; this is translated to a physical processor id by the inter-processor communication hardware.The ordering of segments and hence the numbering of processes depends upon the programming model; some number the controlling (or host) process 0 and the processes of the parallel program 1,2,....n, others number the parallel program 0,1,2,...n-1 and the host n.

Multi-segment programs are created by linking the application's controlling process with the resource management user interface library. The `prun` utility (described later) is a general purpose parallel program loader that is built upon this library; it creates a two segment application in which `prun` itself runs in one segment and multiple copies of a user specified program run in a second.

The application interface to the resource management system is described in the manual *Resource Management User Interface Library.*

Resource Allocation

The execution of a parallel application requires that processing resources (processors and their associated I/O devices) are *allocated* and that the application's processes are then *spawned* onto this resource. Allocation means that the resource is exclusively granted to the application, and spawning means that the application's processes are loaded onto the resource and are executed.

In many cases both stages are handled within the application's controlling process. The controlling process competes with other applications for the resource, blocks until it is available, and then holds the granted resources until completion (which may be forced by the System Administator or by system time-limits). While the application holds the resource other applications, including those belonging to the same user, are blocked until the resource is freed. When you executed `prun` in the previous chapter, `prun` liaised with the resource management system for *X* processors on partition *Y*, and these were then held until your program had completed.

In some cases it is desirable that a *user* may be granted exclusive use of a resource. It may be important that a sequence of the user's applications are run concurrently and without interruption by other users, or it may be desirable to run several concurrent applications on the same resource (e.g. debugging and application processes running side by side). In this case resource may be allocated to a command shell using `allocate`(1), allowing the resources to be held indefinitely and made available to all applications that are executed by the shell. When executed in this environment a controlling process (such as `prun`) will spawn its processes onto the shell's resource; `prun` will not allocate resource itself, and the resource will remain allocated when `prun` completes. Only when the shell terminates will its associated resource be freed.

User Interface to the Resource Management System

The interface to the resource management system is via the user interface library. Meiko have developed a number of utilities from this library that allow you to query resource usage and execute parallel programs. `prun`(1) and `allocate`(1) are both built upon this library.

The following chapter describes the resource management commands in detail. For more information about the user interface library see the *Resource Management User Interface Library.*

Resource Management Commands 4

The following command interfaces to the resource management system are provided:

`prun`	Run a parallel program.
`allocate`	Allocate resources for a sequence of parallel jobs.
`rinfo`	Get information on free resources and running jobs.
`reduce`	Reduction operator for shell script programs.
`gkill`	Send a signal to a process anywhere in the machine.
`gps`	Global process status.
`copyback`	Collect distributed files.
`copyout`	Distribute files over a partition.
`pdebug`	Inspect state of parallel program or core files.

prun — Run a Parallel Program

prun executes a parallel program, or multiple copies of a sequential program, on the CS-2. It spawns multiple copies of a specified executable image onto resource that is allocated by the resource management system.

Identifying the Resource

You identify the processors that will host your application by identifying a partition and the number of processors required. If prun is executed within a shell that has resources allocated to it then prun will not attempt to allocate resource itself.

Identifying the Partition

The -p option is used to specify the partition that your program will execute in. If you do not use the -p option a default partition is used (this is specified by your System Administrator), or the partition identified by the RMS_PARTITION environment variable.

```
user@cs2: prun -p parallel myprog
Hello from myprog
Hello from myprog
```

Number of Processes

Use the -n option to control the number of instances of the program. If you do not specify the -n option then your program will be executed on as many processors as are available in the partition, or the number of processors specified by the RMS_NPROCS environment variable. The following example executes 4 processes:

```
user@cs2: prun -n4 myprog
Hello from myprog
Hello from myprog
Hello from myprog
Hello from myprog
```

Pre-allocated Resources

You can use the `allocate` command to allocate resources to a command shell. In this case all instances of `prun` executed by the shell will use the shell's resource; they will not allocate resource themselves.

Warning – Refer to the description of the allocate command on page 31.

The following example allocates 4 processors from the `parallel` partition to a command shell, and then executes `uname` twice on those resources — in both cases `prun` uses the same resources:

```
user@cs2: allocate -pparallel -n4
user@cs2: prun uname -n
cs2-240
cs2-241
cs2-242
cs2-243
user@cs2: prun uname -n
cs2-240
cs2-241
cs2-242
cs2-243
user@cs2: exit
```

Environment Variables

The following environment variables may be used to identify resource requirements to `prun`; these will be used in the absence of conflicting command line options:

Variable	Meaning
`RMS_BASEPROC`	First processor to use in the partition. Numbering starts at 0 with the first processor in the partition.
`RMS_NPROCS`	The number of processors to use. By default this is the largest allocatable number of processors.
`RMS_TIMELIMIT`	Execution timelimit (seconds); the segment will be signalled after the minimum of this time and any system imposed time limit has elapsed.
`RMS_VERBOSE`	Execute in verbose mode (display diagnostic messages).
`RMS_PARTITION`	The name of the partition to use.
`RMS_IMMEDIATE`	Exit if resources not immediately available. By default `prun` is blocked until resources are available.

Environment Information

`prun` (and any other program using `rms_forkexecvp()`) passes all existing environment variables through to the processes that it executes. In addition it adds the `RMS_RESOURCEID`, `RMS_PROCID` and `RMS_NPROCS` environment variables to identify the allocated resource, each process's host, and the total number of processes to the spawned application:

```
user@cs2: prun -pp1-4 sh -c 'echo $RMS_PROCID'
3
2
0
1
    .
```

The information functions provided by Meiko's message passing libraries also allow processes to query their environment. See for example the `cs_get-info()` function in the CSN library.

Standard I/O

Each process in your application has 3 standard I/O streams: `stdin`, `stdout`, and `stderr` (units 5, 6, and 0 in Fortran). No other file descriptors that may be open at the time the parallel application is started will be open in the parallel children. The use of the streams by parallel programs is different to that of sequential programs (i.e. standard Unix applications that execute independently of all other processes).

The remote processes will be started with the standard output and standard error routed to the same place as the host (i.e. `prun`) process. Normal writes to these file descriptors will have the expected effect, as will the `isatty`(3c) function. Other ioctls functions will almost certainly fail.

Parallel Applications

In a parallel program the use of the 3 I/O streams is as follows:

`stdin`	Available to all processes in a parallel program.
`stdout`	Line buffered output from all processes.
`stderr`	Unbuffered output from all processes.

Any process in a parallel program can read from `stdin`, but when using multiple readers it will be necessary to synchronise them with calls to `ew_gsync()`[1], or some other synchronisation function, between each read of the standard input.

1. A function in the Elan Widget library.

In a parallel program many processes may simultaneously print (on `stdout`) but their output will be interwoven on a line-by-line basis with undefined ordering (which may be different each time you run the program). The `-t` option to `prun` tags each line of output with the process id of the outputting process:

```
user@cs2: prun -t pwd
2 /home/user
0 /home/user
3 /home/user
1 /home/user
```

Multiple Sequential Applications

For sequential processes that are executed by `prun` or other host programs the use of the 3 I/O streams is as follows:

`stdin` Available only to the controlling process (note however that `prun` doesn't use stdin).

`stdout` Line buffered output from all processes.

`stderr` Unbuffered output from all processes.

The standard input is not available to sequential processes that are executed with `prun` (repeatable behaviour cannot be guaranteed when unsynchronised processes read at the same time). As an alternative the processes can read from a file; in the following example several instances of `cat` each read from the same input file:

```
user@cs2: prun -pp1-4 sh -c 'cat < myfile'
```

Each process can also read its own file:

```
user@cs2: prun -pp1-4 sh -c 'cat < tmp.$RMS_PROCID'
```

Similarly each process can direct its output to a unique file:

```
user@cs2: prun sh -c 'uname -n > host.$RMS_PROCID'
```

Program Termination

A parallel program exits when all of its processes have exited. The exit status is the global OR of the status from all of its processes. A non-zero exit status will be accompanied by a message if verbose reporting is enabled.

```
user@cs2: prun -v -pp1-4 csn
csn: process 0 (processor 240) exited with status 1
```

If a process is killed then the resource management system runs a cleanup script called `rmscleanup` that attempts to print the reason why the program was killed. Having done this the program is killed, and the program's exit status indicates which signal was used.

```
user@cs2: prun -pp1-4 csn
^C
csn: process 4 killed by signal 2 on cs2-240 (240)
csn: process 3 killed by signal 2 on cs2-241 (241)
csn: process 2 killed by signal 2 on cs2-242 (242)
csn: process 1 killed by signal 2 on cs2-243 (243)
```

If the program was compiled with debugging enabled (`-g` compiler option) then the cleanup script should indicate the line of code being executed at the time the signal was delivered. This may be enough to determine the reason for failure; if it isn't you should run the program under the TotalView debugger (see the TotalView documentation for more information about this), or use `pdebug` as described on page 45.

The program will be killed as soon as one of the `rmscleanup` scripts has completed. If several processes are killed simultaneously you may get partial output from some instances of `rmscleanup`.

Note that the user's path is used to locate `rmscleanup`, allowing a site or user specific script to be substituted where appropriate. The standard release version is in `/opt/MEIKOcs2/bin`. For example if you need output from multiple processes that are killed together then you could write your own clean-up script:

```
#
/opt/MEIKOcs2/bin/rmscleanup
sleep 30
```

Core Files

The delivery of some signals (SIGSEGV, for example) will cause a program to exit and to dump a core file. Core files are generally large, and the I/O implications of all the processes in a parallel application core dumping simultaneously can be severe. Core files are only created if they can be written to temporary storage local to each processor.

Note that core files are removed as resources are freed. If you want to preserve them then you must allocate resource to your shell using `allocate` before using `prun` to execute your application. Under these circumstances `prun` will not allocates resources of its own, and will not free them when it terminates.

Common Problems

This section identifies some common problems and error messages that may be encountered when running your application.

If you get messages other than those described below, or no explanatory message, then please contact Meiko to determine the cause of the problem. You can contact us at the addresses shown inside the front cover of this manual, or by sending email to `support@meiko.com`.

Program hangs

The controlling process of your application (e.g. `prun`) may block if the requested resources are currently unavailable (resource requests made by `allocate` or a hosted program will behave in the same way). You will not be told that it is blocking unless you enable verbose reporting, either by setting prun's `-v` option or by setting the environment variable `RMS_VERBOSE`, in

which case there will be a delay between the "requesting resources" and "resources allocated" messages. If you don't want to wait then use control-C to interrupt. You can use prun's `-i` option to instruct it not to block.

prun: Permission denied
Trying to access a partition that you are not permitted to use. The `permissions/names` files do not identify you as a valid user of the partition that you requested.

prun: Error: exit while controlling process blocked in barrier
The controlling process is waiting for the processes to join it in the start-up barrier but one or more of the processes have exited. Maybe a process terminated before calling its initialisation function.

prog: no such file or directory
The specified program could not be located using your search path. Add the program's directory to your PATH environment variable.

prun: Error: can't determine machine name
You are not running `prun` on a CS-2 processing element.

prun: Error: Partition manager for *partition* is down
The specified partition is unavailable; specify an alternative partition or ask the System Administrator to restart the partition.

prun: Error: no such partition as *name*
The specified partition does not exist; perhaps you typed the name incorrectly.

ld.so.1: *program*: fatal: librms.so.2: can't open file: errno=2
Killed
This error should not occur for `prun`, but may occur with other programs that are linked with `librms` (resource management user interface library). The dynamic linker couldn't locate a library file. Include the directory `/opt/MEIKOcs2/lib` in your LD_LIBRARY_PATH environment variable, or specify a dynamic library search path with the linker's `-R` option.

EW_EXCEPTION @ 2147483647: 6 (Initialisation error)
Can't find own elan capability
Killed
You tried to execute a parallel program without using `prun`.

rmsloader: Error: process *id* **failed on** *node*: **No such file or directory**
rmsloader: Advice: check that *directory* **is mounted**

Check that the specified directory is mounted on all processors that will host your application. Ask your System Administrator to mount the directory on this processor, and to check the other processors that you propose to use.

allocate — Allocate Resources

The partition manager allocates processing resources to user sessions as and when they are requested and become available. The `allocate` command is used when you wish to run a sequence of commands on the same processors, or when you wish to run several tasks concurrently on the same resource.

The usage of `allocate` is as follows:

```
allocate [-n procs] [-p partition] [program [args...]]
```

You can use allocate to reserve resources for the execution of a specified shell script, in which case the resources are allocate to a sub-shell and freed when execution of the script completes, or you can reserve resources indefinitely to an interactive command shell, in which case parallel applications executed by the shell will all run (possibly concurrently) on the shell's resource.

Allocating Resources to a Shell Script

The following example allocates the resources to the shell that will execute the script; calls to `prun` within this script will execute parallel programs on the allocated resource. `prun` only allocates resources itself if they have not been preallocated by `allocate`:

```
user@cs2: allocate -n8 -p parallel script
```

```
#!/bin/csh
prun preprocess
prun iterate
...
prun iterate
prun postprocess
```

Allocating Resources to a Command Shell

If you run `allocate` without program arguments then it spawns an interactive sub-shell with the resources allocated to it. The resources are freed when the sub-shell exits. In the following example both of the `prun` commands execute concurrently on the `parallel` partition:

```
user@cs2: allocate -p parallel
user@cs2: prun myprog &
user@cs2: prun test
...
user@cs2: exit
```

In the next example the two `prun` commands are executed sequentially, one after the other, both on the same processors in the `parallel` partition:

```
user@cs2: allocate -p parallel
user@cs2: prun uname -n
cs2-241
cs2-242
cs2-243
cs2-244
user@cs2: prun uname -n
cs2-241
cs2-242
cs2-243
cs2-244
user@cs2: exit
```

If we were to run the above example without allocating resources to the shell we could not guarantee that the second use of `prun` would start immediately the first completes, and we could not guarantee that both would use the same processors in the partition.

Warning – You have exclusive use of the resource until the shell terminates; the accounting system will be billing you whether you use it or not.

Debugging Parallel Applications

Core files generated by failed parallel applications are deleted from your filesystem when the program's resource is freed. The core files for programs that are executed using the shell's resources will therefore be retained until the shell itself is terminated.

The resources allocated to a command shell may also be shared concurrently by several parallel applications. This means that debuggers (or more generally any parallel program) may be run alongside the processes of a parallel application. See the documentation for the TotalView debugger, or refer to the discussion of `pdebug`.

Confirming Resource Allocation to a Shell

When allocating resource to an interactive command shell you can check that the resource has been successfully allocated by using `rinfo`.

```
user@cs2: rinfo
PARTITION NPROC     PROCS STATUS        TIME
     root   148   128-285   down
    p2-4      4   244-247     up  3:18:58:43
    p1-4      4   240-243     up  1:15:25:52
   p3-16     16   224-239     up  2:23:59:32
   login      8   248-255     up     1:02:18
   p0-96     96   128-223     up  2:21:02:33
  spares      4   256-259     up     9:38:37

RESOURCE NPROC STATUS   USER     GPID    TIME LIMIT
p1-4.572     4 in-use   user 252.1037    0:07 59:53
```

The above example shows that 4 processors on the partition p1-4 have been allocated to `user` for 7 seconds, and that they will remain allocated for, at most, another 59 minutes 53 seconds.

Alternatively, by adding the following commands to your `.cshrc` file (C-shell users) your shell prompt will change whenever you have allocated resources:

```
if ($?RMS_RESOURCEID) then
   set prompt="${RMS_RESOURCEID}: "
else
   set prompt="$user@`uname -n`: "
endif
```

```
user@cs2: allocate -pparallel
parallel.4: exit
user@cs2:
```

Users of the Bourne shell may add the following to their `.profile` file to achieve the same effect:

```
if [ $RMS_RESOURCEID ] then
   PS1="$RMS_RESOURCEID "
else
   PS1="$ "
fi
```

rinfo — Resource Information

`rinfo` displays information about resource usage and availability. Its default output is in three parts showing the configuration, resource availability, and current jobs (note that the latter sections are only displayed if resources/jobs are active).

- The configuration section shows the partitions, their availability, and their uptime.
- The resource section identifies processing resource that is available to you or currently in use by you; for resources that are in-use the time field specifies how long the resource has been held, and the limit field identifies the maximum remaining time that it can be held (the total of the TIME and LIMIT fields is set by the `timelimit` attribute in the `defaults`(4) file).
- The jobs section identifies the resource, command name, and global process id of your applications' controlling processes.

```
user@cs2: rinfo
PARTITION NPROC     PROCS STATUS        TIME
     root   148   128-285   down
    p2-4      4   244-247     up  3:18:58:43
    p1-4      4   240-243     up  1:15:25:52
   p3-16     16   224-239     up  2:23:59:32
   login      8   248-255     up     1:02:18
   p0-96     96   128-223     up  2:21:02:33
  spares      4   256-259     up     9:38:37

RESOURCE NPROC STATUS   USER     GPID     TIME LIMIT
p2-4.648     4   free                     0:56
p1-4.572     4 in-use   user 252.1037     0:07 59:53
login.89     8   free                     7:40
spares.1     4   free                  9:38:38

  USER NPROC  STATUS PARTITION     GPID TIME COMMAND
  user     4 running      p1-4 252.1037 0:08 dping
```

Querying the Resource Usage of Other Users

The `-a` flag allows you to see all resources and jobs (your own and those of other users):

```
user@cs2: rinfo -a
PARTITION NPROC   PROCS STATUS        TIME
     root   148 128-285   down
     p2-4     4 244-247     up  3:18:58:59
     p1-4     4 240-243     up  1:15:26:08
    p3-16    16 224-239     up  2:23:59:48
    login     8 248-255     up     1:02:34
    p0-96    96 128-223     up  2:21:02:49
   spares     4 256-259     up     9:38:53

  RESOURCE NPROC  STATUS    USER       GPID     TIME     LIMIT
 p2-4.648      4    free                        1:12
 p1-4.572      4  in-use    user  252.1037     0:23     59:37
p3-16.106     16  in-use  george  253.4410     2:26   7:44:14
 login.89      8    free                        7:55
   p0-96      96    free                       10:14
 spares.1      4    free                     9:38:54

   USER  NPROC   STATUS  PARTITION      GPID  TIME  COMMAND
   user      4  running       p1-4  252.1037  0:24    dping
georgev     16  running      p3-16  253.4410  2:26    ring6
```

Listing Active Jobs

`rinfo`'s output can be restricted to a list of current jobs by using the `-j` option. You could combine this with the `-a` option to get a list of all jobs (both your own and those of others). In the following example 252.1037 is the global id of the application; it indicates that the application's controlling process has a process id of 1037 on processor 252.

```
user@cs2: rinfo -j
 duncan 252.1037
```

Querying Resource Usage by a Job

To get summary information about a particular job specify its gpid to `rinfo`. The following example shows that the application used 0.4 seconds of user time, 1.5 seconds of system time, an elapsed time of 26 seconds, with an efficiency of 7.3%.

```
user@cs2: rinfo 252.1037
 sleep: 0.4u 1.5s 0:26 7.3%
```

To get more detailed information about a process specify the `-l` option.

```
user@cs2: rinfo -l 108.477
Job: 108.477  Owner: duncan  Command: sleep
Partition: parallel  Procs: 6
Started: Fri Aug 5 13:55 BST 1994  Timelimit: none
Resource Usage        Per Proc      Total
User time                   0.1        0.4   (secs)
System time                 0.3        1.5   (secs)
Idle time                  25.7      154.1   (secs)
Allocated time             26.0      156.0   (secs)
Elapsed time               26.0      156.0   (secs)
Efficiency                  1.2        1.2   (%)
Page faults                   0          0   (#)
Physical memory             775       4654   (KBytes)
Virtual memory              419       2514   (KBytes)
Memory limit                  0          0   (MBytes)
Input/Output                  8         52   (KBytes)
```

Note that some of the above statistics are only available if the resource management accounting system is enabled.

Querying the Configuration

You can get information about the configuration using either the `-c` or `-p` options: `-c` lists the configuration name and the number of partitions, whereas `-p` identifies the partitions (regardless of their availability) and the number of processors in each:

```
user@cs2: rinfo -c
 daytime    7

user@cs2: rinfo -p
    root   148
    p2-4     4
    p1-4     4
   p3-16    16
   login     8
   p0-96    96
  spares     4
```

The `-l` option can be used to get more detailed information:

```
user@cs2: rinfo -pl
PARTITION NPROC   PROCS STATUS        TIME
     root   148 128-285   down
     p2-4     4 244-247     up  3:18:59:59
     p1-4     4 240-243     up  1:15:27:08
    p3-16    16 224-239     up  3:00:00:48
    login     8 248-255     up     1:03:34
    p0-96    96 128-223     up  2:21:03:49
   spares     4 256-259     up     9:39:53
```

Load Balancing

The -H option allows you to identify the least heavily loaded processor in a partition.

```
user@cs2: rinfo -H p2-4
cs2-247
```

Hostname to Processor Id Conversion

The `-t` option converts a hostname to a processor Id (and vice versa):

```
user@cs2: rinfo -t cs2-247      convert hostname to id
247
user@cs2: rinfo -t 247          convert id to hostname
cs2-247
```

reduce — Global Reduction for Shell Scripts

`reduce` is for use in parallel shell scripts. You might use a shell script to perform a sequence of commands on a number of processors. Calls to `reduce` within the shell script can be used to sychronise the scripts, or to reduce, over all the processors, the success of one of the commands.

The following C-shell script can be used to mount a filesystem on a number of processors in a partition. The return status of each process's `mount` command is reduced over all processes and the logical OR returned to each process. If mount command failed on any processor then those that succeeded will unmount the filesystem before exiting (to ensure that the partition is left in a consistent state). The `reduce` command also acts as a barrier, ensuring that all processes wait until all have reached the barrier point.

```
#!/bin/csh

# gmount: a prun shell script that
# mounts a filesystem across a partition
#

set mnttarg = $argv[$#argv]

mount $argv[*]

set lstatus = $status
set gstatus = `reduce -f or $lstatus`

if ($lstatus != 0) then
   echo "Failed mount on `uname -n`"
endif

if ($gstatus == 0) then
   if (`pinfo -i` == 0) then
      echo "Mounted OK"
   endif
else
   umount $mnttarg
endif

exit $gstatus
```

You would execute the above script using `prun`. The following example tries to mount `/opt` on all processors in the partition. Note that this example must be executed by the superuser.

```
user@cs2: prun -p2-4 gmount nova:/opt /opt
```

The `-f` option is used to specify a reduction function, which may be one of:

`sum` The sum of the arguments over all processes.

`or` The logical OR of the arguments over all processes.

`and` The logical AND of the arguments over all processes.

The `-t` option may be used to specify a timeout (specified in seconds).

gkill — Send a Signal

`gkill` is a machine wide version of the Unix `kill`(1) command. The following example sends signal 9 (`SIGKILL`) to two processes: process 163 on processor 0, and process 165 on processor 4.

```
user@cs2: gkill -9 0.163 4.165
```

`gkill` supports the same signals as `kill`(1). See the `signal`(5) manual page for a complete list.

gps — Global Process Status

`gps` is a machine wide version of the Unix `ps(1)`. It produces a process list for each processor (or a subset of processors) in a partition.

Options to `gps` are the same as `ps(1)` with the following exceptions:

`-p` *partition*	Identifies a partition. If no partition is specified `gps` will use the system default, or the partition identified by the environment variable RMS_PARTITION.
`-n` *nproc*	The number of processors to query. By default all processors in the partition are queried, or the number identified by RMS_NPROCS.
`-b` *baseproc*	Identifies the base processor in the partition. Default is 0, the first processor in the partition.

`gps` obtains the process status from each processor by using the resource management system to spawn an instance of `ps` onto each processor. It filters out of the process lists the `ps` command and the loader process that spawned it onto each processor.

If the resource is already in-use by someone else then `gps` will block until it becomes available. If the resource is in-use by you then `gps` will block unless the resource has been allocated to your command shell (with `allocate`), in which case `gps` will run concurrently with any other job that has been started from your shell.

In the following example all the processors in the `p2-4` partition are allocated to an interactive shell. Both `prun` and `gps` are executed concurrently on the allocated resource.

Warning – Refer to the description of the allocate command on page 31.

```
user@cs2: allocate -pp2-4
user@cs2: prun myprog &
user@cs2: gps
------------- processor cs2-240 (240) -------------
   PID TTY          TIME COMD
   256 ?            0:03 myprog
------------- processor cs2-241 (241) -------------
   PID TTY          TIME COMD
   256 ?            0:03 myprog
------------- processor cs2-242 (242) -------------
   PID TTY          TIME COMD
   254 ?            0:03 myprog
------------- processor cs2-243 (243) -------------
   PID TTY          TIME COMD
   255 ?            0:03 myprog
user@cs2: exit
```

copyback — Collect Distributed Files

`copyback` copies one or more files from the local filesystems of the processors in a specified partition. You might use it, for example, to retrieve result files or core files from the temporary filesystem (`/tmp`) of a partition that has just run your application.

The files are collected from their common path on each filesystem and are stored in the specified destination directory. Each file is given a filename extension that identifies the process Id of the process that wrote it.

The following example collects core files from each process and stores them in the directory `corefiles`:

```
user@cs2: copyback -p p1-4 /tmp/core .
user@cs2: ls core*
core.0    core.1   core.2   core.3
```

If no partition is specified then the environment variable `RMS_PARTITION` is used. No partition need be specified if resource is already allocated.

copyout — Distribute Files over a Partition

`copyout` copies one or more files into the local filesystems of the processors in a specified partition. You might use it, for example, to copy a data file into the local temporary filesystems (`/tmp`) of a partition that will run your application.

The following example copies the file `data` from the current working directory into the `/tmp` filesystem of all processors in the `parallel` partition:

```
user@cs2: copyout -p parallel ./data /tmp
```

If no partition is specified the environment variable `RMS_PARTITION` is used. No partition need be specified in resource is already allocated.

Note that `copyout` is much faster than the equivalent use of `prun` and `cp` for large numbers of processors because it uses the Elan's broadcast hardware.

pdebug — Inspect State of Parallel Program or Core Files

`pdebug` produces a backtrace of a program, either by attaching `adb(1)` to the process (if it is still running) or by extracting the information from a core file. Additionally `pdebug` will report on source line-numbers and source files if it was able to locate the debugging information in the program file (the program must have been compiled with the debugging option enabled).

The usage of `pdebug` is:

```
pdebug [program] [object]
```

Where `program` is the name of your program file, and `object` is the pathname of the object file (if it isn't in the same directory as the executable).

To attach to an active job, or to query a process's core file, the job must be executed using resources that have been allocated to a command shell (see the description of the `allocate` command). By allocating resources to a command shell you can run `pdebug` concurrently with your application, and you will also prevent the resource management system from deleting the core files produced when your program terminates (these files are removed when the program's resource is freed, which in this case will happen when the shell itself terminates).

In the following example a sub-shell is allocated the resources in the `p1-4` partition. A parallel program is executed (in the background) on this resource, and `pdebug` is run concurrently with that program. After you have finished with the resource remember to free it by terminating the sub-shell.

Warning – Refer to the description of the allocate command on page 31.

```
user@cs2: allocate -pp1-4
user@cs2: prun csn &
user@cs2: pdebug csn
-------------------- processor cs2-240 (240) ------------------
elan_usecspin() + 10
elan_waitevent(0x0,0x3fdc0,0xffffffff,0x0,0x0,0x8) + 78
ew_tportRxWait(0x3fdc0,0x0,0x0,0x0,0x1,0xe000c040) + 6c
yp_lookup(0xeffffa88,0x3fdc0,0xeffffa98,0xeffff8b8,0x8,0x108)
+94
_csn_lookupname(0xeffffa98,0xeffffa88,0x1,0x1,0x3fca8,0x5) + c
main(0x1,0xeffffb24,0xeffffb2c,0x37000,0x0,0x0) + 14c
   Line 45 in /home/user/csn/csn.c

-------------------- processor cs2-241 (241) ------------------
main(0x1,0xeffffb24,0xeffffb2c,0x37000,0x0,0x0) + 98
   Line 31 in /home/user/csn/csn.c

-------------------- processor cs2-242 (242) ------------------
elan_usecspin() + c
elan_waitevent(0x0,0x3fca8,0xffffffff,0x0,0x0,0x8) + 78
_csn_test(0x0,0x4,0x3fcc0,0x0,0x40578,0x0) + 3b4
main(0x2,0x2,0x2,0x2,0x0,0x5) + 1cc
   Line 61 in /home/user/csn/csn.c

-------------------- processor cs2-243 (243) ------------------
elan_usecspin() + 10
elan_waitevent(0x0,0x3fca8,0xffffffff,0x0,0x0,0x8) + 78
_csn_test(0x0,0x4,0x3fcc0,0x0,0x40578,0x0) + 3b4
main(0x3,0x2,0x2,0x5,0x1,0x5) + 1cc
   Line 61 in /home/user/csn/csn.c

user@cs2: exit
user@cs2:
```

The above backtrace shows that:

- Process 0 reached line 45 and is executing `csn_lookupname()`.
- Process 1 reached line 31 and is executing `main()`. Further analysis of the program code showed this to be the cause of the failure.
- Processes 2 and 3 both reached line 61 and are executing `csn_test()`.

Parallel Programming 5

Parallel Programming Models

There is no single paradigm for programming parallel systems, different approaches suit different types of applications. The most widely used approaches are **Message Passing**, **Data Parallelism** and **Distributed Processing.**

Message Passing

The message passing model was developed for MIMD (Multiple Instruction Multiple Data) machines in which each processor executes its own program. An application is divided into processes which are distributed over the processors. This division can either be by function, different types of process handle different types of task, or by data where different processes are responsible for managing different data items. Each process operates on its own data, and accesses that of others by passing messages. The message passing model is most powerful when an application needs to be performing many different operations at the same time, but can also be used effectively for large numbers of identical processes.

Data Parallelism

In the data parallel model all of the processors simultaneously perform the same operations on different data. Data parallelism was developed on SIMD (Single Instruction Multiple Data) machines, where the hardware constrains you to this approach. Data parallelism is particularly appropriate in scientific and engineering applications where large arrays of data can be spread across the processors.

Distributed Processing

Distributed processing, parallel processing using operating system communications mechanisms is becoming more and more widespread. A distributed application runs as two or more processes on a network of workstations and servers. In the Unix world such processes communicate by message passing using standard protocols.

Choosing which approach to follow depends on the application and is best done by analogy, for example *My problem is the similar to problem A which works well as a data parallel program* or *My problem has the same structure as problem B which runs efficiently using message passing*. Understanding the implementation techniques used for a variety of efficient parallel algorithms assists this process greatly.

CS-2 systems support a wide range of programming models, which can be used on an application by application basis when writing applications from scratch or importing them from other systems. Meiko employs applications consultants, skilled in parallel programming. Please consult us for detailed advice on which approach will best fit a particular problem.

Message Passing Libraries

CS-2 supports a range of message passing interfaces, allowing applications written on first generation Computing Surface systems or other manufacturers parallel machines and workstation networks to be easily ported and run. The interfaces supported are:

- The Meiko CSN library; used widely on first generation Computing Surfaces.
- PARMACS; available on a variety of systems and widely used in Europe.
- PVM (Parallel Virtual Machine); common on workstation networks.
- The Meiko MPSC library; offers source code compatibility with the Intel IPSC and Paragon systems.

All of the above message passing systems are built upon the Elan Widget library, which is in turn built upon the Elan library. The use of a common interface means that programs built on the higher level message passing can use functions in the lower level libraries for performance critical sections of the code. You can also create mixed paradigm applications, allowing for example, a library to be written using one interface and the program calling it another.

Choosing which interface to use will depend upon a variety of factors including the history of the code, performance characteristics and portability criteria.

Data Parallelism

CS-2 can run data parallel programs written in FORTRAN-90 or subset HPF (High Performance FORTRAN). For more details of this approach see the *Portland Group HPF User's Guide* or the *Adaptor User and Reference Manuals.*

Distributed Processing

Unix applications conforming to POSIX system call and Berkely socket interprocess communication standards can be compiled and run on a CS-2 system. See *SunOS Network Interfaces Programmers Guide* (version 5.0 or later) for details.

Message Passing Libraries

All message passing systems are built upon the Elan Widget library. The message passing functions in these libraries use the Elan Widget library's tagged message ports, or TPORTS. TPORTS offer either blocking or non-blocking, buffered or unbuffered, tagged communications between processes. The utilisation of these options is message library specific; some offer all TPORT functionality, whereas others typically employ a subset.

The following sections briefly describe the key features of the most common message passing systems that are available on the CS-2. Each section includes an example program and a reference to more detailed documentation.

CSN

The CSN library is widely used on Meiko's first generation Computing Surface; use this library to port applications from this machine to the CS-2.

Key features of the CSN are:

- Point-to-point communications.
- Message selection at receiver.
- Supports both blocking and non-blocking communications, both unbuffered.
- Hostless applications; require `prun` to execute.
- C and FORTRAN support.

The CSN offers point to point communications via transports, a process's gateway onto the Computing Surface Network. Message selection at the recipient is by sending transport only (and by inference the sending process). Processes may be assigned multiple transports, and may dedicate each to messages of a specific type or to communications among a specified group of processes; a receiving process may in this case infer not only the sending process but also a message type.

Both blocking and non-blocking communications are supported for both the sender and receiver. Blocking functions delay the process until the communication completes (and the data has been received into the recipients data buffer). Note that a blocking communication is an implicit barrier; both sender and receiver must call their communication functions, and neither may proceed until the transfer has completed.

Non-blocking functions initiate a transfer but do not wait for it to complete before returning control to the caller. In this case the data is transferred between the processes at some arbitrary time. Neither the sender's or receiver's data buffers may be modified or freed until test functions confirm that the communication is complete.

More information about the CSN is available from the *CSN Communications Library for Fortran* (Meiko document number S1002–10M107) or the *CSN Communications Library for C* (document S1002–10M106).

Example

The following (simple) example program illustrates summing numbers using the CSN message passing interface — it illustrates use of the message passing routines.

In the first half of the program each process discovers which processor it is running on using `cs_getinfo()`, opens a CSN `transport` and registers its name. It then looks up the name registered by the next processor. `csn_lookupname()` returns the CSN address `next`; knowing this address enables you to send a message to the transport opened on that processor. In this program messages are sent from node `n` to node `n+1` as if the processors were connected in a ring.

```
#include <stdio.h>
#include <csn/csn.h>

main ()
{
   int rx, tx, sum, i, node, nnodes, localid;
   Transport t;
   netid_t next;
   char name[16];

   csn_init ();

   /* find out home many processors there are */
   cs_getinfo(&nnodes, &node, &localid);

   /* open transport */
   csn_open(CSN_NULL_ID, &t);

   /* register transport then lookup that of peer */
   sprintf(name, "TPT_%d", node);
   csn_registername(t, name);
   sprintf(name, "TPT_%d", (node + 1) % nnodes);
   csn_lookupname(&next, name, 1);
```

Having looked up the necessary address the program initialises the variable `rx` with its node number and sets `sum` to zero. A non-blocking receive is started to gather data, and a non-blocking send to send it on round the ring. Non-blocking

message passing is not required in this example, but has been used to illustrate typical usage and to avoid writing code in which half of the processors perform a blocking send and the other half a blocking receive.

```
    rx = node;
    sum = 0;

    for (i = 0; i < nnodes - 1; i++)  {
       tx = rx;

       /* initiate receive */
       csn_rxnb (t, (char *)&rx, sizeof (rx));
       /* initiate transmit */
       csn_txnb (t, 0, next, (char *)&tx, sizeof(tx));

       sum += tx;

       /* block for completion */
       csn_test (t, CSN_RXREADY, -1, NULL, NULL, NULL);
       /* (not interested in envelope info) */
       csn_test (t, CSN_TXREADY, -1, NULL, NULL, NULL);
    }

    sum += rx;
    if (node == 0)
       printf ("Sum = %d\n", sum);
}
```

`csn_test()` is used to test for the completion of each transfer. These operations are repeated until each processor has received all the data and the resulting sum is printed out.

To compile this CSN application for the CS-2 you will need the following flags:

```
user@cs2: cc -c -I/opt/MEIKOcs2/include csn.c
user@cs2: cc -o csn csn.o -L/opt/MEIKOcs2/lib
-lcsn -lew -lelan
```

To run the program use `prun`:

```
user@cs2: prun -pparallel csn
Sum = 6
```

PARMACS

PARMACS is a portable message passing system for use by FORTRAN programmers. Key features of PARMACS are:

- Point-to-point communications with support for a broadcast tree.
- Message selection by sender and user specified message tag.
- Supports both buffered and unbuffered blocking communications.
- Supports hosted applications; programs initiated by a PARMACS host.
- FORTRAN interface.

PARMACS defines both synchronous and asynchronous communications. In the Meiko implementation these map onto unbuffered or buffered blocking communications (respectively) as provided by the Widget library TPORTs.

The synchronous communications functions will block execution of the caller until the message has been transferred directly from the sender to the recipients data vector. The synchronous communications are an implicit barrier, requiring both sender and receiver to call their communications functions before either may proceed.

The asynchronous use the data buffers that are provided by the Widget library TPORTs. The send function transfer's the message from the user's address space into a system buffer and returns control to the caller when this transfer completes, which in most cases will be instantaneously (note also that the sender's data vector may be modified or freed as soon as the send function completes). The receive function transfers the message from the system buffer to the user's data vector and returns control to the caller as soon as the transfer completes.

PARMACS supports only the hosted programming model. Functions are provided that describe to the host program the placement of processes in the machine. This model was developed primarily for systems with limited interconnect in which process placement was critical to program performance. On the CS-2 this functionality is less significant.

More information about PARMACS is available from the *PARMACS 5.1 Release Notes* (Meiko document number S1002–10M118) and from the ANL/GMD PARMACS User's Guide and Reference Manual.

Example

The following trivial example is part of the standard PARMACS release and is included in source form in `/opt/MEIKOcs2/parmacs/example/parmacs/integral`. It defines 3 processes to evaluate:

$$\int_{1}^{0} \sin x dx$$

The host program divides the interval into 3 and uses a synchronous communication to distribute these among the processes. The partial results are additively passed along the 3 node process to the host which displays the result. Note that the node processes are interconnected as a TORUS, however on the CS-2 this corresponds to a simple linear distribution over the available resource (since all processes are fully connected the distribution is unimportant on the CS-2).

The host process is defined as:

```
c
c   Compute the integral over f(x)=sin(x), lower bound x=0,
c   upper bound x=1. The host loads a ring of processes,
c   sends each process some information, collects the results,
c   and prints them.
c
      integer eight, sixtee
c
c   nproc = number of processes in ring
c
      parameter (eight=8, sixtee=16, nproc=3)
      integer procid(0:nproc+1), neigh(2)
```

```
      real*8 result, bnds(2)
      integer*4 typarr(1), lenarr(1)
c
c   declare environment for macros
c
      ENVHOST
c
c   initialize host environment
c
      INITHOST
c
c   map the nproc processes onto a ring structure
c
      TORUS(nproc,1,1,'node','tempfile')
c
c   load the node processes
c
      REMOTE_CREATE('tempfile',procid(1))
c
c   host process is first and last process in ring
c
      procid(0)       = HOSTID
      procid(nproc+1) = HOSTID
      do 10 i=1,nproc
c
c   for each process in ring: send process id's of neighbours
c
        neigh(1) = procid(i-1)
        neigh(2) = procid(i+1)

        typarr(1) = INTEGER_TYPE
        lenarr(1) = 2
        MSG_FORMAT(1,typarr,lenarr)
        SENDR(procid(i),neigh,eight,10)
c
c   for each process in ring: send bounds for integral section
c
        bnds(1) = (i-1)*(1.d0/nproc)
        bnds(2) =  i   *(1.d0/nproc)
        typarr(1) = DOUBLE_TYPE
        lenarr(1) = 2
        MSG_FORMAT(1,typarr,lenarr)
        SENDR(procid(i),bnds,sixtee,10)
10    continue
```

```
c
c   receive the total integral from last node in the ring, and
c   print it to standard output
c
      RECV(result,eight,il,is,it,MATCH_ID(procid(nproc)))
      print *, 'integral =', result
c
c   kill the node processes and clean up host environment
c
      ENDHOST
      end
```

The node processes are defined as:

```
c
c   Compute the integral over f(x)=sin(x), lower bound x=0,
c   upper bound x=1. The node receives two messages from the
c   host: information on neighbours and bounds of section.
c   It computes its part and combines it with those
c   of the other processes. Since the host process is the last
c   in the ring, the global integral is received there.
c
      integer eight, sixtee
      parameter (eight=8, sixtee=16)
      integer neigh(2)
      real*8 result, mypart, bnds(2)
      integer*4 typarr(1), lenarr(1)
c
c   declarations for macros
c
      ENVNODE
c
c   initialize node environment
c
      INITNODE
c
c   receive information on neighbours and bounds from host.
c   Use synchronous communication to ensure the right order
c   of messages (which use the same tag).
c
      RECVR(neigh,eight,il,is,it,MATCH_ID(HOSTID))
      RECVR(bnds,sixtee,il,is,it,MATCH_ID_AND_TYPE(HOSTID,10))
c
c   compute the integral section
c
```

```
      call comput(bnds, mypart)
c
c   sum up the partial results along the ring
c
      if(neigh(1).ne.HOSTID) then
        RECV(result,eight,il,is,it,MATCH_TYPE(20))
        result = result + mypart
      else
        result = mypart
      endif
      typarr(1) = DOUBLE_TYPE
      lenarr(1) = 1
      MSG_FORMAT(1,typarr,lenarr)
      SEND(neigh(2),result,eight,20)

      ENDNODE
      end
```

You can use the supplied makefile to pre-process and compile both the host and node processes:

```
user@cs2: make
```

Alternatively you can pre-process and compile manually by direct use of `parmacs` and your FORTRAN compiler, as shown below. Note that the PARMACS library is sourced from the PARMACS directory tree, whereas the remaining libraries are sourced from the standard Meiko library directory (both paths must be specified after the `-L` option to the FORTRAN compiler driver). Host programs must be linked with the resource management library (`-lrms`) which is a shared dynamic library; this means that the library search path must be specified to the runtime linker via the `-R` option.

```
user@cs2: parmacs -platform meiko -arch cs2-2.1s \
< host.u > host.f

user@cs2: f77 -o host -I/opt/MEIKOcs2/include \
-L/opt/MEIKOcs2/lib \
-L/opt/MEIKOcs2/parmacs/lib/meiko/cs2-2.3s \
-R/opt/MEIKOcs2/lib:/opt/SUNWspro/lib host.f \
-lparmacs -lrms -lew -lelan
```

You pre-process and compile the node program with the following commands. Note that node programs need not be linked with the resource management library.

```
user@cs2: parmacs -platform meiko -arch cs2-2.1s \
< node.u > node.f

user@cs2: f77 -o node -I/opt/MEIKOcs2/include \
-L/opt/MEIKOcs2/lib \
-L/opt/MEIKOcs2/parmacs/lib/meiko/cs2-2.3s \
node.f comput.f -lparmacs -lew -lelan
```

You execute your application by executing the host program. You specify the resource required by the node processes by setting resource management environment variables. In the following example three processors are allocated from the parallel partition. Note that the number of processors allocated must be the same as the number of processes spawned by the host's REMOTE_CREATE macro.

```
user@cs2: setenv RMS_PARTITION parallel
user@cs2: setenv RMS_NPROCS 3
user@cs2: host
 integral =    0.45969769626009
```

PVM

PVM (Parallel Virtual Machine) is widely used to run parallel applications on workstation networks. Key features of PVM are:

- Point-to-point communications with multicast facility.
- Buffer management with typed data packaging functions.
- Message selection at receiver by sender and user specified message tag.
- Non-blocking send; both blocking and non-blocking receive.
- Both hosted and hostless models supported.
- C and FORTRAN interfaces.

PVM supports both point-to-point and multicast communications between message buffers. The creation of the buffers, nomination of one buffer as the active buffer, and the packaging of typed data into the buffers is handled by PVM buffer management functionality.

A non-blocking send function is provided; this uses the buffered blocking functionality of the Widget library TPORTs. The sending function initiates a message transfer, and blocks the calling process until the data has been copied into a system buffer. The function returns to the caller when the copying is complete, thus signalling that the process's own buffer may be modified or freed.

Both blocking and non-blocking receive functions are provided. The non-blocking variant tests the system buffer for a message and returns immediately, whereas the blocking function will delay the calling process until a suitable message becomes available.

PVM supports both hosted and hostless applications. The hostless application requires a loader program, such as `prun`, to allocate resource and to load the PVM processes onto that resource. The hosted application requires that one of the PVM processes (the host) allocates resource and spawns the remaining processes. The host process may also use PVMs communication functions to cooperate with the node processes in the solution of the task.

More information about PVM is available from the document *PVM User's Guide and Reference Manual*, Meiko document number S1002–10M133. The standard PVM release is described by the *PVM 3 User's Guide and Reference Manual*, prepared by the Oak Ridge National Laboratory (reference ORNL/TM-12187).

Example

The following simple FORTRAN program is an illustration of a *master/slave* or *hosted* PVM application. CS2-PVM can also run *spmd* or *hostless* PVM programs; consult the PVM documentation for a definition of hosted/hostless programming models.

The example consists of two programs: a master and a slave. Both master and slave start by calling `pvmfmytid()`, which sets up the CS-2 environment and initialises the TPORTS. The master then calls `pvmfspawn()` to fork a specified number of slaves onto resource that is allocated by the Resource Manager. The ids of the spawned slaves are returned in the `tids` array.

```
      program pvmHost
      include "/opt/MEIKOcs2/include/PVM/fpvm3.h"
c -------------------------------------------------------------
c Example fortran program illustrating the use of PVM 3.0
c -------------------------------------------------------------
      integer i, info, nproc, who
      integer mytid, tids(0:32)
      double precision result(32), data(100)

c     Enroll this program in PVM
      call pvmfmytid( mytid )

c     Initiate nproc instances of slave1 program
      print *,'How many slave programs (1-32)?'
      read *, nproc

c     Start up the tasks
      call pvmfspawn("pvmSlave",PVMARCH,"CS2",nproc,tids,info)
```

Having spawned the slaves, the master initialises the array `data` with 10 integers by first initialising a PVM send buffer with `pvmfinitsend()`. It then packs three integers types: `nproc` (number of slaves), `tids` (the task array) and `n` (the size of data array), into the send buffer. It also packs the data array and then broadcasts the send buffer to the slaves with `pvmfmcast()`. The master then waits to receive the results back from each of the slaves.

```
c     Initialise data array
      do 20 i=1,10
         data(i) = 1
 20   continue

c     broadcast data to all node programs
      call pvmfinitsend( PVMDEFAULT, info )
      call pvmfpack( INTEGER4, nproc, 1, 1, info )
      call pvmfpack( INTEGER4, tids, nproc, 1, info )
      call pvmfpack( INTEGER4, 10, 1, 1, info )
      call pvmfpack( REAL8,    data, 10, 1, info )
      call pvmfmcast( nproc, tids, 1, info )

c     wait for results from nodes
      do 30 i=1,nproc
         call pvmfrecv( -1, 2, info )
         call pvmfunpack( INTEGER4, who, 1, 1, info )
         call pvmfunpack( REAL8, result(who+1), 1, 1, info )
         print *, "I got",result(who+1), "  from", who
 30   continue

      call pvmfexit(info)
      stop
      end
```

The slaves' determine their task ids with `pvmfmytid()` and the task id of their master with `pvmfparent()`. The slaves then receive the tagged message from the receive buffer with `pvmfrecv()` and unpack `nproc`, `tids`, `n` and the `data` array from the receive buffer.

This is followed by a call to the processing routine `work()`. The result of the calculation is then sent back to the master with `pvmfinitsend()` (initialise buffer), `pvmfpack()` (pack data into buffer) and `pvmfsend()` (send buffer). The master as and slaves then call `pvmfexit()` to terminate the program.

```
      program pvmSlave
      include "/opt/MEIKOcs2/include/PVM/fpvm3.h"
c -------------------------------------------------------
c Example fortran program illustrating use of PVM 3.0
c -------------------------------------------------------
      integer  info, mytid, mtid, me, tids(0:32)
      double precision result, data(100), work

c  Enroll this program in PVM
      call pvmfmytid( mytid )
c  Get the master's task id
      call pvmfparent( mtid )

c     Receive data from host
      call pvmfrecv( mtid, 1, info )
      call pvmfunpack( INTEGER4, nproc, 1, 1, info )
      call pvmfunpack( INTEGER4, tids, nproc, 1, info )
      call pvmfunpack( INTEGER4, n, 1, 1, info )
      call pvmfunpack( REAL8,    data, n, 1, info )

c     Determine which slave I am (0 -- nproc-1)
      do 5 i=0, nproc
         if( tids(i) .eq. mytid ) me = i
  5   continue

c     Do calculations with data
      result = work( me, n, data, tids, nproc )

c     Send result to host
      call pvmfinitsend( PVMDEFAULT, info )
      call pvmfpack( INTEGER4, me, 1, 1, info )
      call pvmfpack( REAL8,    result, 1, 1, info )
      call pvmfsend( mtid, 2, info )

      call pvmfexit(info)
      stop
      end
```

Our `work` routine is very simple:

```
      double precision function work(me,n,data,tids,nproc)
      double precision data(*), sum
      integer i, n, me
      integer tids(0:*)

      sum = 1.0
      do 10 i=1,n
         sum = sum + me * data(i)
 10   continue
      work = sum
      return
      end
```

The PVM programs are compiled for the CS-2 as follows:

```
user@cs2: f77 -o pvmHost -R/opt/MEIKOcs2/lib:/opt/SUNWspro/lib
-I/opt/MEIKOcs2/include -L/opt/MEIKOcs2/lib pvmHost.f -lfpvm3 -lpvm3
-lrms -lew -lelan -lsocket -lnsl

user@cs2: f77 -o pvmSlave -R/opt/MEIKOcs2/lib:/opt/SUNWspro/lib
-I/opt/MEIKOcs2/include -L/opt/MEIKOcs2/lib pvmSlave.f -lfpvm3 -lpvm3
-lrms -lew -lelan -lsocket -lnsl
```

You run the program by executing the host process, using Resource Management environment variables to identify the resource. If you use the RMS_NPROCS variable to specify a number of processors then this must be the same as the number of slave processes spawned by the host.

```
user@cs2: setenv RMS_PARTITION parallel
user@cs2: pvmHost
How many slave programs (1-32)? 4
 I got     1.0000000000000   from  0
 I got     11.000000000000   from  1
 I got     21.000000000000   from  2
 I got     31.000000000000   from  3
```

MPSC

This library allows applications to be ported from the Intel IPSC and Paragon systems. It includes both message passing functionality and a suite of global reduction functions.

Key features of this library are:

- Point-to-point communications with multicast facility.
- Message selection at receiver by sender and user specified tag.
- Blocking and non-blocking communications.
- Both hosted and hostless models supported.
- Information functions to determine message size, sender, and tag.
- C and FORTRAN interface.
- Global reduction functions.

Both blocking and non-blocking send functions are provided; the blocking variant returns control to the caller when the data has been copied into a system buffer or user buffer (thus indicating that the sender's buffer can be freed or modified). The non-blocking variant returns immediately and requires test functions to confirm when the sender's buffer may be modified.

Both blocking an non-blocking receive functions are provided. Test functions are provided to test the availability of a suitable message, and information functions can be used to extract envelope information (sender, tag, and message size).

This library supports both hosted and hostless applications. The hostless application is executed with `prun`; the hosted application requires that one process (the host) allocates resource and spawns the remaining processes. The host process may also use MPSC communication functions to cooperate with the node processes in the solution of the task.

The global operation functions take a vector of data from each process in the application and return to each a result vector. The global operation functions operate more efficiently than the equivalent series of communication and calculation

functions. The global operations are an implicit barrier; all the processes must call the same global operation function with the same arguments, and none may begin its calculations until all are ready.

More information about this library is available from the document *Tagged Message Passing and Global Reduction*, Meiko document number S1002–10M108.

Example

The following example program illustrates a hostless application that sums node numbers by message passing and using a global reduction; non-blocking message passing has been used to illustrate more functionality.

Initialisation is simpler in the MPSC library than some other libraries because the MPSC library only supports one communication end-point per processor.

```
#include <stdio.h>
#include <mpsc/mpsc.h>

main ()
{
   int  i, rx, tx, rxDesc, txDesc, sum, node, nnodes;

   mpsc_init ();

   node   = mynode ();
   nnodes = numnodes ();
```

Messages are sent in a ring and the result printed out on processor 0. MPSC is a tagged message passing system, each message is given a tag (0 in our example) when sent. The receiving process selects to receive the first message with a matching tag.

```
   rx  = node;
   sum = 0;

   for (i = 0; i < nnodes - 1; i++)
   {
      tx = rx;

      /* initiate receive */
      rxDesc = irecv (0, &rx, sizeof (rx));
```

```
        /* initiate transmit */
        txDesc = isend (0,&tx,sizeof(tx),(node+1)%nnodes,0);

        sum += tx;

        /* block for completion */
        msgwait (rxDesc);
        msgwait (txDesc);
    }

    sum += rx;

    if (node == 0)
        printf ("Sum = %d\n", sum);
```

The computation is then repeated using a global reduction. The processes barrier synchronise before exiting.

```
    sum = node;
    gisum (&sum, 1, NULL);

    if (node == 0)
        printf ("Sum (via global reduction) = %d\n", sum);

    gsync();
    exit(0);
}
```

To compile an MPSC application for the CS-2 you will need the following flags:

```
user@cs2: cc -c -I/opt/MEIKOcs2/include mpsc.c
user@cs2: cc -o mpsc mpsc.o -L/opt/MEIKOcs2/lib
-lmpsc -lew -lelan
```

To run the program use `prun`:

```
user@cs2: prun -pparallel mpsc
Sum = 6
Sum (via global reduction) = 6
```

Elan Widget Library

The Elan Widget Library is a high performance message passing library that was developed specifically for the CS-2. It provides a number of low level communication constructs that can be used by developers of higher level message passing libraries (CSN, MPSC, PVM, PARMACS etc. are all built upon the Elan Widget library) or for application developers wishing to optimise communications performance.

The relationship between the Widget library and Meiko's higher level libraries means that Widget library functions may be embedded within most CS-2 applications.

Key features of the Elan Widget library are:

- Global memory management.
- Process grouping.
- Access to Elan DMA engine offering high performance network transfers.
- Support for parallel file I/O.
- Channel and broadcast channels.
- Tagged message ports (TPORTs).
- Exception handling.
- Global synchronisation.
- Group reduction and exchange.
- C interface.

The Widget library memory management functions allow regions of memory to be allocated at the same virtual address on a number of processes. This is an important feature of the Elan Widget library (and is used in the implementation of many Widget library communication functions). It allows processes to transfer data directly into the address space of remote processes without prior handshaking of buffer addresses.

A process group defines a number of processes that wish to cooperate in barriers, reductions, and communications. Processes may belong to more than one group, and group definitions may overlap.

Direct access to the Elan DMA engine is provided, allowing either local or remote store-to-store accesses. In many cases data transfer between processes will use one of the higher level message passing facilities: channels or tagged message ports (TPORTs). Channels provide a full duplex, non-blocking, unbuffered communication between a process pair; only one transmit and one receive may be active on a channel at any one time. Broadcast channels are also supported allowing a process to broadcast to a contiguous range of processes using the Elan's broadcast functionality. Tagged message ports provide a more general communication mechanism, offering communication between arbitrary processes using either blocking or non-blocking, buffered or unbuffered, communications, with message selection at the receiver by either user specified message tag or sender id.

Global reduction functions allow data that is distributed among the processes in a group to be combined according to some user supplied function. Global exchange functions provide a mechanism for distributing data among a process group. Both facilities offer significantly higher performance than the equivalent sequence of communications and calculations, especially when the size of transfer is small (and start-up latencies are more significant).

For more information about this library see the *Elan Widget Library*, Meiko document number S1002–10M104.

Example

The following example is a Widget library implementation of the CSN example described earlier. The processes communicate in a ring, each performing a simple addition (essentially a global reduction operation)

The application begins by initailising the base programming environment with a call to `ew_baseInit()`. This initialises both the `ew_state` and `ew_base` structures: the `ew_state` structure identifies each process's virtual process id and the number of processes in the application; the `ew_base` structure identifies the alloc region (global memory) and TPORT (tagged message port) parameters used in this example.

```
#include <sys/types.h>
#include <ew/ew.h>
#include <stdio.h>


main()
{

    int nprocs, me, left, right;
    EW_ALLOC* alloc;
    EW_TPORT* tport;
    ELAN_EVENT *RxEvent, *TxEvent;
    int rx, tx, sum, i;


    /* Intialise base environment */
    ew_baseInit();

    me = ew_state.vp;           /* My process id */
    nprocs = ew_state.nvp-1;    /* Number procs (discount loader program) */
    alloc = ew_base.alloc;

    left = (me+nprocs-1) % nprocs;    /* proc id of left neighbour */
    right = (me+1) % nprocs;          /* proc id of right neighbour */


    /* allocate space for TPORT descriptor */
    if(!(tport = (EW_TPORT*) ew_allocate(alloc, EW_ALIGN,
                                         ew_tportSize(ew_base.tport_nattn))))
    {
        fprintf(stderr, "Failed to allocate\n");
        exit(1);
    }
```

Each process allocates its TPORT structure as a global object by using `ew_allocate()`. Because global objects exist at the same virtual address in all our processes a sending process can target a recipient TPORT without explicit handshaking of addresses.

Note that the TPORT is initialised with the default number of attention slots (currently 4, as defined by `ew_baseInit()`). The attention slots determines the maximum number outstanding communications that may be present on a TPORT at any time. All of Meiko's current message passing libraries are initialised with this default.

Having initialised the TPORTs the processes barrier synchronise with a call to `ew_fgsync()`. The sychronisation point ensures that no process is able to target a TPORT before its initialisation is complete. The synchronisation uses one of the groups defined during the base initialisation; the `segGroup` is a group of all process in the segment (i.e. all processes excluding the loader program).

```
    /* initialise TPORT */
    ew_tportInit(tport, ew_base.tport_nattn, me, ew_base.tport_smallmsg,
                    ew_base.waitType, ew_base.dmaType);


    /* Wait until every process has initialised its TPORT */
    ew_fgsync(ew_base.segGroup);

        printf("Sum = %d\n", sum);
}
```

Each process receives a non-blocking, unbuffered communication from one neighbour, and initiates a non-blocking unbuffered send to its other neighbour (conceptually the processes are connected in a ring). After `nproc` communications each process has calculated the sum of the virtual process id's; process 0 displays the result.

```
    rx = me;
    sum = 0;

    for (i=0; i < nprocs -1; i++) {

        tx = rx;

        /* initiate receive - unbuffered */
        RxEvent = ew_tportRxStart(tport, 0, left, -1, 0, -1,
                                    (caddr_t) &rx, sizeof(rx));

        /* initate transmit - unbuffered */
```

```
        TxEvent = ew_tportTxStart(tport, EW_TPORT_TXSYNC, right, tport,
                                  0, (caddr_t) &tx, sizeof(tx));

        sum += tx;

        /* block for completion - not interested in envelope information */
        ew_tportRxWait(RxEvent, NULL, NULL, NULL);
        ew_tportTxWait(TxEvent);
    }

    sum += rx;

    if (me == 0)
        printf("Sum = %d\n", sum);
}
```

To compile a Widget library application you will need the following flags:

```
user@cs2: cc -o tport -I/opt/MEIKOcs2/include \
-L/opt/MEIKOcs2/lib tport.c -lew -lelan
```

To run the program use `prun`:

```
user@cs2: prun -n4 -pparallel widget
Sum = 6
```

Elan Library

The Elan library provides direct access to the Elan communication processor. The functions in this library provide application programmers with high performance DMA and event handling functionality.

The Elan Widget library (and therefore all of Meiko's current message passing libraries) are built upon the Elan library. You may therefore embed Elan library functions within most CS-2 applications.

Key features of the Elan library are:

- Local or remote DMA.
- Broadcast DMA over contiguous processes.
- Event test and set functionality.
- C interface.

The Elan's DMA engine allows local, remote, and broadcast transfers. Completion of the DMA can be flagged at either (or both) the sender and recipient with Elan events.

The event functionality allows a process to test the state of an event, to set an event in its own address space, and to queue a DMA transfer on an event. Note that the DMA engine can be used to set remote events; a DMA transfer (possibly transferring 0 bytes) may be used to set events at both the sender an receiver.

For more information about this library see *The Elan Library*, Meiko document number S1002–10M131.

Example

The following example is taken from the Elan library documentation. It shows how to embed Elan library DMA and event functionality within a CSN application.

In this example the process initialisation is undertaken by functions in the CSN library, which indirectly call both the Elan Widget library and Elan library initialisation functions. Data buffers are created using `malloc()` and a DMA descriptor is created by `memalign()` (note that the Elan DMA descriptor must be aligned on an EW_ALIGN boundary).

The DMA will transfer data from process 0 into a data buffer somewhere in the address space of process 1. To signify completion of the transfer an Elan event will be set at both the source and destination process. The event structure and data buffer can exist anywhere in the target process's address space, so blocking CSN communication functions are used to notify these to the DMA sender. Note that you could use the Widget library `ew_allocate()` function to define both as global objects, and thus eliminate the CSN handshaking.

```
#include <stdio.h>
#include <sys/types.h>
#include <elan/elan.h>
#include <ew/ew.h>
#include <csn/csn.h>
#include <csn/names.h>

#define DMASIZE 1024

static unsigned char pattern[] = {0x00, 0x00, 0x00, 0x55, 0x55, 0x55,
                                  0xaa, 0xaa, 0xaa, 0xff, 0xff, 0xff};

main()
{
    Transport t;
    netid_t next;
    char* name;
    int me, nproc, i;
    ELAN_DMA *dmaDesc;
    ELAN_EVENT* event;
    unsigned char* buffer;

    /* Package pointers to remote data objects in one structure so we */
    /* can transfer both in one CSN message passing operation. */
    struct {
        unsigned char* bufferp;
        ELAN_EVENT* eventp;
    } rxbuffers;



    /************* CSN library initialisation functions ****************/

    csn_init();

    cs_getinfo(&nproc, &me, &i);    /* i variable not used */

    if(nproc != 2) {
        fprintf(stderr, "error: need 2 processors\n");
        exit(1);
    }
```

```
/* Build structures in processes heap space */
/* DMA descriptor MUST BE 32 bit aligned. */
dmaDesc = (ELAN_DMA*) memalign(EW_ALIGN, sizeof(ELAN_DMA));
buffer = (unsigned char*) malloc(DMASIZE);
event = (ELAN_EVENT*) malloc(sizeof(ELAN_EVENT));

if(csn_open(CSN_NULL_ID, &t) != CSN_OK) {
    fprintf(stderr, "Cannot open transport\n");
    exit(-1);
}

if( me == 0 ) {

    /* Process 0 is DMA sender; receiver of addresses from CSN transport */

    /* Register my transport */
    if(csn_registername(t, "toProc0") != CSN_OK) {
        fprintf(stderr, "Cannot register transport name\n" );
        exit(-1);
    }

    /* Get pointer to remote event and data buffer for process 1 */
    if(csn_rx(t, 0, (char*)&rxbuffers, sizeof(rxbuffers)) <0) {
        fprintf(stderr, "Error on receive of remote addresses\n" );
        exit(-1);
    }
}
else {

    /* Process 1 is DMA receiver; sender of addresses via CSN transport */

    /* Lookup sender's transport */
    if(csn_lookupname(&next, "toProc0", 1) != CSN_OK) {
        fprintf(stderr, "Cannot lookup transport name\n");
        exit(-1);
    }

    /* Send address of my event and data buffers */
    rxbuffers.bufferp = buffer;
    rxbuffers.eventp = event;
    csn_tx(t, 0, next, (char*)&rxbuffers, sizeof(rxbuffers));
}
```

```
/********** Elan library DMA/Event functionality *********/

if(!elan_checkVersion(ELAN_VERSION)) {
    fprintf(stderr, "error: libelan version error\n");
    exit(1);
}
```

Process 0 defines the DMA transfer by initialising a DMA descriptor. This identifies the size of transfer, source and destination buffers, and events that are set on completion at both the sender and receiver. The transfer is initiated by a call to `elan_dma()`, and is tested for completion in both processes by a call to `elan_waitevent()`.

```
ELAN_CLEAREVENT(event);

if(me == 0) {

    /* Processor 0 is the DMA sender */

    /* Initialise sender with data pattern */
    for(i=0; i<DMASIZE; i++)
        buffer[i] = pattern[i % sizeof(pattern)];

    /* Build the DMA descriptor */
    dmaDesc->dma_type = DMA_TYPE(TR_TYPE_BYTE, DMA_NORMAL, 8);
    dmaDesc->dma_size = DMASIZE;
    dmaDesc->dma_source = buffer;
    dmaDesc->dma_dest = rxbuffers.bufferp;          /* Address received from proc 1 */
    dmaDesc->dma_destEvent = rxbuffers.eventp;      /* Address received from proc 1 */
    dmaDesc->dma_destProc = 1;
    dmaDesc->dma_sourceEvent = event;

    /* Initiate DMA; the event signifies completion. */
    printf("Process %d now transfering %d bytes by DMA\n", me, DMASIZE);
    elan_dma(ew_ctx, dmaDesc);
    elan_waitevent(ew_ctx, event, ELAN_POLL_EVENT);
}
else {

    /* Process 1 is the DMA recipient */

    /* Wait for DMA to trigger dest. event */
    elan_waitevent(ew_ctx, event, ELAN_POLL_EVENT);
```

```
        /* Check received data pattern */
        for(i=0; i<DMASIZE; i++)
            if(buffer[i] != pattern[i%sizeof(pattern)]) {
                fprintf(stderr, "Received data differs\n");
                exit(1);
        }
        printf("Data received and verified by process %d\n", me);
    }
```

To compile this application you must link with the CSN library, the Elan Widget library, and the Elan library, as shown below:

```
user@cs2: cc -o csnDMA -I/opt/MEIKOcs2/include \
-L/opt/MEIKOcs2/lib csnDMA.c -lcsn -lew -lelan
```

You run the program on two processors with `prun`:

```
user@cs2: prun -n2 -pparallel csnDMA
Process 0 now transfering 1024 bytes by DMA
Data received and verified by process 1
```

Glossary *A*

Configuration
The set of partitions that make up a CS-2 system.

Node
See Processing Element.

Pandora
The graphical user interface (GUI) to the resource management system. Pandora uses the facilities of a colour X workstation to display the status of a CS-2 system, to query its usage, and to manipulate its partitions. The user interface is via the familiar X-windows point-and-click system.

PFS
The Meiko parallel filesystem. Allows files to be striped over a number of Unix filesystems. Allows very large files to be created (as large as the total capacity of all the participating Unix filesystems) and provides higher performance data access for parallel programs, which need not compete for access to a single disk device when accessing data.

Partition
A set of processing elements dedicated to performing certain classes of work. Most systems have at lease 2 partitions, one for interactive tasks and one for parallel applications. Additional partitions can be added to support system admin tasks, device management and batch processing. Partitions are set up by the system manager.

Processing Element

A CS-2 system made up of multiple processing elements (or PEs). Each is a SPARC processor, a memory system and an interface to the CS-2 inter-processor communication system. Some processing elements have additional vector some control an I/O system.

Vector Processing Unit

CS-2 uses a Fujitsu vector processing unit to provide high floating point performance on certain classes of application. Each vector PEs has a pair of these processors sharing memory with the SPARC. The compiler assigns vectorisable blocks of code to these processors and scalar code to the SPARC.

Hosted vs. Hostless

Some message passing libraries support both hosted and hostless models, others are limited to just one.

Hostless applications consist of two processes; a host and a number of identical slave processes. The application is initiated by executing the host process which then spawns the slave processes into a partition. All processes, including the host itself, use message passing functions to cooperate and complete the task. Both PVM and the MPSC library may support this model.

Hostless applications have a number of identical slave processes that are spawned onto a partition `prun`. The PVM, MPSC, and CSN libraries support this model.

In either model the host/loader program uses functions in the resource management user interface library to liaise with the Resource Manager for the slave's processing resource. The host/loader executes in one segment (typically in your login partition) and the slaves execute in a second segment within some other partition.

Elan Id

The decimal representation of a node's unique route from the top of the network. For example, the node uniquely identified by the route <5>.<1> will have Elan Id 21 (hint: convert the route to binary 101.01 and convert this to decimal). See the Network Overview (document S1002–10M105) for a description of network routes.

Computing Surface

CS-2 System Administration Guide

S1002–10M126.01

meiko

Meiko's address in the US is:

Meiko
130 Baker Avenue
Concord MA01742

508 371 0088
Fax: 508 371 7516

Meiko's address in the UK is:

Meiko Limited
650 Aztec West
Bristol
BS12 4SD

01454 616171
Fax: 01454 618188

Issue Status:

Draft	
Preliminary	
Release	x
Obsolete	

Circulation Control: *External.*

Typographic Conventions *i*

The following typographic conventions are used in this document and all other Meiko documentation:

Library reference.

Italicised text is used for references to other documents, or emphasised expressions that may be expanded later in the text. Also used in example command lines in place of site specific options.

> See the document *Getting Started — Users Guide.*

> `Password:` *password*

Do not copy this.

Emboldened text is used to emphasise expressions of particular importance.

> It is important that you **do not try this yourself.**

`integer name`

Courier is used for variable names, command names, filenames, and other text that might be entered into the computer system, or for the computer's response to a user's request. See also the use of bold courier below.

> The function `rms_describe()` provides a description of a configuration.

cat file

Bold courier is used when illustrating a dialogue between the computer and a user. Text entered by the user is shown in this font, text displayed by the machine is shown in courier.

```
user@CS2-1 ls /opt/MEIKOcs2
bin/ docs/ example/ include/ lib/ man/
```

Warning – Used to draw the reader's attention to an important note.

Contents

System Overview 1

Introduction

The CS-2 system is a tightly coupled network of SPARC processors running the Solaris operating system. Processors are interconnected by a Meiko designed switch network optimised for high performance inter-processor communications, rather than a conventional LAN.

Many of the administration tasks that must be performed on a CS-2 system are the same as for a SPARC workstation network. However the Solaris operating system has been extended by Meiko in several key areas:

- Resource management.
- Parallel filesystem implemented as a Solaris VFS.
- Interprocessor communications device drivers.
- Support for diagnostic network.

This document describes the additional steps necessary to setup the software and the tools for administration of tasks across large numbers of processors. Related documents are the *Getting Started — Users Guide* and the *Pandora Users Guide*.

1

Resource Management System

The role of the resource management system is to provide a System Administrator with the ability to optimise the use of the resources in a particular CS-2 system. It does this by controlling user requests to login and to run parallel applications, by controlling access, accounting usage, and by collecting performance data.

CS-2 systems are managed as a small number of partitions each containing many, identical, processors. The system management tools are designed to perform identical operations on all processors in a partition, rather than treat each individually. Partitions are created to dedicate resources (processors and their I/O devices) to specific tasks.

Example partitions might include a *login* partition for interactive development, a device management partition servicing disk arrays, and a *parallel* partition for dedicated running of completed applications.

Access to partitions can be restricted to groups of users, and scheduling schemes can be imposed to further control their use. The collection of partitions within the system is called a *configuration*. The System Administrator will typically define configurations relating to the varying workloads that will be imposed on the machine (configurations for day and night-time use are common requirements). Chapter 2 describes the configuration utilities and resource management system daemons.

Parallel Filesystem

The Meiko parallel filesystem allows files to be striped over many data storage devices. Files that are written to a parallel filesystem are written in stripes of a specified size which are distributed over the underlying filesystems (supported base filesystem types are Solaris UFS and NFS).

Striped filesystems offer two significant advantages. Partitions and hence file sizes can be large; each underlying filesystem will hold just a small part of the whole file, and the number of underlying filesystems is unrestricted. Distributed files also offer higher file access bandwidths for parallel applications; processes need not compete for access to a single IO device when accessing their data.

The stripe size and number of underlying filesystems can be dictated by the applications that will use the parallel filesystem. Chapter 3 describes how to setup a parallel filesystem.

Device Drivers

The Elan communication processor provides the interface between each SPARC processor and the data network that joins them together. Solaris device drivers for the communications processor include the following — these drivers are installed as part of the system setup:

`elan` Elan device driver.

`elanip` IP driver for internal data network.

Terminal connections between processors, as used by the login load balancer when connecting a login gateway processor to a user login partition, are routed via a kernel device driver, `shtcirc`.

Diagnostic Network

The Control Area Network (CAN) diagnostic network runs throughout the machine with interfaces to all processing elements. It monitors the health of modules, processors, and peripherals, reporting errors to the resource management system. Tools are provided that allow the System Administrator to use the control network to query the status of the CS-2 hardware, and to initiate module reset or shutdown under software control. Access to the control network is via the `can` device driver, and is normally restricted to root. The utility `cancon` allows privileged users to connect consoles to processors over the control network.

1

Resource Management 2

Commands and Daemons

The resource management system includes three daemons: the machine manager `mmanager`, the partition manager `pmanager`, and the accounting daemon `acctd`. In addition there are utilities to control the configuration, to run parallel programs, and to support administration tasks: `rcontrol` edits configurations, `prun` executes parallel programs, `rinfo` provides utilisation information, `minfo` provides hardware information, `gkill` sends a signal to processes, and `reduce` and `pinfo` support parallel shell programming.

The Machine Manager

The machine manager `mmanager` manages the machine description database and monitors the health of the system. It services requests for information about the hardware and its status. The machine manager is normally started on the operating system server as the system boots (see `/etc/rc2.d/S92mmanager`). It can be run interactively as root, to do this type:

```
root@cs2-0: mmanager -ir1
```

The option `-r` takes a reporting level mask; level 1 enables initialisation debug messages, level 2 enables reporting of each request as it is processed by the server. The `-i` option indicates interactive operation. The `-f` option forces the replacement of an existing `mmanager`.

As the machine manager starts it displays the machine configuration and system defaults:

```
Parsing machine description file /opt/MEIKOcs2/etc/cs2-/machine.des
Network has 3 level(s)
Space for up to 32 modules 128 boards 128 processors 160 switches
Machine has 1 bay
Initialising CS-2 Machine Manager on cs2-
System defaults: ROM(95) H8-ROM(93090611) Broadcast(on) CAN(on)
Partition(parallel) Load statistic(load)
Accounting(off) Halt on error(off)
```

ROM revisions and other diagnostic and configuration information for modules, boards, and processors are displayed during the machine manager's start-up, as shown below. Note the warning messages ("`can't probe module`") which indicate that modules 2 and 3 have been powered down:

```
mmanager: module controller board (m=7,controller) H8 ROM revision 93102018
mmanager: 1x16 switch board (m=7,b=0) H8 ROM revision 93102018
mmanager: module controller board (m=0,controller) H8 ROM revision 93102018
mmanager: Vector board (m=0,b=0) H8 ROM revision 93102912
mmanager: Vector board (m=0,b=1) H8 ROM revision 93102912
mmanager: Vector board (m=0,b=2) H8 ROM revision 93102912
mmanager: Vector board (m=0,b=3) H8 ROM revision 93102912
mmanager: processor cs2-0 (m=0,b=0,p=0) ROM revision 118
mmanager: processor cs2-1 (m=0,b=1,p=0) ROM revision 118
mmanager: processor cs2-2 (m=0,b=2,p=0) ROM revision 118
mmanager: processor cs2-3 (m=0,b=3,p=0) ROM revision 118
mmanager: module controller board (m=1,controller) H8 ROM revision 93102018
mmanager: Vector board (m=1,b=0) H8 ROM revision 93102912
mmanager: Vector board (m=1,b=1) H8 ROM revision 93102912
mmanager: Vector board (m=1,b=2) H8 ROM revision 93102912
...
mmanager: Warning: can't probe module 2
mmanager: Warning: can't probe module 3
...
mmanager: module controller board (m=4,controller) H8 ROM revision 93102018
mmanager: Dino board (m=4,b=0) H8 ROM revision 93102912
mmanager: Dino board (m=4,b=3) H8 ROM revision 93102912
mmanager: processor cs2-16 (m=4,b=0,p=0) ROM revision 115
```

The initial status report identifies the processors, their types, and their status:

```
  0: CAN 000000 Vector Viking+Cache 64MB status: ROM running
  1: CAN 000004 Vector Viking+Cache 64MB status: ROM running
  2: CAN 000008 Vector Viking+Cache 64MB status: ROM running
  3: CAN 00000c Vector Viking+Cache 64MB status: ROM running
  4: CAN 000100 Vector Viking+Cache 64MB status: ROM running
  5: CAN 000104 Vector Viking+Cache 64MB status: ROM running
  6: CAN 000108 Vector Viking+Cache 64MB status: ROM running
  7: CAN 00010c Vector Viking+Cache 64MB status: ROM running
  8: CAN 000200 Vector Viking+VPU  128MB status: Unix level 3
  9: CAN 000204 Vector Viking+VPU  128MB status: Unix level 3
 10: CAN 000208 Vector Viking+VPU  128MB status: Unix level 3
 11: CAN 00020c Vector Viking+VPU  128MB status: Unix level 3
 12: CAN 000300 Vector Viking+VPU  128MB status: Unix level 3
 13: CAN 000304 Vector Viking+VPU  128MB status: Unix level 3
 14: CAN 000308 Vector Viking+VPU  128MB status: Unix level 3
 16: CAN 000400 Dino Viking+Cache   64MB status: Unix level 3
 19: CAN 00040c Dino Viking+Cache   64MB status: Unix level 3
```

Diagnostics

Diagnostic messages from the machine manager are written to the file `/var/adm/mmanager.log`; if the machine manager is run interactively the messages are directed to the standard output device. The machine manager's `-l` option allows you to alter the file used for diagnostic messages.

Data Files

The machine manager reads the system configuration from the `machine.des` file in the directory `/opt/MEIKOcs2/etc/`*machine-name* — this file identifies the hardware resources and their placement within your machine. The system defaults are read from the `defaults` file (also in `/opt/MEIKOcs2/etc/`*machine-name*).

The format of the machine description file is described in *Machine Description File* on page 33, and the defaults file is described in *Defaults File* on page 36.

The Partition Manager

The partition manager `pmanager` runs on every processor in the system. It services system resource allocation requests, controls access, runs parallel programs, and services requests for information about machine load and the state of queued and running jobs.

Partition managers are started and stopped from `pandora` or `rcontrol`, either for the whole configuration or one partition at a time. The partition manager requires support from the `inetd`, and hence requires an entry in the `/etc/services` file:

```
pmanager        800/tcp     # CS-2 Partition Manager
```

Note the use of a secure socket, the `pmanager` can only be started by a root process. An entry in the `/etc/inetd.conf` file is also required:

```
pmanager  stream  tcp  nowait  root  /opt/MEIKOcs2/bin/pmanager pmanager
```

The partition manager can be run interactively; this is useful while setting a system up for the first time:

```
root@cs2-0: prun -p partition start_pmanager -irl1
```

The option `-r` takes a reporting level mask; level 1 enables initialisation debug messages, level 2 enables reporting of each request as it is processed by the server, level 8 enables reporting of job status. The `-f` option forces the removal of an existing pmanager.

Diagnostics

Diagnostic messages from the partition manager are written to the file `/var/adm/pmanager.log` on the first processor in the partition; if it is run interactively the messages are directed to the standard output device. The `-l` option allows you to alter the file used to log diagnostic messages.

Data Files

Access to a partition is controlled by the partition manager using data from the `permissions` and `names` files.

Note that the access control system is enabled by default. Users other than root will not be able to use the system until you create entries for them in the `permissions` file, or disable it by setting `access-control=0` in the `defaults` file.

The `permissions` and the `names` files are described in *Access Control* on page 23. The `defaults` file is described in *Defaults File* on page 36.

Configuration Editor — rcontrol

The utility `rcontrol` allows the System Administrator to set up configurations, create and delete partitions, and assign processors to partitions. Configurations may also be defined and edited with Pandora, the resource management GUI. See the *Pandora Users' Guide* for more information about this tool.

Configuration definitions are held in the filesystem under `/opt/MEIKOcs2/etc`. This directory and those beneath it must be generally readable, but owned and only writeable by `root`. This is usually achieved by a `share` entry in the file `/etc/dfs/dfstab`.

`rcontrol` can be run interactively or as a sequence of commands, its usage is as follows:

```
Usage: rcontrol [-e] [-v] [command args...]
Try 'rcontrol help' for more info
```

To run `rcontrol` interactively type `rcontrol`; it will return with a prompt. You can now run the configuration manipulation commands; `help` is a good place to start — it will show you the commands available:

```
root@cs2-0: rcontrol
rcontrol: help

Usage: help [all | command]

Commands

   help   [all | command]
   add    -c name | -p name | -r <range>
   get    -c name | -p name
   remove -c [name] | -p [name] | -r <range>
   save   [-c name]
   start  [-k] [-i] [-r level] [-p name] name
   stop   [-k] [-p name]
   set    [-d level] [-v]
   show   [-c [name]] [-p [name]]
   exit
```

To get more information about a particular command type `help` followed by the command — `help all` will tell you about all of them. Common command options include `-c` for configurations, `-p` for partitions, and `-r` for processor ranges.

Creating a Configuration

To create a new configuration use rcontrol's `add` command. All configurations initially contain just one partition called `root`. The `root` partition contains all processors in the system; access to it is restricted to super-users.

The following example uses the `add` command to created a new configuration called `day`. The `show` command is used to display its initial partitions:

```
rcontrol: add -c day
rcontrol: show -c
Configuration day:
Partition: root       processors : 0-64 68 72
```

To add partitions to the configuration you use the `add` command with its `-p` option. To add processors to a partition use `add` with the `-r` flag and a processor range. Processors are always added to the working partition:

```
rcontrol: add -p login
rcontrol: add -r 64-72
rcontrol: show -p
Partition: login      processors : 64 68 72
rcontrol: add -p parallel
rcontrol: add -r 0-63
rcontrol: show
Partition: parallel    processors : 0-63
Partition: login       processors : 64 68 72
Partition: root        processors : 0-64 68 72
```

Saving a Configuration to Disk

Once a configuration is complete use `save` to write it to disk. Configurations are stored in the directory `/opt/MEIKOcs2/etc/`*machine-name*`/`*config-name*. The *config-name* in the following example is `day`.

```
rcontrol: save
Confirm save of configuration <day> (y/n): y
```

Restoring a Configuration from Disk

To restore an existing configuration use `get` with the `-c` flag; to select a partition from a configuration use `get` with the `-p` flag. You can use the `add` command to add additional processors to the partitions, or the `remove` command to remove processors.

```
rcontrol: get -c day
rcontrol: get -p login
rcontrol: show
Configuration day:
Partition: parallel   processors : 0-63
Partition: login      processors : 64 68 72
Partition: root       processors : 0-63 68 72
```

Starting and Stopping Partitions

The `start` and `stop` commands instruct the resource management system to start and stop individual partitions or the whole configuration.

To start the `login` partition type:

```
rcontrol: start -p login
```

To stop the `login` partition type:

```
rcontrol: stop -p login
```

The `-k` option allows you to determine whether jobs running on a partition are killed or given time to exit. Note that killed jobs are sent a SIGTERM and given time to exit gracefully before being terminated forcefully.

When a configuration is not specified the working configuration is used, if it is not defined then the `active` configuration is assumed. Hence to start the `active` configuration type:

```
rcontrol: start
```

To shut down the `active` configuration, type:

```
rcontrol: stop
```

The `start` command's `-i` option starts a partition interactively, logging output to the screen. Note that this operation does not terminate until the partition is closed. Without this option the partition manager will log to a file. The `-r` option allows you to set the partition manager's reporting level.

You can replace the current active configuration with an alternative configuration by using the `start` command. When you switch to a new configuration the `start` command usually waits for processes running in the old configuration to complete before the new configuration is used. By specifying the `-k` option the

existing processes are killed. In the following example the configuration `day` will become the active partition. Switching configuration causes a symbolic link to be created in the `/opt/MEIKOcs2/etc/`*machine-name* directory between `active` and the configuration definition. Changes to the configuration cause the partition managers to restart automatically.

```
rcontrol: start -k day
Confirm switch to configuration day (y/n): y
Confirm kill of existing processes (y/n): y
```

A minimal set of partitions is started; only those changed in moving from one configuration to another.

Quitting rcontrol

To leave `rcontrol` type `exit`; end-of-file (`^D`) has the same effect.

```
rcontrol: exit
root@cs2-0
```

Executing Parallel Programs — prun

`prun` executes a parallel SPMD program on the CS-2. It negotiates with a partition manager for a set of processors and then starts an executable image on each of the processors, passing the remainder of the command line to all of them as their argument list.

```
user@cs2-0: prun myprog
Hello from myprog
Hello from myprog
```

`prun` is generally used to run parallel application programs but can also be used by System Administrators and Operators to execute administrative commands on a group of processors.

Partitions

The `-p` option is used to specify the partition that your program will execute in. If you do not use the `-p` option a default partition is used; this is specified in the `defaults` file (see *Defaults File* on page 36,).

```
user@cs2-0: prun -s -p parallel uname -n
cs2-0
cs2-1
```

Note the use of the `-s` option as `uname` is a regular SPARC binary and not a parallel program (see below).

Number of Processors

Use the `-n` option to control the number of instances of the program. The following example executes 4 processes, each on a separate processor in the default partition:

```
user@cs2-0: prun -n4 myprog
Hello from myprog
Hello from myprog
Hello from myprog
Hello from myprog
```

Standard I/O under prun

Programs linked with parallel libraries direct all their standard I/O to the system call server in the controlling process (`prun` in this case). Sequential programs will not normally have been linked in this way and should be run with the `-s` flag.

In the following example the Unix command `uname` is executed on 3 processors in the default partition. Output is displayed processor by processor.

```
user@cs2-0: prun -sn3 uname -n
cs2-0
cs2-1
cs2-2
```

If the -v flag is used then the output will be interleaved with processor identification banners:

```
user@cs2-0: prun -vs uname -n
------------------- processor 0 ------------------
cs2-0
------------------- processor 1 ------------------
cs2-1
```

The -s flag causes output from each process to be written to temporary files in the current directory; these files are copied to the user's terminal and deleted as prun exits, they can be preserved by setting the -l flag.

```
user@cs2-0: prun -ls pwd
/home/phobos/user/cs2parts/rmanager
/home/phobos/user/cs2parts/rmanager
user@cs2-0: ls stdout.* stderr.*
stdout.335.0
stdout.335.1
```

Warning – Note that the -s option requires you to have write permission to the current directory.

Common Problems

prun returns a non-zero exit status and/or prints error messages on stderr. Possible causes of error include:

- Requesting more processors than are available.
  The partition isn't large enough to run your program:

```
prun: Error: rms_forkexecvp failed:
Insufficient resources to satisfy request.
```

  Note that `rms_forkexecvp()` is the library routine that starts parallel programs.

- Your home directory is not mounted on all processors.

```
prun: Error: forkexec failed: System error.
Consult System Manager: No such file or directory
```

- You do not have write permission in the current directory.

```
prun: Error: forkexec failed:
No write permission to working directory
```

- You are not on a CS-2 processing element.
  `prun` returns the following error if executed on a processing element that is not connected to the CS-2 network:

```
EW_EXCEPTION @ ----: 6 (Initialisation error)
 Failed elan_init() 11: No more processes
Killed
```

- If the partition is down then `prun` will fail for users other than root with:

```
prun: User access to comms hardware disabled.
```

or:

```
prun: Error: Partition manager for parallel is down.
```

If `prun` is called by root then it will spawn processes using `rsh` if the partition is down. The `-r` option enforces this behaviour.

Resource Information — rinfo

`rinfo` displays information about resource availability and usage.

Querying your Own Jobs

The `-j` option causes `rinfo` to display information about your own jobs:

```
user@cs2-0: prun -n2 myprog &
user@cs2-0: rinfo -j
user 0.694  running myprog
```

0.694 is the global id of the parallel process; it indicates that the `prun` command is process 694 on processor 0.

More information about your own jobs is available by specifying both the `-j` and the `-l` (long) options. The following example shows that the program is running on 2 processors in the `parallel` partition and that it has been running for 2 minutes 39 seconds.

```
user@cs2-0: prun -n2 myprog &
user@cs2-0: rinfo -jl
USER NPROC STATUS  RESOURCE GPID  TIME     COMMAND
user     2 running parallel 0.694 00:02:39 myprog
```

Querying Everyone's Jobs

Use the `-ja` option to get information about all processes running on the system:

```
user@cs2-0: rinfo -jal
USER NPROC STATUS  RESOURCE GPID TIME     COMMAND
user     2 running parallel 0.50 01:58:39 myprog
jim      2 running parallel 0.60 00:58:23 xmlt1
```

Querying Resource Availability

Two options allow you to query the resources that are available. The `-c` option describes the current configuration, and the `-p` option describes the partitions available listing the partition names and the number of processors in each.

The following example command queries the configuration and partitions that are available to all users: it shows that the active configuration (`day`) consists of three partitions, `root`, `login` and `parallel`:

```
user@cs2-0: rinfo -cpal
CONFIG NPART
day     3

PARTITION NPROC STATUS NJOB NTARG
root         67  down
login         3    up
parallel     64    up     1    3
```

In addition the `-H` option may be used to identify the least loaded processor in a partition. In the following example processor 5 is the least loaded in the `parallel` partition:

```
user@cs2-0: rinfo -H parallel
cs2-5
```

Hostname to Elan (Processor) Id Translation

Use the -t to map from a given hostname to Elan Id, or vice versa:

```
user@cs2-0: rinfo -tl 3
PROC HOSTNAME
   3 cs2-3
user@cs2-0: rinfo -tl cs2-4
PROC HOSTNAME
   4 cs2-4
```

Signalling Processes — gkill

`gkill` sends a signal to a process or process group in a similar manner to the Unix `kill` command. The following example sends signal 9 (`SIGKILL`) to two processes that are identified by their process id's. These processes must be running on the same processor as the `gkill` command itself:

```
user@cs2-0: gkill -9 16301 16302
```

`gkill` can also be used to send a signal to a parallel application. In this case the global pid is specified (this can be obtained from `rinfo`, for example):

```
user@cs2-0: rinfo -jl
USER NPROC STATUS  RESOURCE GPID  TIME     COMMAND
user     2 running parallel 0.694 00:02:39 myprog
user@cs2-0: gkill -KILL 0.694
```

The list of supported signals can be obtained by using `gkill`'s `-l` option:

```
user@cs2-0: gkill -l
HUP INT QUIT ILL TRAP ABRT EMT FPE KILL BUS SEGV
SYS PIPE ALRM TERM USR1 USR2 CHLD LOST WINCH URG IO
STOP TSTP CONT TTIN TTOU VTALRM PROF XCPU XFSZ
WAITING LWP
```

A list of signals is also presented in the `signal`(5) manual page:

```
user@cs2-0: man -s 5 signal
```

Hardware Information — minfo

`minfo` will list machine information for the processor modules in your system. For each of the processor boards it displays the board type, serial number, board revision, Boot ROM and H8 ROM revisions, Ethernet address, total memory, and processor type.

`minfo` interrogates the CAN bus and must therefore be executed by root. It also requires that the machine manager is running.

```
root@cs2-0: minfo
::::::::::::::::::::::::::::::::::::::::::::::::::
Minfo attached to cluster: 0 module: 0 node: 0

**************************************************
Cluster: 0, Module: 0, Board: 0
Board Type: Dino (MK401)
Serial Number: 03797486
Board revision: D (4 in ROM)
Hardware mod status: 0 - invalid value
OBP Revision: 123
H8 Rom Revision: 13:00 6/12/93
EtherNet Address: 8b:80:65
Memory : 64 Megabytes
Processor type: 4 = Viking E-cache
::::::::::::::::::::::::::::::::::::::::::::::::::
```

You can also use `minfo`'s `-c` option to check the machine configuration, ensuring that the processors and switches are correctly configured for the network.

Shell Program Information — pinfo

`pinfo` provides information to parallel shell scripts. Compiled programs can use the facilities of their preferred parallel programming library to learn about the environment that they are being executed in. When a parallel program is implemented as a shell script `pinfo` allows each instance of the shell script to determine the id of it's hosting processor, and the total number of processes in the application. `pinfo` recognises the following command line options:

- `-e` Display physical processor id (Elan Id)
- `-i` Display logical processor number within application.
- `-l` Display physical id of the first processor in this application.
- `-n` Display the number of processors used by this application.

Shell Script Global Reduction — reduce

`reduce` is for use by parallel shell scripts. `reduce` synchronises with all other processes in the application and then performs a reduction function on its arguments. The results are printed to stdout. For example:

```
result = `reduce -f or $status`
```

The above line in a parallel program C-shell script will cause all the processes to synchronise, each will be returned the global OR of the local `status` variables.

Functions that are supported by the `reduce` command are:

`sum` The sum of the arguments over all processes.

`or` The logical OR of the arguments over all processes.

`and` The logical AND of the arguments over all processes.

The following shell script mounts a filesystem on all processors in a partition. Each instance of the script mounts the specified filesystem and then performs a global OR of the return status. If any one process failed to mount the filesystem then it is unmounted from all those that succeeded to ensure that the partition is left in a consistent state.

The shell script can be executed on all processes in the `parallel` partition by using the following `prun` command line:

```
user@cs2-0 prun -p parallel gmount \df ~
deimos:/export/home/deimos /export/home/deimos
```

```
#!/bin/csh

# gmount: a prun shell script that
# mounts a filesystem across a partition
#

# Get command line arguments for the mount command.
set mnttarg = $argv[$#argv]

# Mount the filesystem.
mount $argv[*]

# Check the mount status across all processes;
# global OR of each mount exit status.
set lstatus = $status
set gstatus = `reduce -t 30 -f or $lstatus`

# Write error message identifying processes that failed.
if ($lstatus != 0) then
 echo "Failed mount on `uname -n`"
endif

# If any process failed unmount from those that succeeded.
if ($gstatus == 0) then
 if (`pinfo -i` == 0) then
 echo "Mounted OK"
 endif
else
 umount $mnttarg
endif

exit $gstatus
```

Access Control

User access to each partition is controlled with the `permissions` and `names` files. The `permissions` file specifies users or lists of users that can access a partition; the `names` file assigns a name to a list of users.

The `permissions` file contains configuration specific partition names and is stored alongside the configuration's definition in `/opt/MEIKOcs2/etc/`*machine-name/config-name*. The names file is stored in `/opt/MEIKOcs2/etc`.

Access control is enabled by setting the variable `access-control` within the `defaults` file. The defaults file is described in *Defaults File* on page 36.

Permissions File

The `permissions` file contains one entry for each partition. It can include definitions for partitions that are not in the current active configuration, but these will be ignored. Each entry consists of a partition name, a list of permitted users, and a list of processors that can source requests to the partition. The format of each entry is:

```
partition: access-list [ <proc-range> ]
```

The *access-list* is specified as a comma separated list of names. Names may be user names or access lists defined in the `names` file (described below).

The optional *proc-range* is a comma separated list of processor identifiers. A contiguous range of processor identifiers may be specified by separating the lower and upper identifiers with a hyphen '-'. For example, `<0-2,8-10>` represents processors 0, 1, 2, 8, 9, and 10. The brackets are mandatory.

Lines beginning with a # are treated as comments and ignored. Entries may be continued over more than one line by preceding the ends of all but the last line with a backslash '\'.

Names File

Access lists (as referred to by the `permissions` file above) are defined in the `names` file. Definitions are hierarchical allowing lists to be defined in terms of other lists. Each entry in the `names` file consists of the list name and a space separated list of users or other lists that are members of the list. Recursive definitions are not allowed.

```
list-name: [list] [user-list]
```

Lines beginning with a # are treated as comments and ignored. Entries may be continued over more than one line by preceding the ends of all but the last line with a back slash '\'.

Example Files

The following example `permissions` file specifies that `software` and `engineering` members can use the `parallel` partition, but only `software` members can use the `test` partition. Permitted users of the `parallel` partition can spawn applications from processors 64, 68, or 72, whereas processors 64 and 68 can be used to spawn jobs in the `test` partition.

```
parallel: software engineering <64,68,72>
test: software <64,68>
```

The following example `names` file identifies the users in each of the two groups referred to by the `permissions` file above:

```
software: duncan eric tony documentation
documentation: andy
engineering: gerry mike
```

Note that the `software` group includes all members of the `documentation` group.

Login Load Balancing

The login load balancer uses the CS-2 Resource Management system to distribute user login shells across the processors that are dedicated to interactive development. The balancer uses users' identifiers and the current machine loading to determine the most appropriate processor for each login request.

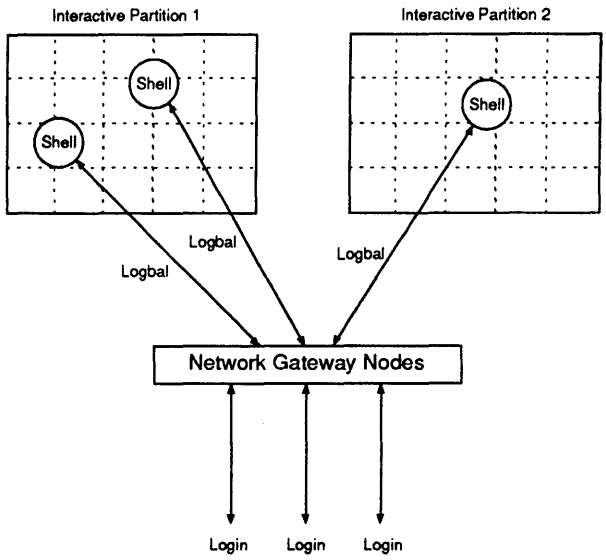
Description

The login load balancer requires two types of partition to be defined in the CS-2 machine (although small configurations may prefer to combine these).

The *network gateway* partition consists of device management processors (SPARC/IO boards with Ethernet, FDDI, or HiPPI interfaces); these are visible externally and run the Unix login processes. The *interactive* partitions contain

processors that are dedicated to executing users' command shells. Multiple interactive partitions may be created, allowing specified groups of users to be separated and allocated dedicated resources.

Figure 2-1 Login Load Balancing



The login load balancer executes on the gateway processors in response to login requests from users — the password file for these users is modified to execute `logbal` in place of a command shell. The login load balancer passes the user's id to the partition manager, which allocates a processor for the user's shell (based upon system load and the default statistic). A login shell is then executed on this processor, with all terminal I/O being routed, via the kernels, between the shell and the network connection on the gateway processor.

Warning – It is not normally possible to issue login requests to any processors other than the gateway processors (although the System Administrator and others may be given this capability).

The statistic that is used to determine processor loading is specified in the `defaults` file in the directory `/opt/MEIKOcs2/etc/`*machine-name*, and can be either user CPU load, system CPU load, idle CPU time, disk activity, page swaps, or CPU load average.

All the user's commands, including sub-shells, are executed on the target (load balanced) processor. Explicit attempts to login to any other processor (using commands such as `login` or `rsh`) must be targeted at a gateway processor; these login requests will also be load balanced.

Disabling User Logins

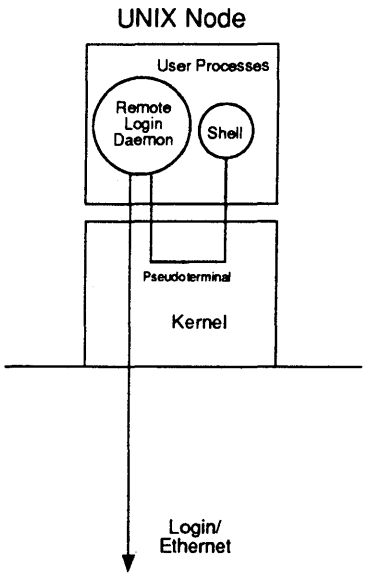
The presence of the `/etc/nologin` file causes the login loadbalancer to deny login requests made by users and to permit only root to login. When user logins are denied the contents of the file are written to the user's terminal.

This feature overrides the contents of the password file — even though a valid login entry exists for a user the login will be denied if `/etc/nologin` exists. This facility may therefore be used for temporary access restriction, in which case the `nologin` file should contain an explanatory message for users.

Implementation

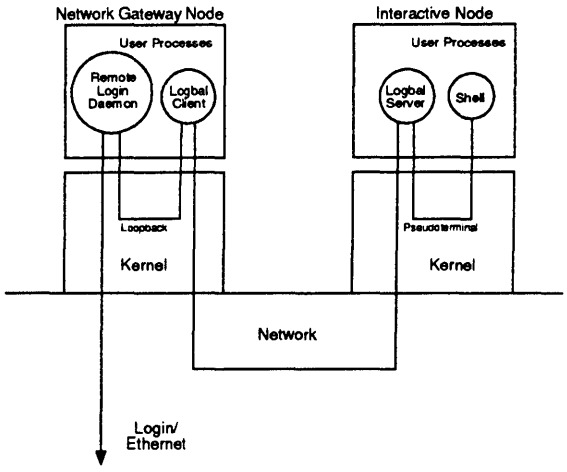
A conventional Unix login uses a pseudo terminal link to connect the login daemon process with the user's login shell, as shown in Figure 2-2.

Figure 2-2 Conventional Unix Login



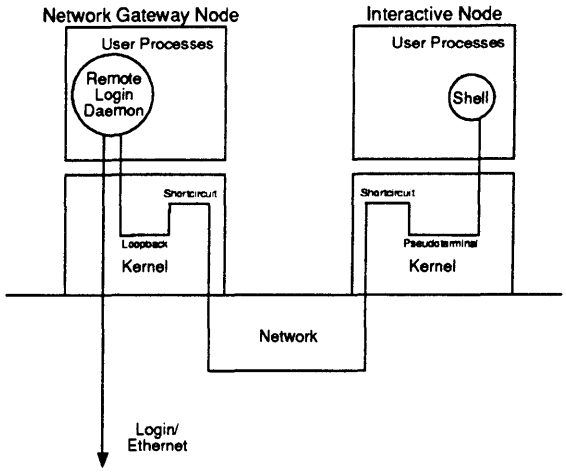
When using the login load balancer a TCP/IP link is created to connect the incoming login request stream and a pseudo terminal on the load balanced node. Two processes act as endpoints for this connection — `logbalclient` and `logbalserver` (see Figure 2-3).

Figure 2-3 Meiko's logbalserver and logbalclient



Having established a connection between the two processors, the two endpoint processes are short-circuited by a device driver, `/dev/shtcirc`. This device allows two kernel streams to be directly connected together without the need for a separate data-forwarding process (Figure 2-4).

Figure 2-4 Short Circuiting



The connection between the original login daemon process and the short-circuited link is set up as a loopback link, simply forwarding data in and out of the short-circuit and acting as a kernel gateway for the login connection. The `ptsfilt` streams module is pushed onto this loopback, to allow the login daemon to perform its internal communications with the remote pseudo terminal link.

Warning – The short-circuit device must be installed in `/kernel/drv/shtcirc`, **and the** `ptsfilt` **streams module must be installed in** `/kernel/strmod/ptsfilt`.

System Administration

Before installing the login load balancer there are several resource management issues that must be addressed. A suitable configuration must be defined to include a network gateway partition called `login`, and one or more interactive partitions. The System Administrator must decide how many gateway processors should be added into the `login` partition, and the identities of the users that are permitted to access them. The Administrator must put similar thought to the interactive partitions, where the login shells are to be executed, only

here the number of partitions needs to be evaluated as well as the distribution of resources and users within them. These issues are discussed fully elsewhere in this document.

Having defined the partitioning and user access rights within the CS-2, the System Administrator can install the login load balancer. This is a three stage process, as shown below[1].

1. Modify the password file entry for each user that is to be load balanced.

The password file is modified so that the load balancer (`logbal`) is executed in the `login` partition whenever the user logs on. The password file entry for the user `mike` would therefore look something like:

```
mike:x:100:10:Mike Smith:/home/mike:/opt/MEIKOcs2/bin/logbal
```

Warning – The password file entry for the System Administrator should not be modified and should refer to a command shell. This will allow access to the gateway processors.

Warning – If you define new login accounts at this stage remember to include corresponding entries in the shadow file (`/etc/shadow`) as described in the operating system documentation.

2. Specify each user's preferred shell.

To identify the preferred shell for each (load balanced) user the file `/opt/MEIKOcs2/lib/logbalshells` lists each user and the pathname for their command shell. The entry for the user `mike` might look like:

```
mike            /usr/bin/csh
```

1. Netgroups could be used in place of the password file modifications described, but note that netgroups may not be supported in future releases of the operating system.

3. Install the remote process execution daemon.

The login load balancer uses Meiko's `logbalserver` remote process execution daemon to start-up the remote shell. This daemon should be executed on each of the processors in the network gateway partition and the interactive partitions, and can be started automatically by adding the following entry to each processor's `/etc/inetd.conf` file — this happens as part of the resource manager package installation.

```
logbal stream tcp nowait root /opt/MEIKOcs2/bin/logbalserver logbal.server
```

The privileged TCP/IP service named `logbal` must be made available, either in all the local `/etc/services` files or, more conveniently, in the NIS/NIS+ services map:

```
logbal 900/tcp     # Meiko login load balancer support
```

Testing the Login Load Balancer

1. Following the installation of the `logbalserver` the System Administrator should be able to use the command line interface (`logbalclient`) to execute processes on remote processors.
2. You can use `pandora` to visualise the placement of user shells to confirm that the login load balancer is functioning correctly.
3. Poor response at the users' terminals implies that the gateway processors may be heavily loaded by login connections — use `pandora` to visualise the loading of the processors. Improved response can be obtained by increasing the number of gateway processors and distributing the login requests among them. Similarly the loading of processors in the interactive domain can be visualised and can be reduced by increasing processor numbers and the distribution of users among them.

Machine Description File

The machine description file `machine.des` in the directory `/opt/MEIKOcs2/etc/`*machine-name* lists the resources in the CS-2 system and their physical location. The file is read by the machine manager.

A machine description is hierarchical. Object descriptions begin at a high level and break down into lower level components. At the highest level the machine is viewed as a single entity; attributes that apply to the machine as a whole are specified here. At the next level there are the three module types — either processor, switch, or peripheral. At the lowest level there are the modules' contents.

The machine description file consists of three basic components: version control, machine attributes, and module attributes. Each component consists of a keyword and attribute/value pairs. The machine manager is insensitive to the case of text, and ignores white space. Attribute/value pairs may be entered separately, one per line, or specified as a comma separated list. Text on a line after a # is ignored.

Example machine description files are stored in the directory `/opt/MEIKOcs2/etc/`*machine_name*; these files are called `templaten.des` and describe machines of various sizes.

Version Control

All machine description files that conform to this documentation should begin with `FileVersion: 1.3`. All text before this attribute is ignored.

Machine Attributes

The list of machine attributes are preceded by the `MACHINE` keyword. The following attribute/values pairs must be specified:

`bays:` *number*
The number of bays in the system. There must be at least one bay; each bay supports up to 8 modules. The number of bays must be defined first as it is used to size data structures.

`levels:` *number*
The number of levels in the switch network.

Module Attribute

A module definition must be created for each module in the system. The format of each entry is:

```
Module type.
Module attributes ...
Board attributes ...
```

Module Type

The module type may be either `Processor_Module`, `Switch_Module`, or `Peripheral_Module`.

Module Attributes

The following module attributes may be specified:

`ModuleAddress:` *number*
The module address is set by the Installation Engineer using a switch on the rear of the module; each module must have a unique number. The module address is specified as a hexadecimal number preceded by 0x.

`Level:` *number*
The level in the CS-2 network at which the module is installed. Level 0 is the top level. This attribute is applicable to processor and switch modules only.

`NetId:` *number*
This attribute must be specified after the `Level` attribute described above, it is only applicable to processor or switch modules. It identifies the module's position in the data network.

`SmallSwitchCards:` *mask*
`SwitchBufferCards:` *mask*
Defines switch cards (MK511) and switch buffer cards (MK512) installed in a module. This attribute is only applicable to processor modules, and must be specified after the `Level` and `NetId` attributes described above. Bits set in the mask correspond to the positions of cards in the module.

`Plane:` *number*
This attribute identifies the switch plane that this module contains, it is only applicable to switch modules, and must be specified after the `Level` attribute described above. A default of 0 is assumed if this attribute is not specified.

`Layer:` *number*
This attribute defines the network layer that this module's switches belong to, it is applicable to switch modules only. Valid settings for this attribute are 0 or 1.

Board Attributes

Board attributes are preceded by the `Board` keyword. A module can have up to 4 boards. The following board attributes may be specified:

`BoardId:` *number*
The position within the module of the board. The right most board (when viewed from the front of the module) is number 0, the left most board is number 3.

`BoardType:` *string*
The type of the board. The following board types are supported:

Board Type	Description
`mk401`	SPARC/IO board.
`mk403`	Vector processing element.
`mk405`	4 processor SPARC/Compute board.
`mk523`	Switch card.
`mk522`	Switch card.

`Name:` *string {string, string, string}*
Defines the hostname for this processor. Multiple names are required for multiprocessor boards. Note that these names must correspond to hostname and Internet address definitions in the `/etc/hosts` file.

`Boot:` *string*
This attribute specifies how processors are to boot. The string may be either `disk`, meaning boot from a local disk, or the name of a boot server. This attribute must follow the Name definition and only applies to processor boards.

Example Machine Description File

The following example machine description file describes a single module machine that contains four MK401 boards:

```
# Example machine description file for 1 module

FileVersion: 1.3

MACHINE
bays: 1
levels: 1

PROCESSOR_MODULE
ModuleAddress: 0x1
Level: 1, NetId: 0, SmallSwitchCards: 0x01
  BOARD BoardId: 0, BoardType: MK401, Name: cs2-0, Boot: disk
  BOARD BoardId: 1, BoardType: MK401, Name: cs2-1, Boot: cs2-0
  BOARD BoardId: 2, BoardType: MK401, Name: cs2-2, Boot: cs2-0
  BOARD BoardId: 3, BoardType: MK401, Name: cs2-3, Boot: cs2-0
```

Defaults File

The `defaults` file contains site specific system defaults. The file may (but need not) be present in the `/opt/MEIKOcs2/etc/`*machine-name* directory. The `defaults` file is read by most resource management daemons and utilities.

The `defaults` file consists of a list of attribute/value pairs. Within the file a # marks the start of a comment, and white space is ignored. Numeric values may be specified in decimal or in hexadecimal (by preceding the number by `0x`).

The following attributes may be specified. Where boolean values are requires, `0`, `off` and `no` are equivalent, as are `1`, `on`, and `yes`.

`access-control` *boolean partition ...*
Enables access control as defined by the `permissions` file. When no partition is specified the access control applies to the whole configuration (all partitions). Access control can be restricted to a subset of the configuration by listing partitions names explicitly.

By default access control is enable for all partitions; if you wish to enable access control for just a few partitions you must remember to explicitly disable the remainder — the easiest way to do this is to specify `access-control off` first.

`accounting` = *boolean*
Either 0 (false) or 1 (true), enables generation of session accounting data. Default is 0.

`h8-rom-revision` = *number*
The minimum ROM revision for the control network. If any board has a lower ROM revision then the system will not start.

`halt-on-error` = *boolean*
Either 0 (false) or 1 (true), instructs the machine manager to halt (or not) if the definition found in the machine description file does not match the machine. Set to 1 while setting up a machine, otherwise 0.

`hardware-broadcast` = *boolean*
Either 0 (false) or 1 (true) specifying the availability of a hardware broadcast capability. The default is 1.

`information-hiding` = *boolean*
Either 0 (false) or 1 (true). If set to true users may only have visibility of those resources they have permission to see.

`log-perm-errors` = *boolean*
Either 0 (false) or 1 (true), enable logging of attempted access violations. Default is 1.

`log-stats` = *boolean*
Either 0 (false) or 1 (true), enable generation of usage logs. Default is 0.

`lbal-statistic = string`
The statistic that is to be used by the login load balancer to determine the least active node for a user login. The following strings are permitted; only one may be specified:

Attribute	Description
`user-cpu`	CPU activity for user applications.
`sys-cpu`	CPU activity for the system.
`idle-cpu`	Idle CPU time.
`disk-io`	Disk IO activity.
`paging`	Page swaps.
`load`	CPU load.

`partition = string`
The name of the default partition for programs to execute in. The default is `login`

`rom-revision = number`
The minimum Open Boot ROM revision that must be present on all boards in the system. If any board has a lower ROM revision then the system will not start.

`timelimit` *seconds partition ...*
Specifies the maximum time, in seconds, that partitions can be held by an application, after which the application is sent a SIGTERM. The application is allowed a short period to react to the signal, specified by `grace-period`, after which a SIGKILL signal is sent.

`grace-period` *seconds partition ...*
Used with the `timelimit` attribute described above.

Example Defaults File

An example `defaults` file is listed below:

```
# CS-2 Resource Management System
# sample system defaults file

access-control off
access-control on parallel batch
timelimit 3000 parallel
grace-period 60 parallel
rom-revision = 120
h8-rom-revision = 0x93091017
lbal-statistic = user-cpu
partition = parallel
```

Procedural Interface to the Resource Management System

The System Administrators interface to the resource management system is via the library `librms.a`. This library is available to both users' and the System Administrator, but note that only the Administrator can use the functions that change a configuration; users are restricted to the query functions.

Querying the Configuration and Executing Programs

The following functions allows users to query the system configuration and to execute programs. The functions described in this section do not require root access privileges.

Querying Resource Availability

The resource management system supports a query interface that allows applications to explore the resources available to them. This interface covers both the hardware and the active configuration. Users make enquiries using the function `rms_describe()`:

```
void *rms_describe(RMS_OBJECT_TYPE type, int objectId);
```

where `type` is the `type` of object to be described and `id` is its identifier. Supported object types are defined by `RMS_OBJECT_TYPES` in the header file `<rmanager/uif.h>` and include:

```
typedef enum {
    RMS_MACHINE       = 0,/* the whole machine              */
    RMS_MODULE        = 1,/* modules                        */
    RMS_BOARD         = 2,/* boards                         */
    RMS_SWITCH        = 3,/* switches                       */
    RMS_PROC          = 4,/* processing elements            */
    RMS_DEVICE        = 5,/* peripherals                    */
    RMS_CONFIGURATION = 6,/* working set of partitions      */
    RMS_PARTITION     = 7,/* individual partition           */
    RMS_TARGET        = 8,/* application target             */
    RMS_JOB           = 9,/* parallel program               */
} RMS_OBJECT_TYPES;
```

To describe a CS-2 system in full call `rms_describe()` with object type `RMS_MACHINE`, it will return a pointer to a machine description structure:

```
typedef struct {
   int nLevels;           /* number of network levels */
   int nModules;          /* number of modules (all types) */
   int nBoards;           /* number of boards */
   int baseProc;          /* first processor */
   int topProc;           /* last processor */
   int nProcs;            /* number of processors */
   int nSwitches;         /* number of switches */
   int nDevices;          /* number of peripherals */
   int nBays;             /* number of bays */
   int layers;            /* bit mask of network layers */
   int serialNumber;      /* machine serial number */
   char name[NAME_SIZE]; /* machine name */
   map_t map;             /* processor map */
   map_t proc_map;        /* processors configured in/out */
   map_t sw_map;          /* switches configured in/out */
   map_t board_map;       /* boards configured in/out */
   map_t module_map;      /* modules configured in/out */
} machine_t;

machine_t *machine = (machine_t *)rms_describe(RMS_MACHINE, 0)
```

There is only one machine so `objectid` is ignored. A machine consists of modules, boards, processors, and switches. Each has an `id` numbered from `0`. The machine description gives the total number of objects of each type.

Applications that access the hardware directly (the test code for example) need machine description information and start their enquiries with a request to describe the machine. Applications that are designed to operate on the active configuration (such as a site specific job scheduler) should start with descriptions of the configuration and its partitions.

```
typedef struct {
    char    name[NAME_SIZE];    /* configuration name          */
    int     nPartitions;        /* number of partitions        */
} config_t;
```

```
typedef struct {
   int id;                  /*logical id of partition*/
   charname[NAME_SIZE];     /*partition name*/
   int baseProc;            /*first processor*/
   int topProc;             /*last processor*/
   int nProcs;              /*number of processors*/
   int baseTarget;          /*first target in partition*/
   int nTargets;            /*number of targets*/
   int baseJob;             /*first job*/
   int nJobs;               /*number of active jobs*/
   time_t timestamp;        /*last time info changed*/
   int active;              /*running or not*/
   map_t map;               /*processor map*/
} partition_t;
```

The following code fragment illustrates how to query information about a particular partition:

```
partition_t *p;
config_t *config;
int id;

config = config = (config_t *)rms_describe(RMS_CONFIG, 0);
for (id=0; id < config->nPartitions; id++) {
    p = (partition_t *)rms_describe(RMS_PARTITION, id);
    if (p && strcmp(p->name,"interesting") == 0) {
        ..................................
        ..................................
    }
}
```

Having discovered that the `interesting` partition is active you could then check how many processors it contains and start a job on them. However, this partition may be in use, further enquiries will tell you who by, how many processors they are using and for how long.

Resource Requests

Details of the resources required by a parallel program are specified by resource requests, defined by the structure `resourceRequest` defined in the header file <rmanager/uif.h>. A `resourceRequest` structure is passed to the function `rms_forkexecvp()` to start a parallel program.

```
typedef struct {
 int baseProc;       /* processor base (relative to partition)*/
 int nProcs;         /* number of processors */
 int memory;         /* MBytes of memory */
 int timelimit;      /* run-time in seconds */
 int verbose;        /* enable verbose reporting */
 int debug;          /* run process under debugger */
 int stdiolog;       /* log stdio to file */
 int solarisBinary; /* multiple copies of a solaris binary */
 int createCore;     /* allow core file creation */
 int tid;            /* target identifier */
 char partition[NAME_SIZE]; /* partition to use */
} resourceRequest;
```

Resource requests can also be set as environment variables. The set of supported environment variables is as follows:

RMS_PARTITION	The partition to use.
RMS_NPROCS	The number of processors required.
RMS_BASEPROC	Id of the first processor to use; usually the first process in a parallel application is loaded onto processor 0 in the partition or the first available processor if other applications are running.
RMS_MEMORY	The minimum memory requirements for each processor, suffixed by K or M (for Kilobytes or Megabytes respectively).
RMS_STDIOLOG	Preserve stdio/stderr from each process. The default is false.
RMS_CORESIZE	Enable core dumping. The default is no core files.
RMS_VERBOSE	Set level of status reporting.

Default values of these resource requests are specified in the `defaults` file. You can determine the default setting of these variables by using the `rms_defaultResourceRequest()` function:

```
resourceRequest* rms_defaultResourceRequest();
```

Load Balancing

The function `rms_lbal()` is passed a partition name and a pointer to an `lbal_t` structure. On return the structure identifies the processor that is least loaded in that partition.

```
int rms_lbal(char* partition, lbal_t* lbalInfo);
```

The `lbal_t` data structure is defined in the header file `<rmanager/uif.h>` as:

```
typedef struct{
 char h_name[NAME_SIZE];    /* Hostname */
 long addr;                 /* Internet Address */
} lbal_t;
```

The statistic that is used to measure processor loading is specified in the `defaults` file.

Warning – `rms_lbal()` is only applicable to processors in an interactive partition.

Editing the Configuration

The procedural interface to the resource management system allows site specific applications to modify the configuration of the system. Such applications must be run as `root`. The configuration editing interface consists of a set of functions that manipulate `rmsobj_t` structures (defined in the C header file `<rmanager/uif.h>`):

```
typedef struct {
    RMS_OBJECT_TYPES type;      /* object type */
    union {
        machine_t   machine;
        module_t    module;
        board_t     board;
        switch_t    sw;
        proc_t      proc;
        device_t    device;
        config_t    config;
        partition_t partition;
        target_t    target;
        job_t       job;
    } objs;
} rmsobj_t;
```

Objects are created with `rms_create()` and combined with `rms_add()`

```
rmsobj_t rms_create(RMS_OBJECT_TYPES type);
int rms_add(rmsobj_t *parent, rmsobj_t *child);
```

They are saved to disk with `rms_put()` and restored with `rms_get()`.When selecting an object with `rms_get()` it is necessary to specify the parent, the object type, and the name

```
int rms_put(rmsobj_t *object));
rmsobj_t rms_get(rmsobj_t *parent, RMS_OBJECT_TYPES type, char *name);
```

To delete an object use `rms_free()`. The object's parent must also be specified. If the `delete` flag is non-zero the object will be deleted permanently — by removing its filesystem state.

```
int rms_free(rmsobj_t *parent, rmsobj_t *object,int delete);
```

The routines `rms_putAttr()` and `rms_getAttr()` get and set object attributes.

```
int rms_putAttr(rmsobj_t *o, RMS_ATTR_TYPES attr, void *value);
void *rms_getAttr(rmsobj_t *o, RMS_ATTR_TYPES attr)
```

Currently supported attributes are:

```
typedef enum {
    RMS_NAME,        /* object name         */
    RMS_DESCRIPTION, /* object description  */
} RMS_ATTR_TYPES;
```

RMS_NAME is used to put or get the object's name. An `rms_get()` used in conjunction with RMS_DESCRIPTION returns a pointer to the object description, this has the same effect as using `rms_describe()`.

The function `rms_commit()` instructs the resource management system to start or stop a partition or configuration specified by `object`. The flag `doKill`, if set true instructs the partition managers to kill existing processes, first with `SIGTERM` and then, 10 seconds later, with `SIGKILL`. The flag `stop` instructs `rms_commit()` to stop a running configuration/partition. The final argument `args` specifies the partition manager arguments (interactive/background, reporting level, etc.).

```
int rms_commit(rmsobj_t *object, int stop, int doKill, pmargs_t args);
```

All configuration editing routines return object pointers or 0 on success, `NULL` pointers or -1 on error depending upon their type.

Warning – Not all operations work on all types of objects.

These routines are used within `rcontrol` and `pandora`. They can be used by local tools designed to visualise or edit the configuration of a machine.

Hostname, Elan ID, Ethernet Address Translation

Two functions are provided for hostname to Elan ID translation.

```
int rms_elantohost(char* hostname, int elan);

int rms_hosttoelan(char* hostname);
```

Return values are 0 or an ElanID on success, or -1 on failure.

In addition two functions are provided for translation from Elan ID to standard 48 bit Ethernet address (of which only the last two fields are used):

```
int rms_ntoelan(struct ether_addr* e);

struct ether_addr* rms_elanton(int elan);
```

The `ether_addr` structure is defined in `<netinet/if_ether.h>`.

See also the description of hostnames and Elan Id's in *CS-2 Node Naming* on page 67.

2

Administering the CS-2 Filesystem 3

Overview

This chapter describes administration of the CS-2 filesystem. It covers both the Solaris Unix filesystem (UFS) and the Meiko parallel filesystem (PFS). It is designed to be read alongside the *SunOS 5.1 Routine System Administration Guide* — this describes the principles of filesystem usage and administration, including the Solaris Virtual File System (VFS) which provides a single set of administrative commands for use with filesystems of all types.

A CS-2 system can have three types of filesystem:

Local	For local temporary I/O.
Global	For generally accessible files.
Parallel	For high capacity and bandwidth.

The Unix Filesystem (UFS)

The Unix filesystem (UFS) is used on all disk devices in the CS-2 system. Disks must be partitioned into slices, labelled, and filesystems built on the slices as described in the *SunOS 5.1 Routine System Administration Guide*. Meiko is developing tools to simplify this process; they will be provided in future software releases.

The Network Filesystem (NFS)

NFS is used to make filesystems globally visible throughout a CS-2 system. An optimised version of NFS uses the CS-2 data network to move data between NFS clients and servers.

In addition NFS is used to mount external filesystems. This is not generally recommended for large systems as it can impose significant external network loading with a consequent reduction of performance of the parallel machine.

The Parallel Filesystem (PFS)

The Meiko parallel filesystem is striped over many data storage devices. It provides a mechanism for creating very large filesystems (greater than any individual UFS filesystem could support) and for providing high bandwidth concurrent accesses to data that is held within the filesystem.

The data storage devices used by the PFS are Solaris filesystems, such as the UFS, which will typically be distributed within the CS-2 using the NFS.

Filesystem Administrative Commands

Most filesystem administrative commands have a generic and a filesystem specific component. The following commands may be used with the PFS (using the `-Fpfs` option where appropriate):

`df`	Report number of free blocks in filesystem.
`mkfs`	Makes a new filesystem.
`mount`	Mount a filesystem.
`mountall`	Mounts all filesystems specified in a filesystem table.
`umount`	Unmount a filesystem.
`umountall`	Unmounts all filesystems specified in a filesystem table.

The other filesystem administrative commands provided by the operating system do not have a direct application to the PFS, although they will be used to administer the filesystems that the PFS is built upon.

Manual Pages

Both the specific and generic filesystem commands have manual pages; the specific filesystem manual page is a continuation of the generic one. To look at a specific manual page append an underscore and the filesystem type (`ufs` or `pfs` in this case) to the generic command name. For example, type `man mount` and `man mount_pfs`.

How to Find out the Type of a Filesystem

The `fstatvfs`(2) and `statvfs`(2) system calls return a generic superblock describing a filesystem, including it's size, free space, and filesystem type. For a PFS the type string will be "pfs", for an NFS mounted filesystem it will be "nfs", and for a local Unix filesystem it will be "ufs".

You should add the following test into any program that performs PFS specific operations; `fd` is a file descriptor for an opened file:

```
static int checkPfsFile (int fd) {
   struct statvfs buf;

    if (fstatvfs (fd, &buf) < 0)
      return (-1);

    if (!strcmp (buf.f_basetype, "pfs"))
      return (0);

    return (-1); /* not a PFS file */
}
```

To identify a mounted filesystem's type from your command shell you should examine the `/etc/mnttab` file as shown below. This file is updated by the operating system whenever a filesystem is mounted or unmounted.

```
root@cs2-0: grep /mnt /etc/mnttab
/pfs_admin/map /mnt pfs rw,suid 749463391
```

`/mnt` is the mount point in your filesystem.

The Solaris documentation describes other ways that you can identify a filesystem's type (using the `mount (1m)` command for example). These will show the underlying data filesystem but will not identify the PFS filesystem itself.

Setting Up a Unix Filesystem

The Unix filesystem is used to support local disk I/O. Disks must be formatted (this is usually done by the manufacturer), partitioned into slices, labelled, and filesystems must then be created on the slices.

The following example shows the steps necessary to set up a disk for local swap and temporary I/O. It is typical of the setup for a disk that is attached to a vector processing element.

1. **Partition the disk creating one large slice — usually slice 7. Label the disk.**
   The *SunOS 5.1 Adding and Maintaining Devices and Drivers* manual includes a description of this process using the `format` command and its `partition` and `label` options.

2. **Reboot the processor using the reconfigure option.**
   Use the `boot` command with its `-r` option. This will install the appropriate device drivers.

```
ok boot -r
```

3. **Build a filesystem.**
   Use the `newfs` command, taking great care to specify the correct device as an argument. For example:

```
# newfs /dev/rdsk/c0t0d0s7
```

4. **Mount the filesystem.**
   First create a mount point, add an entry to the `/etc/vfstab` file, and then mount the disk.

```
# mkdir /scratch
# mount /dev/dsk/c0t0d0s7 /scratch
```

   The entry in the `/etc/vfstab` file would look like:

```
/dev/dsk/c0t0d0s7 /dev/rdsk/c0t0d0s7 /scratch ufs 1 yes -
```

Globally Mounting a Unix Filesystem

CS-2 uses NFS to make filesystems globally accessible. It is necessary to `share` the filesystem, create mount points for the filesystem, and create a vfstab entry.

In the following example the processor `cs2-0` shares the `/global` filesystem with all processors in the `parallel` partition.

1. **Share the filesystem.**
   Create an entry in the `/etc/dfstab` file on `cs2-0` to specify that the `/global` filesystem is available for remote mounting:

```
share -Fnfs -orw /global
```

   Execute the `shareall` command so your edits take effect immediately.

```
# shareall
```

2. **Create the mount points.**
   You need to create mount points on all the processors in the partition. You can do this with `prun`, using it to execute the `mkdir` command.

```
# prun -p parallel -s mkdir /global
```

3. **Mount the filesystems.**
   You can mount the filesystem on all processors in the partition with the `prun` command, using it to execute the `mount` command.

```
# prun -p parallel -s /usr/sbin/mount -Fnfs -oelan cs2-0:/global /global
```

4. **Edit the vfstab files.**
   Edit the vfstab files to ensure that the filesystem is mounted automatically whenever a node is (re)booted.

   During the setup of your CS-2 a `vfstab.dir` directory is created in `/export` on your main server and is mounted on `/etc/vfstab.dir` by all the processors in your system. Within this directory are a number of `vfstab` files; the names' of these files defines the processors that they apply to:

`vfstab.clients`	Filesystems to be mounted by all client processors.
`vfstab.`*hostname*	Filesystems to be mounted by *hostname*.
`vfstab.global`	Filesystems to be mounted by all processors.
`vfstab.site`	Filesystems mounted by all processor from workstations and other processors on your local site network.

   The vfstab files in `/etc/vfstab.dir` are read as each processor enters run level 3. The script `/etc/rc3.d/S74nfs.cs2` is defined to read only those vfstab files that are appropriate to its host processor.

   When defining a new filesystem for global mounting on all processors in your machine add the following entry into the `vfstab.global` file. If the filesystem is used by a limited group of processors create a `vfstab.`*hostname* file for each processor, and add the following line into each file:

```
cs2-0:/global - /global nfs - yes -
```

   Note that each processor retains its own `/etc/vfstab` which will define the local filesystems that are to be mounted by the processor (i.e. those on local disks).

5. **Check the global mount.**
   You can use `prun` to check that the filesystem is mounted on each processor in the partition.

```
# prun -p parallel -sv df /global
```

The Parallel Filesystem

The PFS is in two parts: a map filesystem and a data filesystem.

The map filesystem contains a map from PFS filenames to data filenames on the underlying data filesystems, and it also includes mount points for the data filesystems. References to the map filesystem are made to the `mount(1m)` command whenever a PFS is mounted.

The data filesystem is distributed. It consists of a number of networked Unix filesystems, the number and types of these filesystems are specified by the System Administrator when the PFS filesystem is created. Files that are written to the PFS are striped over these data disks using a stride and an offset. The stride is specified by the System Administrator at the time the PFS is created and must be an integer multiple of the logical block size of the underlying data filesystems, which for a UFS filesystem is 8 Kbytes. The offset determines which of the underlying filesystems will store the first stripe; this is automatically randomised on a per-file basis over those filesystems that are local to the PFS (or all underlying filesystems if none of them are local).

The default mapping of PFS files onto the underlying data filesystems has been chosen to balance space utilisation over all the data filesystems and to minimise network activity for small files. This strategy is set at filesystem creation time, but can be changed by PFS extensions to `ioctl(2)`.

Performance Factors

The physical location of the underlying filesystems is unimportant to the PFS — they may all be attached to one CS-2 processor, distributed among a number of CS-2 processors, or they may be attached to networked workstations. You may use disks that are dedicated to the PFS, or use partitions on your system disks.

Your choice will however impact upon the performance of your PFS. Dedicated disks, each attached direct to a CS-2 node via a dedicated bus, will offer the highest performance.

Before you create a parallel filesystem you will need to consider:

- The number of participating data filesystems.
- The stride size used to distribute the data over the available data filesystems.

Factors that will influence your decisions are:

- The number of filesystems that you can allocate to the PFS, and their placement relative to the processes that must access them.
- The size of the files that you expect to create.
- The file access required by parallel processes; the stride size for the filesystem should be chosen so that the processes in a parallel application rarely have to compete for access to the same file slices.

Support Files

A number of files are created by `mkfs(1m)` when the PFS filesystem is created. The following files are created in the map filesystem:

`.pfsid`	Identifies this directory in relation to others participating in the PFS.
`.pfsdflt`	Default mapping for new files.
`.pfsmap/`	Map file directory. Contains the names of files and directories that are created under the PFS. The inode numbers for these files are used to create the corresponding data files on the data filesystems.
`.pfsdata`*n*	A number of files with numeric suffixes, one for each of the data filesystems. These files are symbolic links to the mount points for each of the data filesystems. The mount points will be within the `/pfs_admin` directory (created my `mkfs` and described below).

`mkfs` creates a directory at the root of the map filesystem. This directory, called `/pfs_admin`, contains mount points for all of the underlying filesystems that are used by the PFS. These mount points are used by `mkfs` to initialise the data filesystems with

`pfsid` identifier; all the filesystems are unmounted before `mkfs` completes.

The `/pfs_admin` directory is also created by the `mount(1m)` command on each processor that mounts the PFS — again this will contain the mount points for the underlying data and map filesystems. This directory and the mount points within it must not be deleted or altered until you have unmounted the PFS.

Data Filenames

Files and directories that are created under the PFS map onto filenames in the map filesystem and to data files on all of the underlying data filesystems. The names of the data files are derived from the inode number of the filename as it is stored in the map filesystem.

The following table summarizes the mapping from map file inode number to data filename. For a map file with inode number 26485 a data directory hierarchy of `/d02/d64/85` is created on each of the data filesystems, each instance of the file 85 containing stripes from the users files.

Inode Number	Data Filename
1	/01
99	/99
100	/d01/00
9999	/d99/99
26485	/d02/64/85

Creating a Parallel File System

Each of the storage devices that will be used by the PFS must be formatted and initialised with a filesystem, for example the UFS. All the filesystems (both map and data) must be available for remote mounting by the CS-2 processors that will use the PFS.

The following initialisation procedure may be used on any of the CS-2 processors that have access to your designated map and data filesystems. An example of how to create a PFS on the local system disks of a small CS-2 system is included in Appendix A.

1. **Identify the data filesystems.**
   The data filesystems can be on disks that are mounted directly on your CS-2 processors, or they can be on remote devices that are available via an external network.

   Having identified the data filesystems you must create a file that includes the network pathname of these filesystems; the file contains one line for each file system, each line being a full network address (in the form *host:pathname*). The name of this file is unimportant — later examples assume it is called `/tmp/fslist` — see Step 3. The following example file identifies two filesystems, one hosted by `deimos` and one by `phobos`.

```
deimos:/pfsdata
phobos:/pfsdata
```

   When referencing remote filesystems it is important that the filesystem list refers directly to the host of each filesystem and not to a local mount point on your CS-2 processor. By using a local mount point all PFS accesses will be routed through the local node — an unnecessary point of congestion.

   You must ensure that all the data filesystems are owned by `bin`, in group `bin` (See `chown`(1m) and `chgrp`(1m)). The permissions must be set to give root read, write, and execute permission (7*xx*).

2. **Identify the map filesystem.**
   The data filesystem can also be on a disk that is local to the CS-2 or on a remote machine. Only a relatively small amount of space is required for the map filesystem as it contains only filename maps and symbolic links.

   In the following example, the map filesystem will be created in `/pfsmap` on `ganymede` — see Step 3.

You must ensure that the map filesystem is owned by `bin` and in group `bin` (see `chown`(1m), and `chgrp`(1m)). The permissions must be set to give root read, write, and execute permission (7*xx*).

3. **Initialise the map filesystem.**
   You initialise the PFS with `mkfs`(1m).

   The following example defines a PFS with 2 data filesystems (`nfs=2`), as identified by the filesystem list in `/tmp/fslist`. Files will be distributed among the data filesystems in 8Kbyte portions; this stride **must** be an integer multiple of the block size used on the underlying data filesystems. The map filesystem in this example resides on `ganymede`, and the filesystem list is `/tmp/fslist`.

```
cs2-0# mkfs -Fpfs -onfs=2,fslist=/tmp/fslist,stride=8K,verbose,force \
ganymede:/pfsmap
```

The `mkfs` command builds the map files. During the initialisation process the data filesystems are temporarily mounted in `/pfs_admin` in your root filesystem (`mkfs` creates the directory if it doesn't already exist).

Mounting the PFS

Having initialised the PFS it can now be mounted by any CS-2 processor that wishes to use it.

You mount a parallel filesystem with the `mount`(1m) command, specifying the map filesystem as the target for the mount. If the map filesystem is not local to the processor you must use the network pathname for the map filesystem. For example:

```
cs2-5# mount -Fpfs ganymede:/pfsmap /mnt
```

Access to the parallel filesystem is restricted by the permissions associated with the PFS directory itself (`/mnt` in this example). You should therefore adjust accordingly the permissions of the PFS filesystem on all the processors that mount the PFS. For example:

```
cs2-5# chmod 777 /mnt
```

Testing the Parallel Filesystem

You can test the parallel filesystem by using the `mkfile(1)` command to create some large files, as shown below. Note that you should test the PFS from a user account and not as root.

```
user@cs2-5% cd /mnt
user@cs2-5% mkfile 10k bigfile
user@cs2-5% mkfile 100k hugefile
user@cs2-5% ls -l /mnt
total 224
-rw------T 1 root other  10240 Sep 28 15:57 bigfile
-rw------T 1 root other 102400 Sep 28 15:57 hugefile
user@cs2-5% df /mnt
Filesystem     kbytes  used  avail capacity  Mounted
/pfs_admin/map 1923524 118 1731066     0%    /mnt
user@cs2-5%
```

Each of the data directories on the data filesystems will now contain files with sizes that are approximately equal to the total filesize divided by the number of data filesystems. See *Data Filenames* on page 57 for the naming conventions that are used in the data directories.

If you get an I/O error while writing files into the PFS you should check that your PFS stride size is compatible with the block size of the underlying data filesystems. The PFS stride size must be an integer multiple of the filesystem block size.

Creating and Accessing Files

PFS files can be created and manipulated by the standard I/O functions, such as `open`(2), and by any of the Unix I/O commands.

Parallel applications built upon the Elan Widget library may use the widget library's parallel file I/O functionality — see the description of EW_PFD(3x) in the Widget library documentation. Note that all of Meiko's current message passing libraries are built upon the Elan Widget library.

Where use of the Elan Widget library is not appropriate the processes in a parallel application may use the `open`() and `fseek`() calls to similar effect. In this case the distribution of data among the processes must correspond to the stride size and number of filesystems in the PFS. To avoid competition for I/O devices each process in the application should aim to write its data to a file stripe that is shared by no others.

Changing/Examining the Mapping of PFS Files with ioctl(2)

By default PFS files are distributed over all the underlying data filesystems using the stride size specified at filesystem creation time and a random offset that is generated on a per-file basis.

The mapping of PFS files onto the underlying data filesystems can be examined or changed with `ioctl`(2), which has been extended to include the following PFS requests:

Request	Meaning
PFSIO_GETMAP	Get a file's mapping. Requires a pointer to a pfsmap_t structure as the third argument to ioctl(2).
PFSIO_GETFSMAP	Get the default file mapping. Requires a pointer to a pfsmap_t structure as the third argument to ioctl(2).
PFSIO_SETMAP	Set a new file mapping. Requires a pointer to a pfsmap_t structure as the third argument to ioctl(2). If you change the mapping for an existing file then the file will be truncated and its contents deleted. You must therefore create a new file with the required mapping and copy the contents of the old file into the new.
PFSIO_GETLOCAL	Identifies the stripes of a PFS file that are local to the node that called ioctl(2). Requires a pointer to a pfsslices_t structure as the third argument to ioctl(2).
PFSIO_GETFSLOCAL	Identifies the stripes of a filesystem that are local to the node that called ioctl(2). Requires a pointer to a pfsslices_t structure as the third argument to ioctl(2).

Associated Data Structures

The pfsmap_t and pfsslices_t structures are defined in the header file /usr/include/sys/fs/pfs_map.h, and are described below.

pfsmap_t

The pfsmap_t structure is used by ioctl(2) to return information about a file's current mapping, or can be used specify a new mapping. The structure is defined as:

```
typedef struct pfsmap
{
   pfsslice_t pfsmap_slice; /* where in the pfs */
   u_long pfsmap_type;      /* type of mapping */
   union                    /* variant parameters */
   {
      pfsmap_1d_t m_1d;
   } pfsmap_u;
} pfsmap_t;
```

The type of mapping is described by the `pfsmap_t.pfsmap_type` field; currently only `PFSMAP_1D` (one dimensional maps) are supported.

The `pfsslice_t` structure defines the offset and number of data filesystems (by default count is the total number of data filesystems, and offset is a random number in the range [0,(`count`-1)]):

```
typedef struct          /* slice of a pfs */
{
   u_long ps_base;      /* starting offset */
   u_long ps_count;     /* # of file systems */
} pfsslice_t;
```

The `pfsmap_1d_t` structure defines the stride (by default this is 1, i.e. every data filesystem):

```
typedef struct
   u_long stride;
} pfsmap_1d_t;
```

pfsslices_t

The `pfsslices_t` structure is used by `ioctl`(2) to return information about PFS slices.

A slice is a file or part of a file that is distributed over a number of data filesystems. The `pfsslice_t` structure describes a single slice in terms of its offset and number of filesystems. The `pfsslices_t` structure is used by `ioctl(2)` to return information about a number of slices in a single call.

```
typedef struct
{
   int pss_size;              /* size of pss_slice */
   int pss_count;             /* entries in pss_slice*/
   pfsslice_t pss_slice[1]; /* slices */
} pfsslices_t;
```

```
typedef struct          /* slice of a pfs */
{
   u_long ps_base;      /* starting offset */
   u_long ps_count;     /* # of file systems */
} pfsslice_t;
```

Example

The following example uses `ioctl()` and the PFSIO_GETMAP request to examine the mapping used by a number of PFS files:

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/statvfs.h>
#include <sys/fs/pfs_map.h>


static int checkPfsFile (int fd)
{
   struct statvfsbuf;

   if (fstatvfs (fd, &buf) < 0)
      return (-1);

   if (!strcmp (buf.f_basetype, "pfs"))
```

```
        return (0);

    return (-1);
}


main(int argc, char** argv)
{

    /* Usage: showmap file ... */

    int fd;
    int file;
    pfsmap_t map;


    for(file = 1; file < argc; file++)
    {
        if( (fd = open(argv[file], O_RDONLY)) <0)
        {
            fprintf(stderr, "Can't open %s\n", argv[file]);
            exit(1);
        }

        if(checkPfsFile(fd))
        {
            fprintf(stderr, "%s not a PFS file\n", argv[file]);
            exit(1);
        }

        if(ioctl(fd, PFSIO_GETMAP, &map) < 0)
        {
            fprintf(stderr, "ioctl failed to get file map\n");
            exit(1);
        }

        printf("%s: ", argv[file]);
        printf("starts at filesystem %u, for %u filesystems: ",
            map.pfsmap_slice.ps_base, map.pfsmap_slice.ps_count);
        printf("stride is %u\n", map.pfsmap_u.m_1d.stride);

    }
}
```

When used to display the default mapping used by a number of PFS files, you will note that the offset used by each file is uniformly distributed over all the available filesystems to ensure that disk utilisation is uniform:

```
user@cs2: showmaps *.c
csn.c:   starts at 1, for 4 filesystems: stride 8192
csf.c:   starts at 0, for 4 filesystems: stride 8192
ptrce.c: starts at 2, for 4 filesystems: stride 8192
ptrcf.c: starts at 3, for 4 filesystems: stride 8192
ring.c:  starts at 0, for 4 filesystems: stride 8192
tcsn.c:  starts at 1, for 4 filesystems: stride 8192
tmsg.c:  starts at 2, for 4 filesystems: stride 8192
yp.c:    starts at 3, for 4 filesystems: stride 8192
ypthd.c: starts at 1, for 4 filesystems: stride 8192
```

CS-2 Node Naming 4

Machine Name

The choice of the generic machine name is made by the System Administrator; it can consist of any alphanumeric characters, but should be kept short. Choosing a name that can be easily used to generate individual hostnames is advisable — this means that the generic name should not end in a numeral.

The machine name is stored in the file `/etc/rms_machine`.

All of the examples shown in our documentation use the machine name `cs2-`.

Hostnames

Each processing element within the CS-2 will have a hostname which (for simplicity) should include both the machine name and a processor Id (`cs2-4`, for example). In addition nodes that are network gateways — those having external network interfaces — may also have other names associated with them.

/etc/hosts File

The names and internet addresses of all CS-2 processing elements are recorded in the `/etc/hosts` file; the following example shows the entries for an 8 element CS-2 machine in which the machine name is `cs2-`. Processing element 0 is a network gateway node and has aliases for use on the external network.

```
192.131.108.190 cs2-0 cs2
192.131.108.191 cs2-1
192.131.108.192 cs2-2
192.131.108.193 cs2-3
192.131.108.194 cs2-4
192.131.108.195 cs2-5
192.131.108.196 cs2-6
192.131.108.197 cs2-7
```

The IP subnetwork will normally have been allocated by Meiko from a batch designated for the data networks of CS-2 systems. You should advise Meiko if you would prefer to obtain the numbers from your own internet provider.

Node numbers must be unique and lie in the range 1–254. Hostnames must correspond with those in the machine description file.

Network Id

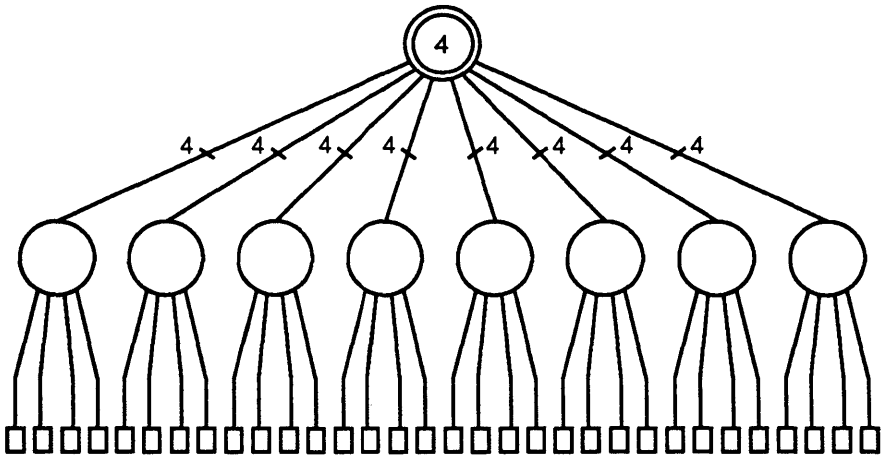
The Elan Id of a processing element is a decimal representation of its network address. The way in which network addresses are obtained is fully described in the *Communication Network Overview* — in summary they are obtained by defining the Elite link number that the data must pass out of as it works its way down from the top of the switch network. For the first processing element in the first module the route will be 0.0...0; that is, the data exits from link 0 at each network stage. The top switch (the switch at the uppermost switch level) will always have all eight of its links pointing down to lower levels, so routes will always take the form: <0-7>.<0-3>...<0-3>.

Determining an Elan Id is straight forward if the routes are displayed in their binary form. The Elan Id is simply the decimal representation of the binary route:

Route	Binary Representation	Elan Id
0.0.0	000 00 00	0
0.1.0	000 01 00	4
0.2.0	000 10 00	8
0.3.0	000 11 00	12

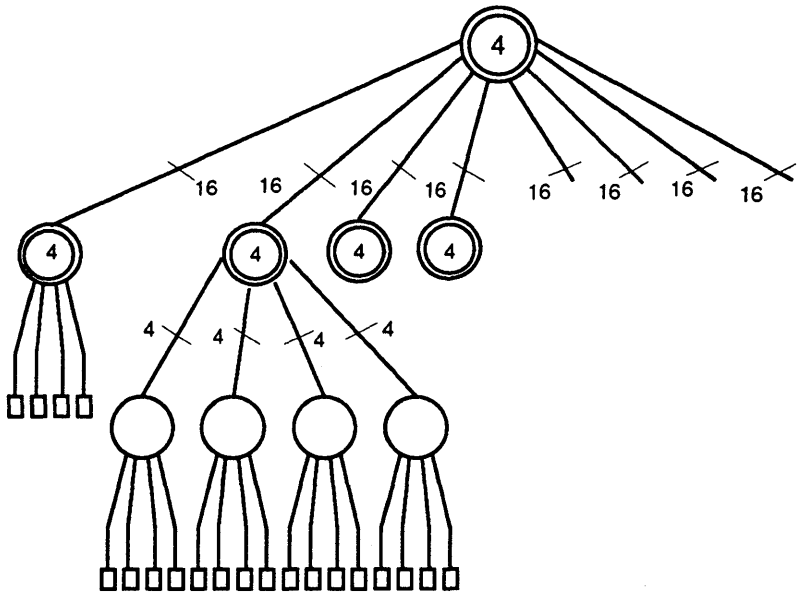
Figure 4-1 shows a fully populated 2 level CS-2 network. For this network the Elan Ids are allocated sequentially, the processor on the extreme left having Elan Id 0, and the processor on the far right having Elan Id 31 (its route being 8.3).

Figure 4-1 A 2 level network



Allocation of Elan Id's becomes more complex when SPARC/Compute boards and single SPARC boards are both installed in the CS-2 system. Modules populated with SPARC/Compute boards include one level of switching integral to the module, whereas modules with single SPARC processor do not. Networks consisting of mixed module types have processors at more than switch level.

Figure 4-2 A 3 level network with processors at mixed levels



In this case all network routes are expressed by three components; processors on level 2 have do not need the third component so it is always set to zero. This means that the single SPARC boards at level 2 have Elan Id's that increment by 4.

	Route	Binary Representation	Elan Id
Level 2 Processors	0.0.0	000 00 00	0
	0.1.0	000 01 00	4
	0.2.0	000 10 00	8
	0.3.0	000 11 00	12
Level 3 Processors	1.0.0	001 00 00	16
	1.0.1	001 00 01	17
	1.0.2	001 00 10	18
	1.0.3	001 00 11	19

Hostname, Elan ID, Ethernet Address Translation

Several functions are provided in the resource management system interface library, librms.a, for translation of processor hostname to/from Elan Id or Ethernet address. These functions are described in *Procedural Interface to the Resource Management System* on page 46.

The `rinfo(1)` command can also be used to provide a map from hostname to Elan Id via it's `-t` option. See *Hostname to Elan (Processor) Id Translation* on page 19.

Creating a Parallel Filesystem *A*

This appendix shows how to create a parallel filesystem that spans 2 CS-2 filesystems. The example uses four CS-2 processors: `cs2-1` and `cs2-2` host the two data filesystems, `cs2-0` hosts the map filesystem, and `cs2-5` will mount the PFS.

1. **Identify the data filesystems.**
   On both `cs2-1` and `cs2-2` create the directory `/pfs_admin/data`. Ensure that the data directory is available for remote mounting by other processors by using the `share`(1m) command.

   The following example shows the commands that are executed on `cs2-1` (repeat these on `cs2-2`):

```
cs2-1% su
password: root-password
cs2-1# mkdir /pfs_admin
cs2-1# mkdir /pfs_admin/data
cs2-1# share -Fnfs -orw /pfs_admin/data
cs2-1# chown bin /pfs_admin /pfs_admin/data
cs2-1# chgrp bin /pfs_admin /pfs_admin/data
```

In this example we have made the directory available for remote mounting by typing the share command at the prompt. You may prefer to add this command into `/etc/dfs/dfstab` so that the directory remains available if the machine is rebooted.

The name of the data directory is arbitrary, but the use of `/pfs_admin` is significant because this directory will be used by the `mount`(1m) command if we later choose to mount the PFS on this processor. By placing the data directory within `/pfs_admin` all the PFS files are kept in one place.

The `/pfs_admin/data` directory could be a mount point for a dedicated disk device, or simply a directory on your system disk.

2. **Identify the map filesystem.**
On `cs2-0` create the directory `/pfs_admin/map`. Ensure that the data directory is available for remote mounting by other processors by using the `share`(1m) command.

```
cs2-0% su
password: root-password
cs2-0# mkdir /pfs_admin
cs2-0# mkdir /pfs_admin/map
cs2-0# share -Fnfs /pfs_admin/map
cs2-0# chown bin /pfs_admin /pfs_admin/map
cs2-0# chgrp bin /pfs_admin /pfs_admin/map
```

Similar comments apply to the map directory as to the data directories. You may prefer to place the share command in the `/etc/dfs/dfstab` file, and the choice of name for the map directory is arbitrary (using `/pfs_admin/-map` keeps your filesystem neat). In general you need not place the map filesystem on a dedicated disk as it is unlikely to become large.

3. **Initialise the map filesystem.**
   You must first create a file that identifies the data filesystems that will be used by your PFS. In this example the file is called `/tmp/fslist`:

```
cs2-0# cat > /tmp/fslist
cs2-1:/pfs_admin/data
cs2-2:/pfs_admin/data
^d
```

   You initialise the PFS with `mkfs`(1m):

```
cs2-0# mkfs -Fpfs -onfs=2,fslist=/tmp/fslist,stride=8K,verbose,force \
/pfs_admin/map
```

   The `mkfs` command is executed on the host of the map filesystem (`cs2-0`). Note that `mkfs` creates a subdirectory called `/pfs_admin/mnt` which is used to temporarily mount the data filesystems while the PFS is being initialised. If `/pfs_admin` did not already exist (because the map filesystem is elsewhere) then it will be created.

   Note that the stride (in this case 8k) **must** be an integer multiple of the block size that is used by the underlying data filesystems.

4. **Mounting the PFS.**
   You mount a parallel filesystem with the `mount`(1m) command, specifying the map filesystem as the target for the mount. To mount the PFS on `cs2-5`, for example:

```
cs2-5# mkdir /pfs
cs2-5# mount -Fpfs -orw cs2-0:/pfs_admin/map /pfs
```

   You may mount the PFS on as many CS-2 processors as are necessary, including the hosts of the underlying data and map filesystems (if that is appropriate).

Access to the PFS filesystem is set using the permissions on the PFS directory (in this case `/pfs`) using `chmod`(1), `chown`(1), and `chgrp`(1) as required.

5. **Test the PFS.**
   Login to any of the processors that have mounted the PFS (but not as root). Confirm that you can create a file (`mkfile`(1) can be used to create large empty files) and that it is visible to all the other processors that have the PFS mounted.

   For example, with the PFS mounted on `cs2-5`, create a 10Kbyte file:

```
user@cs2-5: mkfile 10k /pfs/test5
```

   Assuming that the PFS is also mounted on `cs2-8`, check that the file is visible and that you can create a file from this processor too:

```
user@cs2-8: ls -l /pfs
-rw-r--r-- 1 staff 10240 Jun 8 06:55 test
user@cs2-8: mkfile 10k /pfs/test8
```

Computing Surface

Pandora User's Guide

S1002–10M125.01

meiko

Meiko's address in the US is:

Meiko
130 Baker Avenue
Concord MA01742

508 371 0088
Fax: 508 371 7516

Meiko's address in the UK is:

Meiko Limited
650 Aztec West
Bristol
BS12 4SD

Tel: 01454 616171
Fax: 01454 618188

Issue Status:

Draft	
Preliminary	
Release	x
Obsolete	

Circulation Control: *External*

Contents

Using Pandora 1

Introduction to Pandora

Pandora is a window onto the Computing Surface and can be used by both System Administrator and the users of a CS-2 system. There is little difference in these two classes of operation, the main exception being that only the System Administrator will be able to make changes in the overall system configuration. Both Administrator and user alike will be able to perform system query and monitoring operations.

Pandora is used to map the CS-2 resources into configurations that best match the requirements of different working groups using the machine. These configurations can then be edited by the System Administrator to mirror changes in those requirements as resource is added or removed, or as the use of the machine switches from daytime interactive operation to night-time batch operation.

It is also used to run applications, recognise and report system errors, and perform diagnostic tests on most aspects of the machine.

Starting Pandora

Pandora has a number of command line options to modify its behaviour:

-g	Use this option if you are using a grey scale display; Pandora requires a colour display by default.
-s	No shadows. Pandora usually draws shadows behind objects so that the display is more visually appealing. You may prefer to disable the shadows, particularly when visualising large systems.
-v	Verbose mode. Pandora writes diagnostic information to its command shell.

Pandora is normally executed as a background task from your command shell. You must ensure that your DISPLAY environment variable identifies your workstation's screen.

```
user@cs-2: /opt/MEIKOcs2/bin/pandora -v &

Pandora attached to Cluster: 0 module: 0 node: 12
```

Pandora responds by confirming connection to a CS-2 node and by displaying the root panel on your display. In the above example, Pandora is executed on processor 12, which is in module 0 in Cluster 0 (a Cluster is a 3 bay/24 module unit).

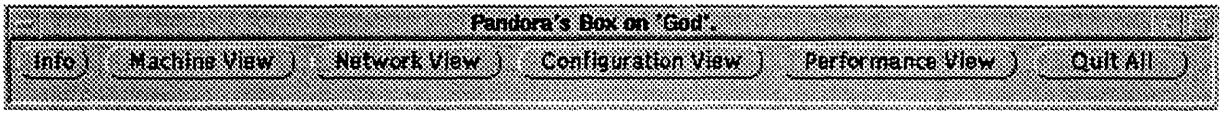
Interaction with Views

Pandora consists of a number of views, each representing some aspect of the CS-2 configuration or operation, and is manipulated by point-and-click operations through an X-Windows interface. This section explains how views are initiated and how objects within the views are manipulated.

The Root Panel

Views are selected from Pandora's a root panel, which will appear shortly after starting `pandora`:

Figure 1-1 The Root Panel



The root panel includes buttons for each of the supported views, a Quit button, and an Info button. A left mouse click on any of the view buttons will invoke that view. A left mouse click on the Info button will display Pandora's release version, and a brief revision history. The Quit All button is used to close all views and to shut-down Pandora.

View Control Panel

All of Pandora's views have a common interface; a control panel along the top of the view and a pop-up function menu. The buttons in the control panel have the following meanings:

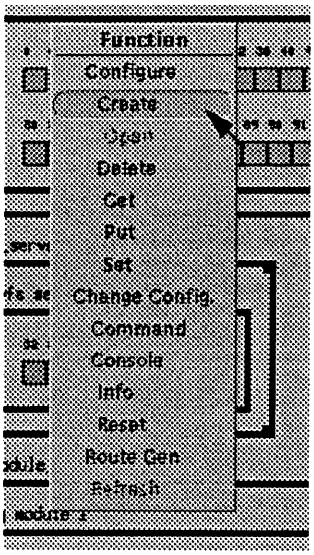
Color Space	This is a pull-down menu that allows you to select a Color Space for this view (more on this later) — the contents of this menu change for each view. To view the contents of this menu click and hold the right mouse button over the menu and drag the mouse down towards the foot of the screen. Release the mouse button when the pointer is over the selected entry. Each menu entry has a pull-out colour chart. To view this chart move the mouse to the right of the menu entry and then release the mouse button. This will cause the selected Color Space to become active and the reference chart to remain on screen.
Root Menu	This button is used to bring the Root Menu to the front of the window stack, making it visible if it has become hidden or closed into an icon. A click with the left mouse button activates this button.

Abort	This button is usually inoperative (and shaded-out) and becomes available only for a limited number of functions that have an abort facility.
Quit View	This button closes-down the view. A click with the left mouse button activates this button.

Function Menus

Each view has a pop-up function menu. To view this menu you must press and hold the right mouse button down while your pointer is within the view's display area. You select a function by moving the mouse over the desired entry and then releasing the button.

Figure 1-2 The Function Menu for the Configuration View



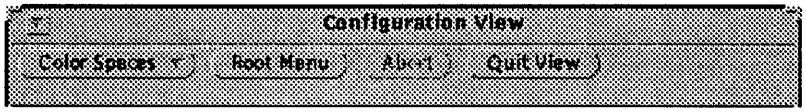
Iconising a View

Views can be closed to an iconic representation by a left-mouse click in the small square button at the extreme top-left of the view's frame, and can be re-opened by a double left-mouse click over the icon. Even when the views are iconified they still maintain state although changes to that state are not redrawn until the view is reopened.

Interface Conventions

The appearance of windows, buttons, and menus will conform to either the OPEN LOOK or MOTIF conventions. The functions assigned to each of your three mouse buttons will be subtly different for the two environments: for the OPEN LOOK environment the left-mouse button is used when making selections, such as pressing a panel button, the middle mouse button adjusts or extends a selection, and the right-mouse button is used for menu selection; in the MOTIF environment the left mouse button is also used to select a menu from a view's menu bar.

Figure 1-3 View Buttons



Also common to all views are the methods used for interaction and control, they are:

- Object selection and manipulation.
- Function selection and application.
- Function selection short-cuts.
- Color Space selection.
- Information viewer.

These are described in the following sections.

Object Selection and Manipulation

Once a view has been opened, objects can be created or manipulated within it. Some views, for example the Configuration View, will already have objects present when the view opens, others will need objects to be copied from other views.

Objects can be selected or grouped together in *Regions Of Interest* (ROI). A ROI is described by a box that is created by mouse operations. There are two methods that can be used in ROI creation, *object-clicking* and *rubber-banding*.

Object Clicking

A mouse click is a mouse button down followed immediately by a mouse button up. During the interval between these events the mouse location must not change. If a click occurs within the bounds of an object (processor, switch etc.) then an ROI is created and positioned around that object, and is signified by a change of the object's color.

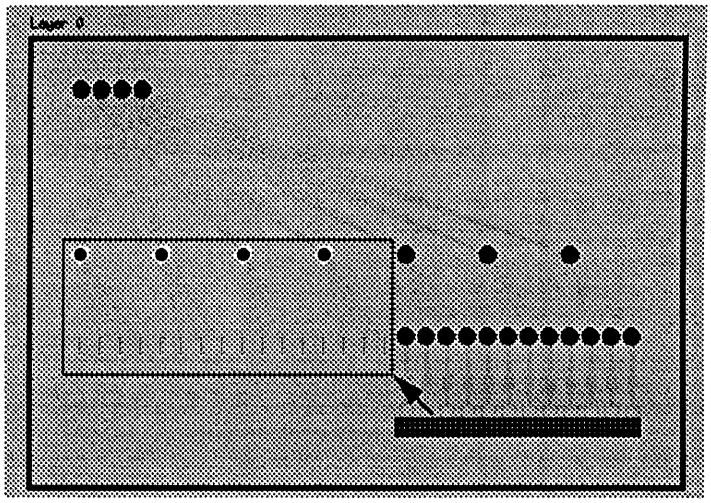
The left mouse button and the middle mouse button can be used for object clicking, each performing a different function. A left mouse click will reset any previously defined ROIs before creating one about the selected object. If no object was selected, that is if the operation took place outside the bounds of any object, the list of selected ROIs is effectively nulled. A middle mouse click will add or delete the selected object in the current ROI list. An ROI will be deleted from the current ROI list if the middle mouse click occurred within a previously defined ROI.

Rubber Banding

ROI creation can also be performed by rubber-banding. Rubber-banding is performed by dragging the mouse over a region with the left-mouse button depressed. The initial left-mouse down initialises the ROI and the following left-mouse up completes the operation. As the mouse is dragged the extent of the ROI is outlined by a box. When the ROI is complete, its screen extent will snap around any objects that were enclosed completely by the ROI. Unlike the left-

mouse click, which resets the ROI list, left-mouse rubber-banding always adds to the currently selected ROI list. These ROIs can be deleted by a middle-mouse click within the bounds of the ROI.

Figure 1-4 ROI Creation



Dragging and Dropping ROIs

Once objects have been selected by the creation of ROIs they can be manipulated by drag-and-drop operations. There are two types of drag-and-drop operation; the drag-and-add operation and the drag-and-find operation.

The drag-and-add operation allows ROIs to be dragged from one view and added into another. For example, when defining a new configuration you might drag the selected processors from a Machine View into the Configuration View where a new partition is being defined. The drag-and-add operation is defined by a left mouse down within an object while holding down the SHIFT key, followed by a mouse drag into the target view. During a drag-and-add operation the mouse icon changes to a tied sack.

The drag-and-find operation allows ROIs to be dragged from one view, and the objects they represent to be highlighted within the existing display of another view. For example, when tracing a faulty switch component you might drag the selected switch from the Network View into the Machine View so that's its physical placement in the machine can be identified. The drag-and-find operation is

defined by a middle mouse down within an object while holding down the SHIFT key, followed by a mouse drag into the target view. During a drag-and-find operation the mouse icon changes to a question mark.

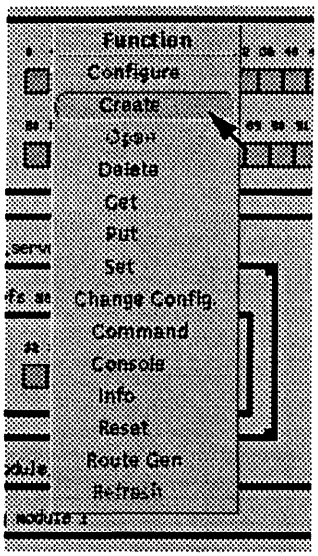
During a drag-and-drop type operation (of either type) all the ROIs in the selection will be outlined to indicate their inclusion in the drag-and-drop operation. The objects contained by the ROI list can be dragged for as long as the mouse button is depressed. When the button is released the objects will be dropped into the view. If the addition of the selected objects to the view is not understood, or the objects are dropped outside a view, then they are discarded and the drag-and-drop has no effect.

Function Selection and Application

All views have a common set of Function Menu items. Selectable items are drawn in solid text, invalid functions are 'greyed out' and are not selectable. The actual operations carried out as a result of function selection are in many cases view specific and a detailed description of each can be found in a the following chapters. The application of some view functions may take a number of seconds if the function is complex and is applied to a ROI list that contains many objects; for example querying the error state of all switches in a machine. During this time the cursor will change to a stopwatch, the clock in the bottom corner of the view will freeze, and the view will not allow editing of the selected ROI list. The cur-

sor will resume its pointer representation when the function has completed. The interaction with other views is not affected during this period, unless of course, further complex actions are attempted in those views as well.

Figure 1-5 Function Menu



The Function Menu is generated by a right-mouse down over the view's display area. Menu items can be selected by dragging the mouse over the items with the button depressed. To select an item (and therefore a function to apply) let the mouse button up over the desired selection; the function will then be applied to current ROI list.

Function Selection Short-cuts

Many of the items selectable from the Function Menu have keyboard short-cuts. A single keystroke will have the same effect as having opened the Function Menu, selecting an item, and applying it to the current ROI list. Keyboard short-cuts are view and Color Space specific, and are described in later chapters with the description of each view.

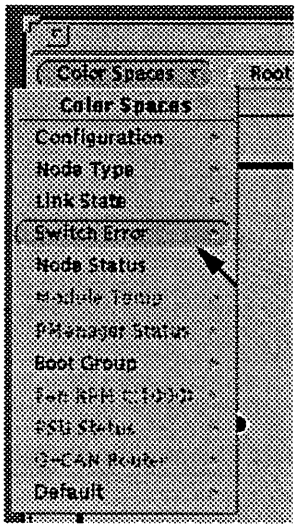
Color Space Selection

Pandora consists of a collection of views, each representing the CS-2 in a different way. A key feature of this tool is its ability to overlay more than one type of information in any of the views.

Color Spaces are used to add this extra dimension to the views. They are simply a mapping of some system attribute to a color range that is used to paint objects in the view. The ability to overlay Color Spaces provides a powerful query mechanism which can lead to interesting composite views that can be of great help when configuring and monitoring a system.

For example, the Network View represents a logical machine consisting of processors, switches and their interdependency. Switch and link errors are indicated here by the application of an appropriate Color Space. If, after running diagnostic tests, switches or processors need to be isolated or replaced, dragging them into the Machine View would quickly identify the board and module that contain the problematic element.

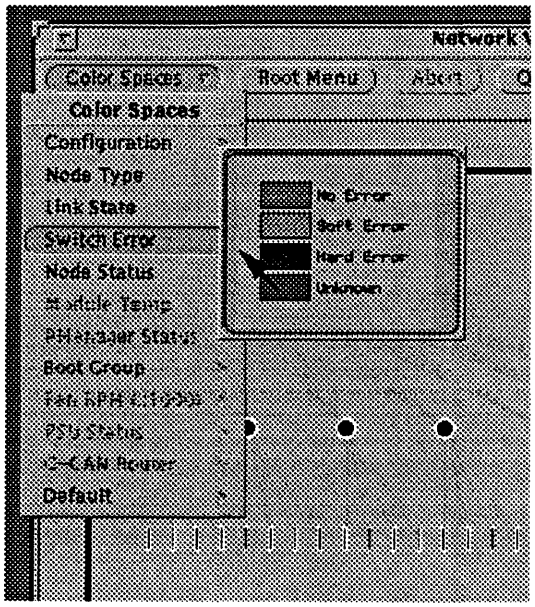
Figure 1-6 Pandora Color Space



Color Space selection is controlled from a Color Space Menu, which is activated by a pull-down menu in the view's control panel. This menu has pull-right items indicated by the arrow to the right of the menu items. Items and their attached Color Spaces are selected in the same manner as Function Menu items described above, the pull-right items are activated by dragging the mouse over an item's arrow before releasing the mouse button.

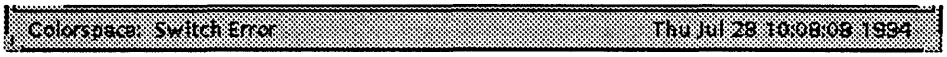
The pull-right item represents the entries of that particular Color Space and remains in the view when selected. These act as a reference key while a Color Space is in operation. The reference key need not be selected to change Color Space, selection from the main Color Space Menu is enough if you do not require the key.

Figure 1-7 Color Space Reference Keys



It is possible to have more than one reference key pinned to the workspace at any one time. In this case there is no need to return to the main Color Space menu to select one of these spaces; a left-mouse down over a pinned Color Space will select it as the active Color Space. The active Color Space is indicated in a frame footer.

Figure 1-8 Frame Footer (Network View)

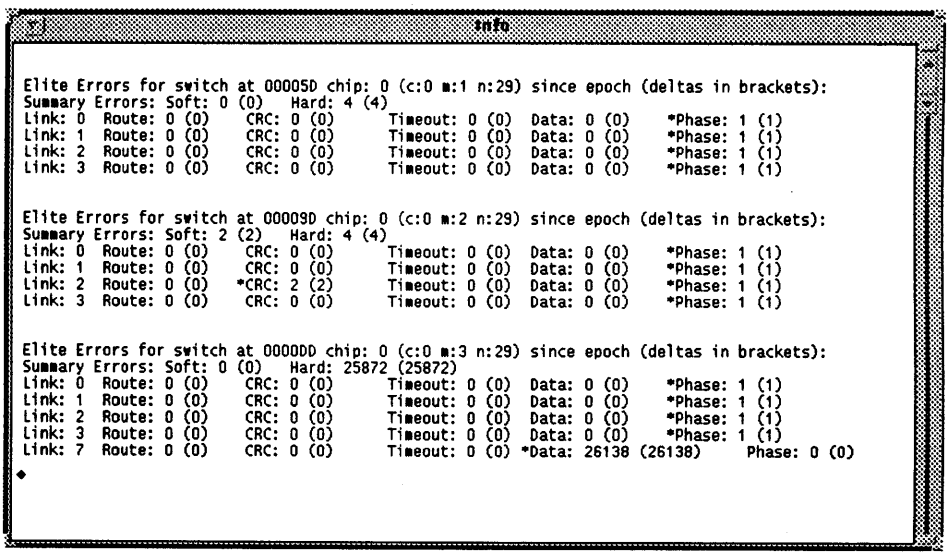


Information Window

All views have an Info function available from their Function Menu. The Information Window is a text window in which information about selected objects is written; this window is used when a more detailed textual description is required than can be represented by one of the Color Spaces. Pandora also uses the Information Window to display some of its diagnostic messages.

The Information Window is automatically created when information is first written to it. You can iconise the window by clicking on the button at the top left hand corner of the window, and open an iconised window by a double left mouse click on the icon. If information is written to the Information Window while it is iconised then the icon flashes to draw your attention.

Figure 1-9 Link Errors displayed in the Information Window



System Files

A number of files are either created by Pandora or used by the System Administrator/system software to pass information into Pandora. These files are:

- Defaults file.
- Event Log cache files.
- Geometry files.
- Route tables.

Defaults Files

At start-up Pandora reads the contents of the `defaults` file in `/opt/MEIKOcs2/etc/`*machine-name*`/pandora`. This file identifies default processor attributes; these are used in the Network and Configuration views by the Set function when no alternative values are explicitly stated within Pandora.

The `defaults` file is created by Meiko and should not be changed.

Event Logs

A number of log files are used by the Resource Management System to store resource events. The information in these files is used by Pandora to determine the state of the machine and to update its display as the state changes. The following files are used, all reside in `/opt/MEIKOcs2/etc/`*machine-name*:

- `machine.log` — machine events.
- `module.log` — module status.
- `board.log` — board status.
- `switch.log` — switch status.
- `proc.log` — processor status.
- `device.log` — device usage events.
- `config.log` — configuration events.
- `partition.log` — partition manager events.
- `resource.log` — processors being used or freed by user applications.
- `job.log` — user applications starting and stopping.

Note that these are binary files; to determine their contents you must use the `eventbrowser`(1) utility. Note also that these files are circular buffers of finite size; they will not consume large amounts of disk space.

Geometry Files

Pandora preserves the layout of the Configuration View between sessions by saving layout information into a file. The layout of partitions within each configuration definition is saved in a `config.geom` file alongside the configuration's definition (in the directory `/opt/MEIKOcs2/etc/`*machine-name*`/`*config-name*). Pandora uses a default layout if there is no `config.geom` file.

Route Tables

Route tables are generated by the Route Gen. function in the Configuration View. A number of files are created, one for each processor in the partition, which define routes from the processor to all other processors in the partition.

The files are created in a `routes` subdirectory alongside the partition's definition (in `/opt/MEIKOcs2/etc/`*machine-name*`/`*config-name*`/`*partition*). The name of each route file is the same as the Elan Id of the processor that it was generated for.

These routes may be installed in the processors' route tables by `rmsroute`(1m).

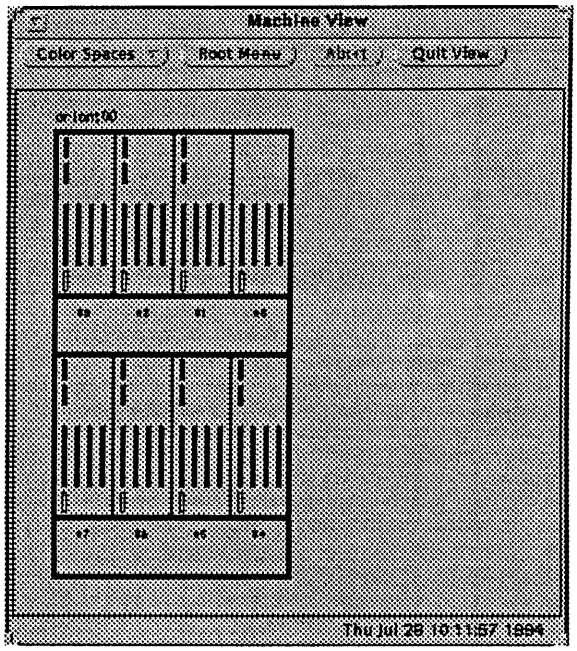
1

Machine View 2

The Machine View is a representation of the machine as it would appear if you were stood in front of it. This view is used for tracking objects selected in the more abstract views back to the physical machine, enabling the identification of pieces of hardware resulting from selection of objects such as processors and switches.

The modules are grouped in bays of up to 8 modules. Within each module small switch boards are shown in the top of the module, processor and large switch boards in the middle, and the module controller at the bottom.

Figure 2-1 Machine View (Default Color Space)



Color Spaces

The Machine View supports the following Color Spaces:

- Configuration — identifies modules that are configured-out.
- Module temperature — the temperature at strategic points within the modules.
- Fan speeds — the speed of the cooling fans in the module.
- Power supply status.
- G-CAN routing information.
- Default — shows processor, switch, and module control cards.

Configuration

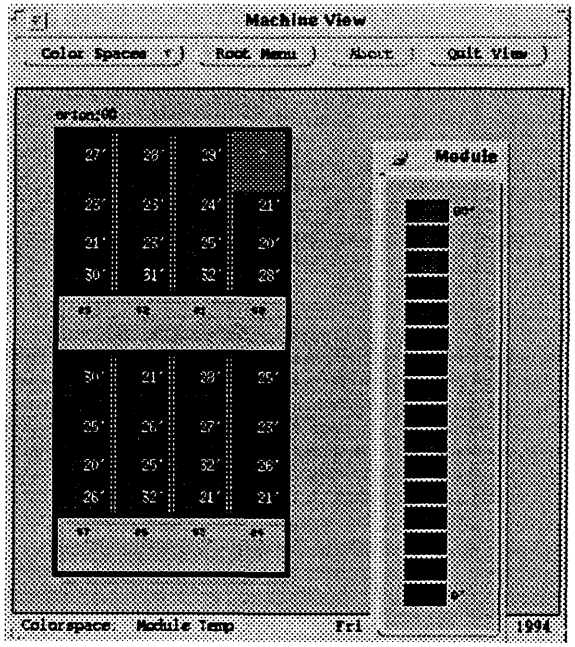
The Configuration Color Space compliments any one of the other Color Spaces. It shows objects that have been configured-out in grey; configured-in objects are coloured using the active Color Space.

Objects that are configured-out have been made unavailable. You can configure-out objects by using the Configure function, described later (see page 25).

Module Temp.

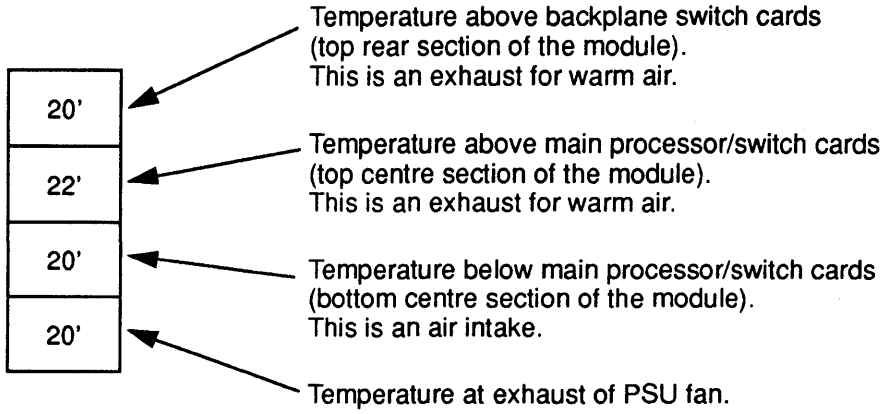
The Module Temperature Color Space shows the approximate temperature recorded by the 4 sensors that are fitted to each module. Overlaid onto the colour display are the 4 temperatures readings (in Celsius).

Figure 2-2 Pandora's Temperature Display



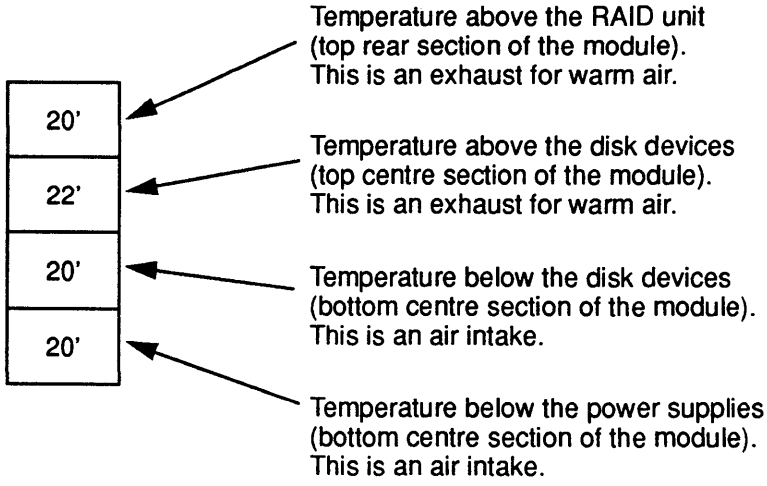
The mapping from displayed temperatures to the sensor positions in the processor and switch modules is shown below:

Figure 2-3 Temperature Display for Processor/Switch Modules



The mapping from displayed temperatures to sensors position in the disk modules is shown below:

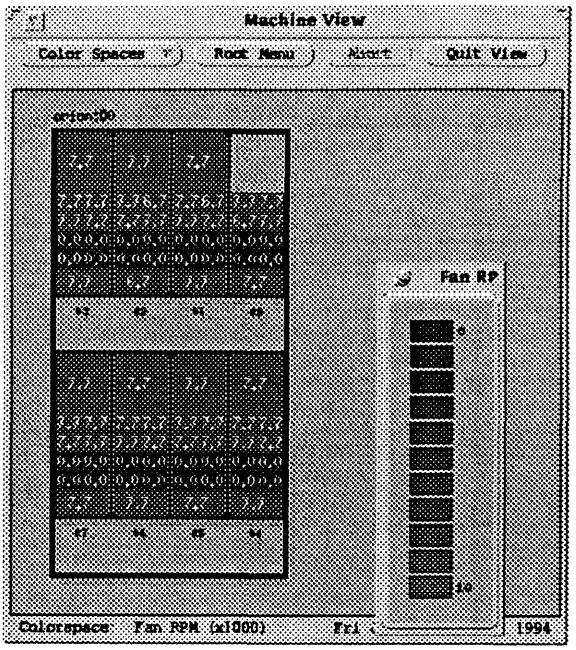
Figure 2-4 Temperature Display for Disk Modules



Fan RPM

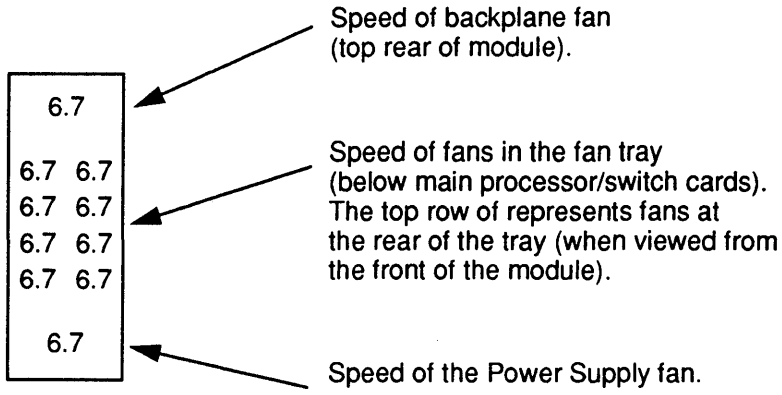
The Fan RPM Color Space shows the approximate speed (×1000 RPM) of the cooling fans in each module.

Figure 2-5 Pandora's Fan Speed Display



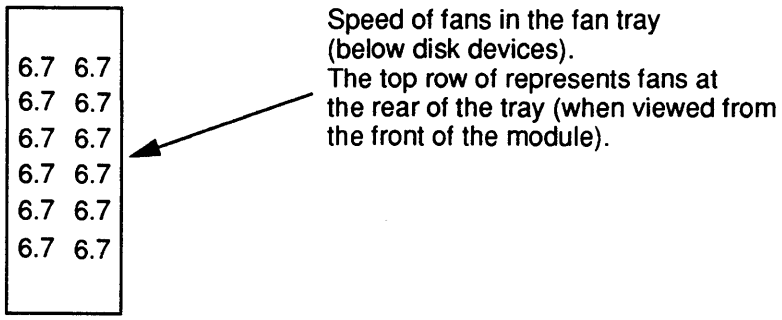
The mapping from displayed speeds to the fan positions in the processor and switch modules is shown below:

Figure 2-6 Fan Speed for Processor/Switch Modules



The mapping from displayed speeds to the fan positions in the disk modules is shown below:

Figure 2-7 Fan Speed for Disk Modules



PSU Status

The PSU Color Space displays the status of the power supplies as green (good) or red (error). For processor and switch modules (which contain a single power supply) the whole module is coloured according to the PSU status. For disk modules the display is divided into 3, with the top section representing the left power supply (when viewed from the rear of the module).

G-CAN Router

The G-CAN Router Color Space displays each module's configuration on the Global CAN bus (G-CAN).

Running throughout your CS-2 system is a Control Area Network (CAN) that is used to carry diagnostic and control information throughout the machine. It allows Pandora to determine the status of the principle CS-2 components, to reconfigure and reset those components, and to provide remote console connections to the Unix processors.

The limitations of the CAN hardware mean that a single network cannot be used to connect all the components of your system — instead a hierarchy of networks is used. At the lowest level the CAN controls objects within a module; the SPARC processors, the H8 processors, and the module controller all have interfaces to the CAN at this level. At the intermediate level there is the X-CAN, which connects the modules in a Cluster (3 bay/24 module system); the transition from module CAN to the X-CAN is handled by each module's controller. At the highest level is the global CAN, or G-CAN, which carries the network between Clusters. The interface between the X-CAN to the G-CAN is via one of more of the modules within each Cluster.

The nomination of a module as a G-CAN router can be made either by one of the Unix processors or the H8 processors. When using the G-CAN Router Color Space, Pandora identifies the G-CAN routers as shown below:

No G-CAN.	No global CAN bus for this system.
Unix control, always routing.	This module is a G-CAN router.
Unix control, never routing.	This module is not a G-CAN router.

H8 control, routing.	The H8 controls G-CAN routing.
H8 control, potential router.	This module is able to become a router.
H8 control, not routing.	This module is not able to become a router.

Default

The Default Color Space is selected when the Machine View is first displayed. It uses a different colour for each of the 3 board types: processors, switches, and module controllers. The Machine View shown in Figure 2-1 uses the Default Color Space.

Keyboard Short-cuts

The following keyboard short-cuts operate in the Machine View; type the character with the mouse-pointer in the view's display area. All the short-cuts execute functions from the Function Menu (described below).

Key	Function	Description
`i`	`Info`	List information about the selected objects.
`s`	`Set`	Set the various object attributes.

Function Menu

The function menu is viewed by pressing and holding the right mouse button while the mouse is within the view's display area. The following functions are supported in the Machine View:

- Configure — change configuration state.
- Set — set module attributes.
- Info — get information about objects.
- Reset — reset processors.
- Refresh — redraw the display.
- Finder — locate objects using specified search criteria.

Configure

This function is used to change the configuration state (availability) of processors and switches. The ROI should include processor and/or switch boards; if the ROI includes a module then the function is applied to all processors and switches in that module.

Selecting this function causes a dialogue box to appear allowing the objects in the ROI to be configured-out (or configured back in). An additional option allows you to change the link state of the links that connect to the selected objects. Select the required options and use Apply to make them take effect; use the Ignore button to dismiss the dialogue box without changing configuration state.

When configuring-out an object you should enable the link state change (unless you intend to change the state manually from the Network View). Setting the link state causes the link to be put into an Acking state (i.e. inoperative, but acknowledging packets before consuming them) — this is usually the preferred state because it allows broadcasts to include the range of configured-out objects without causing the broadcast to fail. When configuring-in objects you should enable the link state change to restore the links to an operative state. In this case the link is put into a Nacking state for 20 seconds before being put under Auto (H8 control); this ensures that the link is in a coherent state before being made available.

Note that confirmation of link state changes is written to Pandora's Information Window, and can be visualised with the Link State Color Space in the Network View.

See also the Set function in the Network View.

Set

This function is used to set various module attributes. The ROI must include modules.

The following options appear within a dialogue box. Select the required options and use the Apply button to make them take effect. Use the Ignore button to dismiss the dialogue box without changing the module's attributes.

GCAN router control
Options are: `Ignore`, `Auto (H8 control)`, `No GCAN`, `Always route`, `Never route`. This attribute defines the module's connection to the G-CAN. The normal setting is `Auto`; `ignore` means that the current setting is to remain unchanged.

Set GCAN router id
Options are `no` or `yes`. If specifying yes you must list the G-CAN id of the network that this module connects to. Remember that several G-CAN networks can be present in a system. Numbering of the network begins at 0.

Read module NVRAM
Options are `no` or `yes`. Selecting yes causes the module controller's NVRAM values to be written to Pandora's Information Window. The contents of the NVRAM are useful only to Meiko engineers for diagnostic purposes.

Flush to NVRAM
Options are: `no` or `yes`. Use this option to write the switch state to the module NVRAM.

Thermistor lower trigger
Enter a minimum permitted operating temperature (in Celsius). A level 1 alert is generated if any sensor records a temperature lower than this setting. Note that alerts are recorded by Pandora but currently have no other effect.

Thermistor upper trigger
Enter a maximum permitted operating temperature (in Celsius). A level 2 alert is generated if any sensor records a temperature that is higher than this. Note that alerts are recorded by Pandora but currently have no other effect (although safety devices within the module will shutdown the unit if the temperature becomes unsafe).

Fan lower trigger
Enter a minimum number of fans; if the number of operational fans falls below this limit then an alert is generated (but note that these alerts are not currently used).

Info

This function displays information about boards and modules; the information is written into Pandora's Information window. The ROI must include boards or modules.

For boards the following information is displayed:

- The board's configuration status (configured-in/out).
- The board's CAN address.
- The board's logical id within the module (it's slot id).
- The board type (e.g. MK515, MK401 etc.).
- The ROM revision for the board's H8 controller.
- The board's module id.
- The number of processors on the board.
- The number of switches on the board.

When used to obtain information about a module the following information precedes the descriptions of the module's boards:

- The module's configuration status (configured-in/out).
- The module's CAN address.
- The module's type (e.g. processor or switch).
- The switch level that this module connects to, and the module's network id.
- The Elan Id of the first processor in the module, and the number of processors.
- The thermistor and fan trigger levels.
- The module's power supply status.

Reset

Used to reset a board's H8 processor, or to reset a module. The H8 processor is principally used for controlling the board's interface to the CAN bus. Reset a board/module H8 if the CAN interface does not work as expected.

If the ROI includes a module then a dialogue box allows you to reset the module (temporary power-down), reset the module controller's H8 processor, or reset the module controller's H8 NVRAM (used to hold state and diagnostic information for the module).

If the ROI includes a module controller then the dialogue box allows the controller's H8 or H8 NVRAM to be reset.

If the ROI includes a processor or switch board then the dialogue box allow's the board's H8 processor to be reset.

Refresh

Used to redraw the display.

Finder

Used to locate objects according to the specified search criteria. The objects that match the search criteria are shown highlighted in the Machine View.

This function generates a dialogue box allowing you to search for Boards, Switches, or Processors. Selecting one or more of these options produces additional prompts that allow you to specify the search criteria. You may supply information to as may of these prompts as are necessary. The criteria are applied using a logical AND within each object type, and a logical OR between object types. Where a numeric search criteria may be entered it is often possible to enter either a numeric range (e.g. 1-4), or a comma separate list of ranges (e.g. 1-4,7,9-10).

Criteria that may be applied to **Boards** are:

- Board type — matches board type.
- ROM revision — matches H8 ROM revision.

- Number or procs — matches boards with the specified number of processors (a number range such as 1-4 matches boards with 1, 2, 3, or 4 processors).

Criteria that may be applied to **Switches** are as follows. Note that processor boards will be highlighted if the search criteria identifies switches that are fitted to those boards.

- Switch level — matches switches at the specified level (0 is uppermost).
- Switch plane — matches switches at the specified plane (0 is uppermost).
- Switch layer — matches switches in the specified layer (either 0 or 1).
- Net Id. — a network id (or range of Ids).

Criteria that may be applied to **Processors** are:

- Memory MB — matches boards fitted with specified memory (a number range such as 32-64 matches boards with between 32 and 64MBytes).
- Elan Id — a network Id (or range of Ids).
- ROM revision — matches boot ROM revision.
- Host name — locates the processor with the specified hostname.
- Boot device or host — used to identify a server processors (the other search criteria identify one or more client processors).

Interaction with Other Views

You can drag the following objects from the Machine View to other views:

- You can drag processor and switch boards into the Network View to determine their placement in the CS-2 network, and to determine their operational status. You can drag modules into the Network View to determine the placement or status of all the switches and processors in the module.
- You can drag processor boards into the Configuration View to determine the Elan Id of the processor's on that board, or to determine their operational status. You can drag modules into the Configuration View to determine the Elan Ids or status of all the processors in the module.

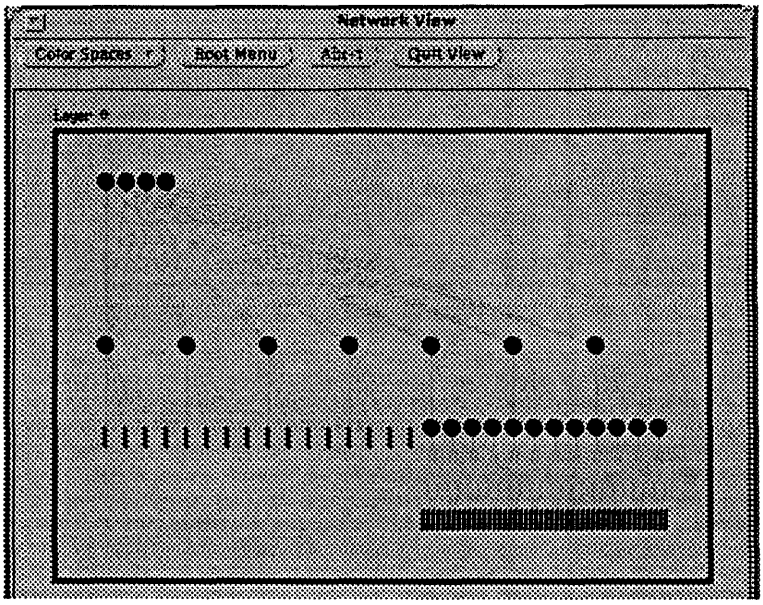
You can also drag objects from other views into the Machine View:

- You can drag processors and switches from the Network View to determine their physical placement (useful when the Network View identifies network faults).
- You can drag processors and partitions from the Configuration View to determine the physical placement of the processors (useful when the Configuration View identifies a faulty processor).

Network View 3

The Network View represents the CS-2 logically as a fat-tree diagram with switches at the tree nodes and processors at the leaves. This view is capable of showing every processor, switch, and link in a machine and is used primarily for system maintenance and diagnostic purposes.

Figure 3-1 Network View (Default Color Space)



Color Spaces

The Network View supports the following Color Spaces:

- Configuration — identifies processors/switches that are configured out.
- Node type — identifies processor types.
- Link state — status of Elan/Elite network links.
- Switch error — error conditions on the Elite network switches.
- Node status — the boot status of the processors.
- Boot group — identifies servers and clients.
- Default — identifies switches and processors.

Configuration

The Configuration Color Space compliments any one of the other Color Spaces. It shows objects that have been configured-out in grey; configured-in objects are coloured using the active Color Space.

Objects that are configured-out have been made unavailable. You can configure-out objects by using the Configure function, described later (see page 37).

Node Type

The Node Type Color Space identifies the types of processor in your network. This will currently be one of the following:

Viking	Texas Instruments Viking Processor
Viking + Ecache	TI Viking processor with external (2nd level) cache.
Pinnacle	ROSS Pinnacle.
605	ROSS 605.
VPU	Dual Fujitsu vector processors.
cVPU	Cache coherent dual vector processors.

Link State

The Link State Color Space identifies the state of all the links in the network. The link state display is updated by the Get function (described later on page 38) and is limited to the links in the ROI.

For the purposes of this view each link is divided into two equal lengths, each half belonging to the object that it connects to. The link may therefore be drawn in two colours to show the error status recorded by the component at each end.

The following link states are shown by this Color Space:

Active	Link is running normally without timeout.
Active with timeout	Link is running normally with a timeout specified. The timeout applies to packets waiting at a switch input for the output link to become available. This timeout is an attribute of the switch.
Nacking	Link is inoperative; packets will be Nacked and consumed as they reach the switch's output link.
Acking	Link is inoperative; packets will be Acked and consumed as they reach the switch's output link.
Reset	Held in reset (maybe the attached processor has been removed).
Unknown	Indeterminate state.

H8 controlled links are shown by solid lines — this is the normal state. Explicit controlled links are dashed — these are link's that have had their state changed explicitly by the System Administrator. Link states may be explicitly changed by either by using the Set function or as a side effect of configuring-out processors or switches — see the description of the Set function on page 40, and the Configure function on page 37.

Note that the Set function includes a Drawing Mode option specifying how this view is updated. When querying the state of several (possibly overlapping) ROIs you will need to adjust the drawing mode so that each query forces the results of a previous query to be erased from the display, otherwise the display may become difficult to interpret.

Switch Error

The Switch Error Color Space identifies errors on the switches. The switch error display is updated by the Get function (described later on page 38) and is limited to the switches in the ROI.

For the purposes of this view each link is divided into two equal lengths, each half belonging to the object that it connects to. The link may therefore be drawn in two colours to show the error status reported by the components at each end. In addition the color of the switch icon itself is changed, and included within the icon there maybe an S and/or H to indicate the error type (either Soft or Hard errors).

Switch error states are:

No error	The desirable operating state.
Soft error	Route errors, CRC errors, or timeout errors on the link.
Hard error	Phase errors (sender/receiver frequency drift or noise on clock line), or data errors.
Unknown	Indeterminate condition

Note that the Set function includes a Drawing Mode option specifying how this view is updated. When querying the state of several (possibly overlapping) ROIs you will need to adjust the drawing mode so that each query forces the results of a previous query to be erased from the display, otherwise the display may become difficult to interpret.

Node Status

The Node Status Color Space identifies the boot status of the processors. The status will be one of:

Configured Out	Unavailable.
Powered down	Powered down.
Inactive	Indeterminate status.
Needs FSCK	Needs a filesystem check with `fsck`(1m).
At 'ok'	At boot ROM prompt.

Self test	Performing start-up self test.
ROM loading code	Loading kernel code.
ROM running code	Running kernel code.
CAN driver loaded	CAN device driver installed.
Unix,RLS single user	Unix single user mode.
Unix,RL1 Root FS only	Unix run level 1, only root mounted.
Unix,RL2 Multi-user, no NFS	Unix run level 2, local filesystems mount.
Unix,RL3 multi-user	Unix run level 3, multi user with NFS.
Unix,RL0 halting	Unix run level 0, halting.
Unix,RL5 reboot -a	Unix run level 5, interactive reboot.
Unix,RL6 reboot	Unix run level 6, rebooting.

Boot Group

The Boot Group Color Space identifies each processor as either a boot-server or a client. Clients have no attached system disk of their own and rely on a server processor to host their root filesystem and to source the operating system code.

Note that MK405 (quatro) boards are always configured as clients because they have no disk capability.

Default

The Default Color Space is selected when the view is first displayed. It uses a different colour for the processors and switches. The Network View shown on page 31 uses the default Color Space.

Keyboard Short-cuts

The following keyboard short-cuts operate in the Network View; type the character with the mouse-pointer in the view's display area. All the short-cuts execute functions from the Function Menu (described below):

Key	Function	Description
g	Get	Update the display of Link Status or get a Switch Error context. Requires the Link State or Switch Error Color Spaces.
s	Set	Set link or processor attributes.
i	Info	Write information about processors and/or switches to Pandora's Information Window.

Function Menu

The function menu is viewed by pressing and holding the right mouse button while the mouse is within the view's display area. The following functions are supported in this view:

- Configure — change object's configuration state.
- Open — change style of network display.
- Get — restore a switch context or force update of Link Status display.
- Put — save a switch context to a file.
- Set — set link or processor attributes.
- Info — write information about processors or switches to the info window.
- Reset — reset processors or switches.
- Refresh — redraw the display.
- Finder — locate objects using specified search criteria.

Configure

This function is used to change the configuration state (availability) of processors and switches. The ROI should include processors and/or switches.

Selecting this function causes a dialogue box to appear allowing the objects in the ROI to be configured-out (or configured back in). An additional option allows you to change the link state of the links that connect to the selected objects. Select the required options and use Apply to make them take effect; use the Ignore button to dismiss the dialogue box without changing configuration state.

When configuring-out an object you should enable the link state change (unless you intend to change the state manually with the Set function). Setting the link state causes the link to be put into an Acking state (i.e. inoperative, but acknowledging packets before consuming them) — this is usually the preferred state because it allows broadcasts to include the range of configured-out objects without causing the broadcast to fail. When configuring-in objects you should enable the link state change to restore the links to an operative state. In this case the link is put into a Nacking state for 20 seconds before being put under Auto (H8 control); this ensures that the link is in a coherent state before being made available.

Note that confirmation of link state changes is written to Pandora's Information Window, and can be visualised with the Link State Color Space.

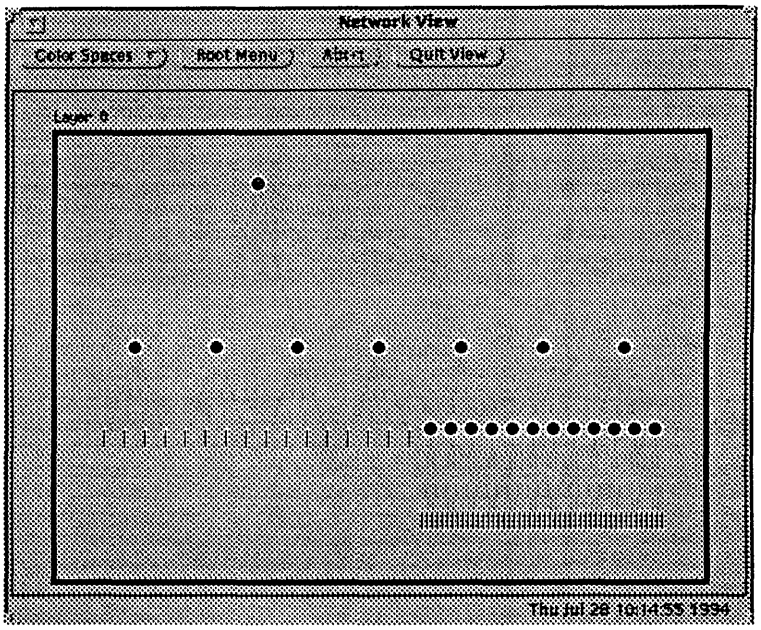
Open

This function is used to iconify groups of switches in order to make the view less congested. The ROI must include one or more switches.

Drawing the entire switch network of a large machine results in an image that is complex. To simplify this view groups of switches can be closed, so that switches at a common level and netId stack up, one behind the other, reducing the number of lines required to represent links.

Link state and errors are represented in order of their importance, that is, a link that shows a hard error will always overdraw one with a soft error, which in turn will always overdraw a link with no errors. The resolution of which switch owns the link in error can be achieved by opening that group of switches. Switch groups that have been closed are drawn differently to those which are open in that their representation changes from a single circle to concentric rings upon close.

Figure 3-2 Opened Network Display (compare with Figure 3-1)



Get

This function is only operative for the Link State or Switch Error Color Spaces.

Link State Color Space

When used with the Link State Color Space and an ROI that includes at least two connected objects (either processors or switches), this function forces Pandora to re-read the status of the connecting links and to redraw them according to the Link State Color Space. Typing the letter 'g' in the display area will also execute this function.

Switch Error Color Space

Warning – Note that the behaviour of this function when invoked via the keyboard is different to its behaviour when invoked via the function menu.

When used with the Switch Error Color Space, this function operates on *Switch Contexts*. A switch context is an error report taken at a specific time during the machine's operation.

Typing the letter 'g' into the display area will cause the current switch context to be fetched — the results are displayed within the Network View using the Switch Error Color Space, and are also written to the Information Window. The information that is displayed in the Network View indicates the errors that were reported since the last query. The information listed to the Information Window is more precise, and lists both absolute and relative error counts in addition to the error types. The following error reports from an Elan processor and an Elite switch were cut from the Information Window:

```
Elan Errors for proc at 000341 elanId: 0x51 (c:0 m:13 n:1) since epoch (deltas in
brackets):
Input: 0 (0) *Output: 4 (1)
CRC before ACK: 0 (0) EOP before ACK: 0 (0)
CRC after ACK: 0 (0) EOP after ACK: 0 (0)
Link0: 0 (0) Link1: 0 (0)
```

```
Elite Errors for switch at 00031D chip: 1 (c:0 m:12 n:29) since epoch (deltas in
brackets):
Summary Errors: Soft: 8 (8) Hard: 8 (8)
Link: 0 Route: 0 (0) *CRC: 1 (1) Timeout: 0 (0) Data: 0 (0) *Phase: 4 (4)
Link: 1 Route: 0 (0) *CRC: 4 (4) Timeout: 0 (0) Data: 0 (0) *Phase: 3 (3)
Link: 3 Route: 0 (0) CRC: 0 (0) Timeout: 0 (0) Data: 0 (0) *Phase: 1 (1)
Link: 4 Route: 0 (0) CRC: 0 (0) *Timeout: 3 (3) Data: 0 (0) Phase: 0 (0)
```

In some cases it is desirable for the switch error display to be relative to some other time frame, and not the last time it was examined. For example, you may wish to query the switch errors throughout the working day, or during the execution of a program, and on each occasion you wish the display to be relative to the start of the day or program, and not relative to the last query.

In this case you need to save a switch context at the start of the event (programme, day, or whatever) and to restore it just before each switch error query. To save a switch context you must first fetch it (by selecting a ROI and typing 'g'), and then save it to a file with the Put function. To retrieve a switch context you fetch it from the file with the Get function, and then update the display by typing 'g'. Having restored the switch context you can get an error report that is relative to it by typing 'g' for a second time.

Set

This function is used to set either link or processor attributes; the ROI must include either switches or processors (but not both).

Processor Attributes

If the ROI includes only processors then the following attributes can be specified. In the absence of user specified values those specified in Pandora's `defaults` file will apply; specifying `ignore` causes the current setting to remain unchanged.

Auto boot
Options are: `Ignore`, `False`, or `True`. When set to true the processor will automatically reboot when the processor is Pulse Reset (see the Reset function); if set to false the processor will remain at the Open Boot prompt awaiting a manual reboot.

Boot device
Options are: `Ignore`, `Elan`, `Disk`, or `Net`. Specify the boot device for this processor. Elan means boot from a boot server via the Elan network, disk means boot from the local system disk, and net means boot from Ethernet.

Boot Args
Options are: `Ignore`, `noargs`, `-s`, `kadb`, `kadb -s`, `kadb -r`. Specifies the default arguments used when rebooting.

Console Device
Options are: `Ignore`, `keyboard/screen`, `CAN`, `ttyA`, `ttyB`. Used to specify the connection used by this processor's console. keyboard/screen means the graphics terminal and keyboard, CAN means the CAN bus, and ttyA/B means the serial ports. You should select CAN if you intend to allow Pandora to create a remote console connection to this processor.

Enable Console stealing
Options are: `no` or `yes`. Only one console connection may exist to any one processor. With console stealing disabled (set to no) requests for a console will be refused if a console connection exists elsewhere. If console stealing is enabled the existing connections will be terminated immediately to allow the new connection to be created.

Auto Configure Boot params
Options are: `no` or `yes`. If enabled (set to yes) allows the processor's Elan Id and other per-processor values to be allocated automatically. See Network Options, below.

Elan Protocol Options — various
A number of Elan options can be specified; these are intended for Meiko engineers only and should not be adjusted.

Network Options — various
A number of network options can be specified, including the processor's Elan Id, it's position in the switch network, and the id of it's boot server. Typically these are assigned automatically by specifying the Auto Configure Boot Params option, as described above.

Switch Attributes

If the ROI includes switches the following attributes can be set:

Broadcast top
Options are: `ignore`, `sane`, `3`, or `7`. Identifies the highest link number on the switch that points down in the network (towards the processors). Usually links 0–3 point down, although switches at the top of the hierarchy may have all 8 links going down. The sane option forces an auto-configuration.

Switch timeout
Options are: `ignore`, `off`, `22-29us`, or other times. This specifies the timeout for message packets that are queued at a switch input (these will be waiting for an output link that is currently in use by some other message).

Set linkstate
Options are: `ignore`, `Auto (H8 control)`, `Active`, `Nacking`, or `Acking`. The Auto option means that the state of links directly connected to an Elan communications processor is determined by the processor's heartbeat signal; the link state is active while the heartbeat is present, and Nacking if the heartbeat stops (for other links the Auto option is equivalent to Active).

The remaining options allow the System administrator to specify the link state (but note that the switch itself will always override user specified states). Use the Nacking option for links that are inoperative; packets leaving the switch

output will be Nacked and consumed. Use the Acking option for inoperative links within a broadcast range; packets leaving the switch output will be acked (allowing the broadcast to succeed) and consumed.

Set linkstate method
Options are: `common link selection`, or `bitmap selection`. This attribute specifies how links are identified to the Set Linkstate attribute (described above). Common link selection means that the link is identified by selecting the components at each end within the ROI. The bitmap selection means that only a switch need be included in the ROI; the link is identified by the Link selection attribute described below.

Link selection (if bitmap selection)
Options are: `0`, `1`, `2`, `3`, `4`, `5`, `6`, or `7`. Used with the Select Linkstate Method described above. This attribute identifies a link.

Enable T/O errors on N&G
Options are: `no` or `yes`. When enabled this options causes timeout errors on Nacking links to be recorded. Timeout errors occur on packets waiting at a switch input that have been unable to connect with the required switch output for more than the switch's timeout period. When the connection time's out the packet is Nacked and Gobbled (consumed) — i.e. N & G.

Flush to NVRAM
Options are: `no` or `yes`. Use this option to write the switch state (such as Broadcast Top) to the module NVRAM.

Perform boundary scan
Options are: `no` or `yes`. The ROI must include both ends of a link. This performs a simple connectivity test to validate the links. The output is written to the Information Window.

Perform loop-back scan
Options are: `no` or `yes`. This test requires a loop-back connector. This test is for use by Meiko Engineers only.

Loop-back links to test
Options are: `0`, `1`, `2`, `3`, `4`, `5`, `6`, or `7`. For use with the loop-back scan test (for Meiko Engineers only).

Gather Outputter Blocked Counts

Options are: `no` or `yes`. This generates a count for each of the switches in the ROI of the number of packets that could not be directly routed to a switch's output link (possibly because it was temporarily in use by some other message). The view is updated so that links with low blocked counts are drawn in green, and links with high blocked counts are shown in red. Additional information is also written to the Information Window. Useful for identifying points of congestion in the network.

Perform switch performance metric

Options are: `no` or `yes`. Used to test the switch's performance against it's design specification. This test is for Meiko Engineer's only.

Switch LED on

Options are: `ignore`, `packets`, `errors`, `congestion`, `congestion/blocked`. For use with switches that are fitted to large switch cards (MK523, MK529 etc.). Determines the circumstances in which the status lights on the board's front panel are illuminated. The light is toggled for each instance of the specified event.

Drawing mode

Options are: `Add selection to current view`, or `Clear view`. Used to specify the display method used for the results of network functions. When the ROI applies to a subset of objects in the display area most network functions will only change the coloring of the objects in the ROI. When several function calls have been made, each to a different and possibly overlapping ROI, it is possible that the display will become unclear. By setting this attribute to Clear View the display is cleaned before the results of each function is displayed.

Info

This function lists information about the selected switches or processors in the Information Window. The information displayed is the current settings of the switch or processor attributes that are specified with the Set function (see above).

Reset

Used to reset switches or processors.

Processor Reset

If the ROI includes processors then a dialogue box asks for confirmation. A second dialogue box appears offering the following reset options:

Send Break
Halt the specified processors immediately. For use only with diskless client processors.

Hold in Reset
Hold the processors in a reset state.

Pulse Reset
Uses the processor's Auto Boot attribute (see the Set function) to determine whether the processor is to be booted following the reset.

Halt Procs
Equivalent to an `init 0` command. A graceful shutdown of the specified processors.

Boot Procs
Used to boot the specified processors.

Switch Reset

If the ROI includes switches the following options are presented in a dialogue box:

Switches
Options are: `Ignore` or `Reset`. Use this option to reset a switch when the switch becomes inoperative and other linkstate changes fail to restore it to an operative state.

H8s
Options are: `Ignore` or `Reset`. Reset the H8 that controls this switch.

H8 NVRAM
Options are: `Ignore` or `Reset`. Reset the NVRAM in the module controller clearing all state. This has the side effect of power cycling the module.

Refresh

Redraw the display.

Finder

Used to locate objects according to the specified search criteria. The objects that match the search criteria are shown highlighted in the Machine View.

This function generates a dialogue box allowing you to search for Boards, Switches, Processors, or a specified Network Route. Selecting one or more of these options produces additional prompts that allow you to specify the search criteria. You may supply information to as may of these prompts as are necessary. The criteria are applied using a logical AND within each object type, and a logical OR between object types. Where a numeric search criteria may be entered it is often possible to enter either a numeric range (e.g. 1-4), or a comma separate list of ranges (e.g. 1-4,7,9-10).

Criteria that may be applied to **Boards** are as follows.

- Board type — matches board type. When the search targets a Quatro the result will identify both the processors and the switches that are fitted to the board.
- ROM revision — matches H8 ROM revision.
- Number or procs — matches boards with the specified number of processors (a number range such as 1-4 matches boards with 1, 2, 3, or 4 processors).

Criteria that may be applied to **Switches** are:

- Switch level — matches switches at the specified level (0 is uppermost).
- Switch plane — matches switches at the specified plane (0 is uppermost).
- Switch layer — matches switches in the specified layer (either 0 or 1).
- Net Id. — a network id (or range of Ids).

Criteria that may be applied to **Processors** are:

- Memory MB — matches boards fitted with specified memory (a number range such as 32-64 matches boards with between 32 and 64MBytes).

- Elan Id — a network Id (or range of Ids).
- ROM revision — matches boot ROM revision.
- Host name — locates the processor with the specified hostname.
- Boot device or host — used to identify a server processors (the other search criteria identify one or more client processors).

The **Routes** option allows you to define a route through the network in terms of its start point (a processor Id) and a route through each network component. When you apply this option all components along the route are highlighted. You can use the Get function with the LinkState color space to determine the state of the selected route.

- Start Elan Id — the network address of the processor at the start of the route.
- Route string — a comma separated list of Elan/Elite links. The first number represents an Elan link and will be either 0 or 1 (i.e. layer 0 or 1). The remaining numbers will be Elite links in the range 0–7; these identify the link by which the route leaves the switch. Conventionally links 0–3 connect to lower network levels and 4–7 to upper levels. Top switches are configured with all links connecting to lower levels.

Interaction with Other Views.

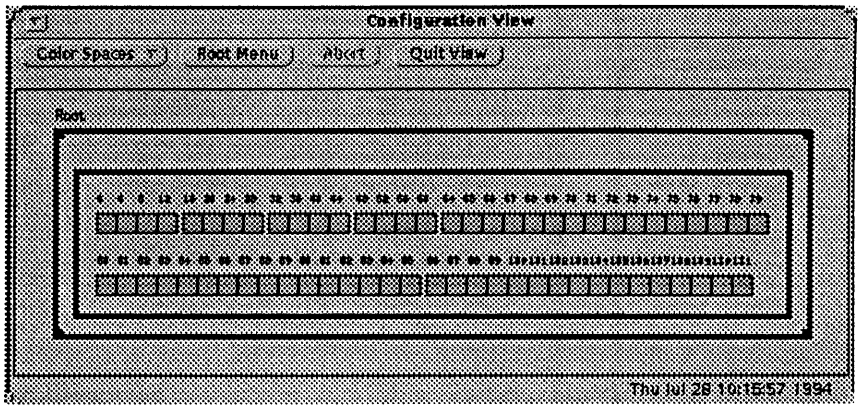
You can drag the following objects from the Network View to other views

- You can drag processors or switches into the Machine View to determine their physical placement. This is useful when a fault is identified and the appropriate component must be removed and inspected.
- You can drag processors into the Configuration View to determine which partition they are in.

Configuration View 4

The Configuration View, as its name suggests, is primarily used to configure the machine into resource groups. These resource groups or *partitions* will become the targets for users who wish to run applications on the machine and represent a collection of processors. Partitions are used to control access to a machine that has been configured into a shared resource. Different partitions are created over a machine to allow the System Administrator to allocate specific resources to groups of users or run different scheduling policies simultaneously. Partitions are grouped into higher level objects called *configurations*. Any number of configurations can be created and edited at once in this view but there is only one *active configuration*. The active configuration is the resource split and scheduling policy currently in effect. Only the System Administrator has the permission to change the active configuration.

Figure 4-1 Configuration View (Node Status Color Space)



Color Spaces

The Configuration View supports the following Color Spaces. Note that the root configuration is always displayed using the Node Status Color Space.

- Configuration — identifies processors that are configured-out.
- Node type — identifies processor types.
- Node status — the boot status of the processors.
- Pmanager status — the status of the Partition Managers.
- Boot group — identifies servers and clients.
- Default — included for compatibility with other views (not useful here).

Configuration

The Configuration Color Space compliments any one of the other Color Spaces. It shows objects that have been configured-out in grey; configured-in objects are coloured using the active Color Space.

Objects that are configured-out have been made unavailable. You can configure-out objects by using the Configure function, described later (see page 52).

Node type

The Node Type Color Space identifies the types of processor in your network. This will currently be one of the following:

Viking	Texas Instruments Viking Processor
Viking + Ecache	TI Viking processor with external (2nd level) cache.
Pinnacle	ROSS Pinnacle.
605	ROSS 605.
VPU	Dual Fujitsu vector processors.
cVPU	Cache coherent dual vector processors.

Node Status

The Node Status Color Space identifies the boot status of the processors. This is the default Color Space for this view. The status will be one of:

Configured Out	Unavailable.
Powered down	Powered down.
Inactive	Indeterminate status.
Needs FSCK	Needs a filesystem check with fsck(1m).
At 'ok'	At boot ROM prompt.
Self test	Performing start-up self test.
ROM loading code	Loading kernel code.
ROM running code	Running kernel code.
CAN driver loaded	CAN device driver installed.
Unix,RLS single user	Unix single user mode.
Unix,RL1 Root FS only	Unix run level 1, only root mounted.
Unix,RL2 Multi-user, no NFS	Unix run level 2, local filesystems mount.
Unix,RL3 multi-user	Unix run level 3, multi user with NFS.

Unix,RL0 halting	Unix run level 0, halting.
Unix,RL5 reboot -a	Unix run level 5, interactive reboot.
Unix,RL6 reboot	Unix run level 6, rebooting.

Pmanager Status

The Pmanager Color Space identifies the status of the Partition Managers. The status will be one of:

PManager Down	Partition manager is no longer running.
PManager Timeout	Partition manager started but then failed (maybe a partition manager already running).
PManager Starting	Partition manager is starting-up.
PManager Running	Partition manager is running and ready.

Boot Group

The Boot Group Color Space identifies each processor as either a boot-server or a client. Clients have no attached system disk of their own and rely on a server processor to host their root filesystem and to source the operating system code.

Note that MK405 (quatro) boards are always configured as clients because they have no disk capability.

Default

The Default Color Space is common to all of Pandora's views; it uses a different colour for the processors, switches, and module controllers. In the Configuration View, which does not show switches or controllers, this Color Space has limited value.

Keyboard Short-cuts

The following keyboard short-cuts operate in the Configuration View; type the character with the mouse-pointer in the view's display area. All the short-cuts execute functions from the Function Menu (described below):

Key	Function	Description
g	Get	Get a configuration definition from a file and add the configuration to the display.
s	Set	Set partition/configuration name or processor attributes.
c	Console	Get console connections to the selected processors.
i	Info	List information about the selected processors in the Information Window.

Functions

The function menu is viewed by pressing and holding the right mouse button while the mouse is within the view's display area. The following functions are supported in this view:

- Configure — change object's configuration state.
- Create — create a new configuration or partition.
- Delete — delete processor, partition, or configuration.
- Get — fetch a configuration definition from disk.
- Put — save a configuration definition to disk.
- Set — set partition/configuration names or processor attributes.
- Change Config. — start or stop a configuration or partition.
- Command — execute a command on selected processors.
- Console — get console connections to selected processors.
- Info — list information about selected processors in the Information Window.

- Reset — reset the selected processors.
- Route Gen. — generate route tables for all the processors in the partition.
- Finder — locate objects using specified search criteria.

Configure

This function is used to change the configuration state (availability) of processors included in the ROI.

Selecting this function causes a dialogue box to appear allowing the processors in the ROI to be configured-out (or configured back in). An additional option allows you to change the link state of the links that connect to the processors. Select the required options and use Apply to make them take effect; use the Ignore button to dismiss the dialogue box without changing configuration state.

When configuring-out an object you should enable the link state change (unless you intend to change the state manually with the Set function). Setting the link state causes the link to be put into an Acking state (i.e. inoperative, but acknowledging packets before consuming them) — this is usually the preferred state because it allows broadcasts to include the range of configured-out processors without causing the broadcast to fail. When configuring-in processors you should enable the link state change to restore the links to an operative state. In this case the link is put into a Nacking state for 20 seconds before being put under Auto (H8 control); this ensures that the link is in a coherent state before being made available.

Note that confirmation of link state changes is written to Pandora's Information Window, and can be visualised with the Link State Color Space in the Network View.

Create

This function is used to create configurations and partitions. Create is context sensitive in that it will create an object whose type will depend upon the current ROI. While editing configurations no external actions are taken so it is perfectly safe to create, edit and delete configurations as much as you wish. To a configuration definition to disk you use the Put (or Change Config.) function. To make a configuration the new Active configuration you use the Change Config. function.

A configuration will be created if there is no ROI (i.e. nothing currently selected in the view). The configuration initially has no name (indicated by the '-') and is represented by a box. To assign a name to the configuration use the Set function.

A partition will be created if the ROI includes a configuration; this too will be represented by an unnamed box that can be named with the Set function. You can create as many partitions within a configuration as you require.

Processors are added into partition definitions by dragging them from the root configuration (or any other configuration definition). As processors are added to a partition's definition the size of the partition's bounding box is automatically adjusted. Note that processors must be moved into the partition by a drag-and-add operation as described on page 7 (i.e. shift and left mouse button).

The boxes drawn around the configurations and partitions can be resized and moved with the mouse. To resize a configuration you first select it and then click and hold the left mouse button, while depressing the SHIFT key, on a corner of the configuration's outer box; drag the mouse to the desired position and release the mouse button. To move either a configuration or partition you first select it and then click and hold the left mouse button, while depressing the SHIFT key, within the object's display area; drag the mouse to the desired position and release the mouse button. The positions of partitions are stored in the geometry files (see page 15) and are reused the next time Pandora displays the configuration. The positions of the configurations are not stored and will always be relative to other configurations shown in the view.

Note that when the Configuration View is first displayed only the root and Active configuration are displayed. To view other configuration definitions you must explicitly restore them from disk by using the Get function.

Delete

This function deletes the processors, partitions, or configurations in the current ROI, and (in the case of processors and partitions) resizes the bounding boxes.

If the selection was the active configuration it is only removed from the view, it is still the active configuration and is still in effect over the machine. Configurations will still remain in the filestore after a delete.

Note that you can't delete processors from the root partition.

Get

This function does not require a ROI.

This function will load a configuration previously saved by the Put function. After this function is selected a scrolling file browser is created containing names of configurations saved. A selection can be made from this list by a single left-mouse-click over the name of the configuration you wish to load. The selected configuration will be drawn into the view and becomes a selectable object capable of being edited.

Note that getting a configuration simply adds it onto the display; it does not modify the active configuration or your filesystem. To make the configuration the new Active configuration use the Change Config. function

Put

This function will save the selected configuration to the filestore. The ROI must define a single configuration.

The configuration should be named using Set prior to this operation; the configuration is saved in `/opt/MEIKOcs2/etc/`*machine-name*`/`*configuration-name*.

Set

If configurations or partitions are selected in the ROI then this function allows the setting of the object name and in the case of partitions the user groups that are allowed to use the partition as a target for applications. User groups are lists of conventional Unix groups. A user's group must match one of the partition access groups before permission to use that resource will be granted. See the `group(4)` and `permissions(4)` man pages for further information.

If processors are selected in the ROI then the following processor attributes can be set:

Auto boot
Options are: `Ignore`, `False`, or `True`. When set to true the processor will automatically reboot when the processor is Pulse Reset (see the Reset function); if set to false the processor will remain at the Open Boot prompt awaiting a manual reboot.

Boot device
Options are: `Ignore`, `Elan`, `Disk`, or `Net`. Specify the boot device for this processor. Elan means boot from a boot server via the Elan network, disk means boot from the local system disk, and net means boot from Ethernet.

Boot Args
Options are: `Ignore`, `noargs`, `-s`, `kadb`, `kadb -s`, `kadb -r`. Specifies the default arguments used when rebooting.

Console Device
Options are: `Ignore`, `keyboard/screen`, `CAN`, `ttyA`, `ttyB`. Used to specify the connection used by this processor's console. keyboard/screen means the graphics terminal and keyboard, CAN means the CAN bus, and tty-A/B means the serial ports. You should select CAN if you intend to allow Pandora to create a remote console connection to this processor.

Enable Console stealing
Options are: `no` or `yes`. Only one console connection may exist to any one processor. With console stealing disabled (set to no) requests for a console will be refused if a console connection exists elsewhere. If console stealing is enabled the existing connections will be terminated immediately to allow the new connection to be created.

Auto Configure Boot params
Options are: `no` or `yes`. If enabled (set to yes) allows the processor's Elan Id and other per-processor values to be allocated automatically. See Network Options, below.

Elan Protocol Options — various
A number of Elan options can be specified; these are intended for Meiko engineers only and should not be adjusted. Default values are read from the `defaults` file.

Network Options — various
A number of network options can be specified, including the processor's Elan Id, it's position in the switch network, and the id of it's boot server. Typically these are assigned automatically by specifying the Auto Configure Boot Params option, as described above. Default values are read from the `defaults` file.

Change Config.

This function allows partitions in the Active configuration to be started or stopped, and allows a new configuration to be made active. The ROI must include either partitions or configurations.

This function uses `rcontrol(1)` to change the configuration. Output from `rcontrol(1)` is written to text window — type the return key at the prompt to dismiss the window.

The following options appear in a dialogue box when this function is selected:

Output
Options are Normal or Interactive. Selecting Interactive instructs Pandora to request confirmation before changing the configuration.

Action
Options are Start or Stop. Specifies whether the selected partition/configuration is to be started or stopped.

Kill Jobs
Options are No or Yes. Specifies when the configuration change takes place. By setting this option to yes, all current jobs are killed and the configuration change happens immediately; otherwise existing jobs are allowed to complete before the configuration is changed.

When using this function to change to a newly defined configuration, the new configuration definition will automatically be saved to disk if it has not already been saved (with the Put function).

Command

This function allows a command or user application to be executed on all processors in the selected partition. The ROI must identify a single partition in the Active configuration.

This function generates a dialogue box with a partially completed `prun(1)` command line (`prun -p` *partition_name* `-v`). Append the name of your program to this line and use the apply button to execute it. Program output is written to a dedicated window.

See also the manual page for `prun(1)`.

Console

This function will initiate a remote console connection to each of the processors in the ROI.

Each console connection is started in its own window. Typed characters are normally transmitted directly to the remote processor. A tilde (~) character appearing as the first character of a line is an escape signal which directs some special action. Recognised escape sequences are:

`~^D`	Drop the connection and exit.
`~.`	As above.
`~#`	Send a BREAK to the remote processor.
`~?`	Print a summary of the tilde escapes.

See also the man page for `cancon(1m)`.

Info

This function writes to the Information Window detailed information about each of the processors in the ROI. The information includes:

- Hostname
- NVRAM settings (as specified by the Set function).
- Configuration and boot status.
- Hardware configuration (board type and position in the network).

Reset

Resets all of the processors identified by the ROI. A dialogue box prompts for confirmation. A second dialogue box appears offering the following reset options:

Send Break
Halt the specified processors immediately. For use only with diskless client processors.

Hold in Reset
Hold the processors in a reset state.

Pulse Reset
Uses the processor's Auto Boot attribute (see the Set function) to determine whether the processor is to be booted following the reset.

Halt Procs
Equivalent to an `init 0` command. A graceful shutdown of the specified processors.

Boot Procs
Used to boot the specified processors.

Route Gen.

This function generates a number of files listing the network routes between processors in the partition. The ROI must identify a partition.

A route file is generated for each processor, each file identifying the network routes to the other processors in the same partition. The routes may be generated using either a random or a scatter algorithm and may be on either (or both) of the two switch layers (both options are selectable from the dialogue box).

The route files are stored in a directory called `routes` alongside the definition of the partition (i.e. in `/opt/MEIKOcs2/etc/`*machine-name*`/`*config-name*`/`*partition-name*). The filename for each route file is the same as the Elan Id of the processor that it was generated for. Note that the format of the route files is not documented and is subject to change.

Route files are loaded into each processor's route tables with `rmsroute`(1m).

Finder

Used to locate objects according to the specified search criteria. The objects that match the search criteria are shown highlighted in the Configuration View.

This function generates a dialogue box allowing you to search for Boards or Processors. Selecting one or more of these options produces additional prompts that allow you to specify the search criteria. You may supply information to as may of these prompts as are necessary. The criteria are applied using a logical AND within each object type, and a logical OR between object types. Where a numeric search criteria may be entered it is often possible to enter either a numeric range (e.g. 1-4), or a comma separate list of ranges (e.g. 1-4,7,9-10).

Criteria that may be applied to **Boards** are:

- Board type — matches board type.
- ROM revision — matches H8 ROM revision.
- Number or procs — matches boards with the specified number of processors (a number range such as 1-4 matches boards with 1, 2, 3, or 4 processors).

Criteria that may be applied to **Processors** are:

- Memory MB — matches boards fitted with specified memory (a number range such as 32-64 matches boards with between 32 and 64MBytes).
- Elan Id — a network Id (or range of Ids).
- ROM revision — matches boot ROM revision.
- Host name — locates the processor with the specified hostname.
- Boot device or host — used to identify a server processors (the other search criteria identify one or more client processors).

Interaction with Other Views

You can drag the following objects from the Configuration View to other views:

- You can drag processors into the Machine View to visualise their physical position in the machine. This is useful if the Configuration View identifies a processors that needs to be removed from the machine.
- You can drag processors into the Network View to determine their placement in the network.

- You can drag partitions into the Performance View to visualise processor performance.

You can drag the following objects into the Configuration View:

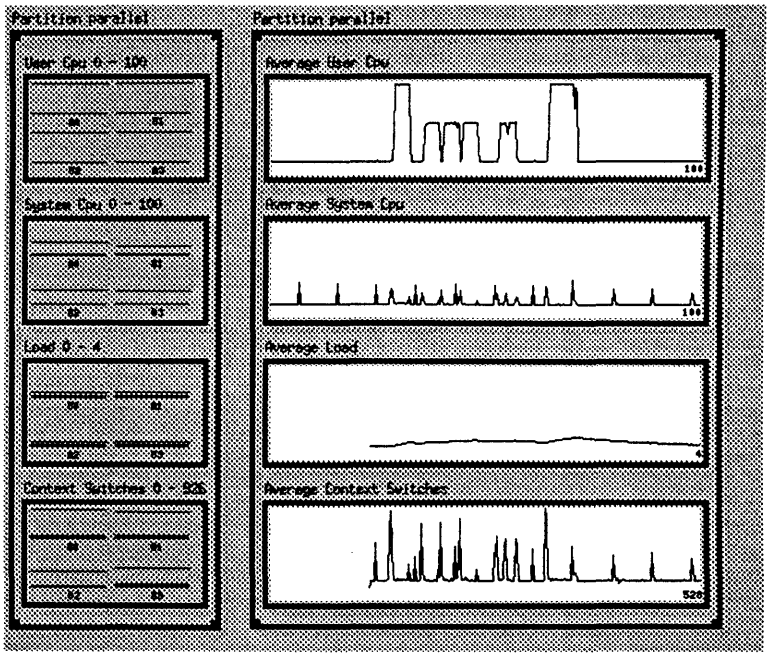
- You can drag processors from the Machine View into the Configuration View to identify the partitions that use the selected processors.
- You can drag processors from the Network View.

Performance View 5

The Performance View is used to visualise the performance of the processors in a partition. Statistics can be gathered and displayed as bar graphs for each node in the partition, or a summarising graph can be displayed showing the average, minimum, or maximum over all processors in the partition.

The Performance View shown in Figure 5-1 shows both bar graph and 'wiggle trace' displays for the processors in the Parallel partition. The bar graph displays on the left show the instantaneous User CPU, System CPU, Load, and Context Switched for each of the four processors in the partition. The trace displays on the right show Average User CPU, Average System CPU, Average Load, and Average Context Switches calculated over all processors in the partition.

Figure 5-1 Performance View



Color Spaces

The Performance View has no Color Spaces.

Keyboard Short-cuts

The following keyboard short-cut operates in the Performance View; type the character with the mouse-pointer in the view's display area. The short-cut executes a function from the Function Menu (described below).

Key	Function	Description
s	Set	Set the statistics to be gathered.

Function Menu

The function menu is viewed by pressing and holding the left mouse right mouse button while the mouse is within the view's display area. The following functions are supported in the Performance View:

- Delete — remove a graph from the display.
- Set — set statistics to be gathered.
- Reset — reset high water marks on bar graph displays.

Delete

This function is used to remove one or more graphs from the display. The ROI should include one or more graphs. Include a partition in the ROI to delete the displays for the whole partition.

Set

This function is used to create additional statistics graphs for a partition, or to determine the statistics that are gathered by default when statistics gathering is started for a new partition.

If the ROI is empty the Set function defines the default statistics that are gathered when a partition is selected. The following options are displayed within a dialogue box. Select the Individual option to create a bar graph display for each processor in the partition. Use the Average, Minimum or Maximum options to create a trace display showing the average/minimum/maximum calculated over all processors in the partition. You may select more that one statistic.

User CPU
This is the time spent executing user code.

System CPU
This is the time spent executing kernel code.

Total CPU
Time executing code (sum of User and System CPU statistics).

Page Faults
Page fault count.

Load
The number of jobs in the run queue.

Free Real Memory
Free memory; this is the maximum amount of physical memory.

Free Memory
Free memory including swap space.

Context Switches
The number of context switches.

Disk R/s
Disk reads per second. When bar individual bar graphs are displayed the values shown in each graph are taken over all disks connected to each node.

Disk W/s
Disk writes per second. When bar individual bar graphs are displayed the values shown in each graph are taken over all disks connected to each node.

Disk RW/s
Disk read-write count per second. When bar individual bar graphs are displayed the values shown in each graph are taken over all disks connected to each node.

Disk Read Kb/s
Disk read bandwidth (KBytes per second). When bar individual bar graphs are displayed the values shown in each graph are taken over all disks connected to each node.

Disk Write Kb/s
Disk write bandwidth (KBytes per second). When bar individual bar graphs are displayed the values shown in each graph are taken over all disks connected to each node.

Disk RW/s
Disk I/O bandwidth (KBytes per second). When bar individual bar graphs are displayed the values shown in each graph are taken over all disks connected to each node.

Layout
Options are either columns or rows. Used when selecting individual graphs (i.e. one bar graph per node) to determine the layout of the graphs on the screen. Used with the Dimensions option.

Dimensions
When individual bar graph displays are created for each node in the partition the graphs are arranged in a grid. Use this option and the Layout option to specify with the width (columns) or height (rows) of the grid. These options are useful when displaying individual bar graphs for all the processors in a large partition, where large numbers of graphs will be displayed.

If the ROI includes a single partition you can add graphs to the display, or remove graphs from it. A dialogue box appears which includes all the statistics listed above. Select un-ticked boxes to add the statistic to the display; by selecting a ticked box (thus clearing it) you will remove the display for that statistic.

Reset

This function is used to reset the high water marks on the bar graph displays. Use the reset function to reset this line to zero.

Interaction with Other Views

You can drag partitions from the Configuration View into the Performance View.

5

Common Operations 6

This chapter describes some common administration and testing functions; new users of Pandora can use these procedures to become familiar with Pandora, whereas existing users can use them as a quick reference.

Refer to the previous reference chapters for more information about the functions used in these procedures.

System Configuration

System configuration functions are undertaken in the Configuration View.

Defining a New Configuration

This section describes in outline how to define a new configuration and how to make it Active. You will typically use these steps when configuring your machine for the first time, or when defining a new configuration to meet the changing needs of your users.

1. **Clear the current ROI.**
   A left click somewhere in the view (but not over a network object) will clear an existing ROI and ensure that no objects are selected.

2. **Create a new configuration.**
   Select the Create function; an empty box should appear representing the new configuration.

3. **Assign a configuration name.**
   The configuration is created without a name (just '-'). Assign a name by selecting the new configuration in the ROI and by using the Set function.

   The choice of name is yours. The configuration name will map to a filename on your disk so your choice should conform to file naming conventions (alphanumeric characters only).

4. **Create a new partition.**
   Select the configuration in the ROI and use the Create function; and empty box should appear within the configuration.

5. **Assign a partition name.**
   The partition is created without a name (just '-'). Assign a name by selecting the new partition in the ROI and by using the Set function.

   The choice of name is yours. The partition name will map to a filename on your disk so your choice should conform to file naming conventions (alphanumeric characters only).

   If you have used the `groups`(4) and `permissions`(4) files to define user groups you can restrict access to the partition to one or more of these groups. Enter the group names at the prompt, or leave the entry empty to allow unrestricted access.

6. **Add the processors to the partition.**
   Drag the required processors from the root partition (or any other configuration shown in the view).

   Select the first processor with a single left click within the root partition. You can add additional processors to the ROI by selecting them with a single middle click. A drag-and-add operation will copy the selected processors to the new partition; hold down the shift key, select one of the processors in the ROI by pressing and holding the left mouse button, and drag the processors into the new partition. Release the mouse button to drop the processors into the new partition.

   You can add as many processors as you wish, using as many drag-and-add operations as you wish. You can remove a processor from your new partition by selecting it and using the Delete function.

7. **Create and name the remaining partitions.**
   Repeat steps 4, 5, and 6 to define the remaining partitions in the configuration.

8. **Save the configuration to disk.**
   Select the configuration in the ROI and use the Put function.

   A new directory hierarchy is created on your disk in `/opt/MEIKOcs2/etc/`*machine-name*.

9. **Make the configuration Active.**
   Select the configuration in the ROI and use the Change Config. function.

   A dialogue box will appear: select Normal and Start. The Kill Jobs option has no effect unless an active configuration already exists, in which case you should disable the Kill Jobs option if you want existing jobs in the configuration to terminate normally before the configuration is changed.

Changing a Configuration

This section describes how to change the active configuration.

- How to stop a single partition.
- How to start a partition.

Stopping a Partition

This section describes how to stop a partition manager; this will prevent users from running parallel applications on the partition. The status of each node remains unchanged by this operation.

You will typically stop a partition before reconfiguring your system, or before shutting down one of more of the processors in the partition.

1. **Select the partition in a ROI.**
   Use a single left mouse click within the partition.

2. **Change the configuration.**
   Select the Change Config. function.

A dialogue box appears. Select Normal and Stop. You should opt to Kill Jobs if you wish the partition to stop immediately; otherwise existing jobs running on the partition will be allowed to stop normally.

Starting a Partition

This section describes how to start a partition manager. You will typically start a configuration after performing routine system administration on its nodes, or after the partition manager fails unexpectedly.

1. **Select the partition in a ROI.**
   Use a single left mouse click within the partition.
2. **Change the configuration.**
   Select the Change Config. function.

   A dialogue box appears. Select Normal and Start. The setting of the Kill Jobs option is unimportant if the partition manager had already stopped.

If you are using this procedure to restart a failed partition manager, the Pmanager Status Color Space will give more information about the new partition manager. The Node Status Color Space will identify faulty processors which may be responsible for the failure of a partition manager.

Network Tests

You can use the Network View's Link State and Switch Error Color Spaces to query the status of network components, as shown below. Note that in both cases the test result is relative to the previous test (or system start-up if no previous query).

To test the network components thoroughly you will typically query the initial state of the network, exercise the network, and then query the network state for a second time to identify recent errors. You can use any program you like to exercise the network; `rtest(1)` or the boundary scan enabled by the Set function are suitable. You may also use the Set function to enable a count of timeout errors.

Link Tests

1. **Select the Link State Color Space.**
2. **Select the Links in a ROI.**
   Ensure that the components at both ends of the links are selected.
3. **Update the Link State display for the selected links.**
   Use the Get function (either by selecting it from the function menu or by typing 'g' into the view).

Now exercise the network and repeat the above steps to reveal new errors.

Note that in Step 2. you can either select the link ends manually with the mouse, or you can use the Finder's Route option to describe the route in terms of a processor and routes through the Elan and Elite switches.

Switch Errors

1. **Select the Switch Error color Space.**
2. **Select the switches in a ROI.**
   You can select the processors if you wish to query the Elan communications processors.
3. **Update the Switch Error display for the selected components.**
   Use the Get function by typing 'g' into the view (do not select Get from the Function Menu) to update the display and write a summary of the switch errors to the Information Window.

Now exercise the network and repeat the above steps to reveal new errors.

Locating Faulty Components

Faulty network switches or communication processors can be located by dragging the component from the Network View and into the Machine View. This will identify the board and module that holds the faulty component. Use a drag-and-find operation to move the component between the views (shift and middle mouse button).

Before removing a faulty board from your system be sure to configure-out the processors and switches that are fitted to it. This will prevent user applications and system software accessing the components.

Warning – Only trained engineers may remove hardware components from your system. If in doubt contact Meiko for advice.

Processor Diagnostics

Errors on the processor boards can often be overcome by either rebooting the processor or by using the processor's ROM functions to perform simple system tests.

Getting a Console Connection

You can get a connection to a processor's console by using the Console function in the Configuration View:

1. **Select the processor in a ROI.**
   You can select several processors if you wish.
2. **Select the Console function.**
   Select the function from the Function Menu or by typing 'c' into the view. Each console appears in a separate window.

Note that only one console connection may be created for each processor. You must enable the processor's console stealing attribute if you want to take the connection from another user — see the Set function.

Having obtained a console connection all the usual diagnostic and system administration functions are available — see the Solaris documentation set. You can use the console connection to shutdown/reboot a processor, or to perform the diagnostic tests in the processors ROM.

Rebooting

Processors can be reset from within either the Network or Configuration Views:

1. **Select the processor in a ROI**
   You can select several processors if you wish.
2. **Select the reset function.**
   Select the reset function from the Function Menu. A dialogue box asks for confirmation.
3. **Specify the type of reset operation.**
   A list of reset options is displayed. Use the Send Break option on diskless clients to cause an immediate shutdown. Use Pulse Reset to perform the equivalent of a power-cycle. Use the Halt Procs option to shutdown processors gracefully (equivalent to init 0). Use Boot Procs to boot processors that have already been reset.

A number of processor attributes specified by the Set function affect processor booting. The Auto Boot attribute specifies the behaviour of the processor following a Pulse Reset operation. The Boot Args attribute specifies the arguments that are passed to the Boot Procs operation.

Performance Visualisation

You use the Performance View to visualise the utilisation of partitions in on CS-2.

Typically you will define the default statistics that will be gathered before you copy a partition into the Performance View. You then copy partitions into the Performance View by dragging them from the Configuration View. You can change the displays of individual partitions where the default is inappropriate.

Creating a New Display

you use the Set function to define the default display that will be produced for partitions that are dragged into the Performance View.

1. **Select the Performance and Configuration Views from the Root Panel.**
   The Configuration View will identify all the partitions in your system. The Performance View will initially be empty.

2. **Identify the statistics you want to view.**
   With the mouse in the Performance View select the Set function (either from the function menu or by typing 's'). Identify the statistics that you want to display.

   For example, select Average from the User CPU option to get a histogram display showing the Average CPU utilisation for all processors in a partition. Select Individual from the Disk RW/s option to get individual bar graph displays showing the disk bandwidth on each node.

   Use the Layout and Dimensions options to define the layout of your bar graphs displays (only necessary if you intend to visualise a large partition).

3. **Select a partition.**
   Select a partition in the Configuration View. Use a drag-and-add operation (shift left-mouse) to copy the partition into the Performance View. The statistics that you specified in Step 2. will be displayed. Repeat for any other partitions that you want to visualise.

Changing an Existing Display

You use the Set function to change an existing display to include additional statistics or to remove them (note that you can also use the Delete function to remove statistics from the display).

1. **Identify the Partition.**
   Include in the ROI the partitions' whose displays you want to change.

2. **Specify the new statistics.**
   Use the Set function (either by typing 's' or by using the mouse) to obtain the dialogue box. Identify the statistics that you want to add to the display by selecting them with the mouse. Remove statistics from the display by un-selecting them.

Computing Surface

Elan Widget Library

S1002–10M104.04

meiko

Meiko's address in the US is:

Meiko
130 Baker Avenue
Concord MA01742

508 371 0088
Fax: 508 371 7516

Meiko's address in the UK is:

Meiko Limited
650 Aztec West
Bristol
BS12 4SD

Tel: 01454 616171
Fax: 01454 618188

Issue Status:

Draft	
Preliminary	
Release	x
Obsolete	

Circulation Control: *External*

Contents

Elan Widget Library

This document includes manual pages for the Elan Widget Library. It also includes a detailed description of the error messages that are produced by this library.

Compilation

Applications that use the functions in this library must be linked with `libew.a` and `libelan.a`, both in the directory `/opt/MEIKOcs2/lib`. In addition Elan Widget programs reference header files from the directory `/opt/MEIKOcs2/include`. Both library and include file directories must be specified in the compiler command line, as shown below:

```
user@cs2-0: cc -c -I/opt/MEIKOcs2/include myprog.c
user@cs2-0: cc -o myprog myprog.o -L/opt/MEIKOcs2/lib -lew -lelan
```

1

libew — **elan widget library**

Synopsis

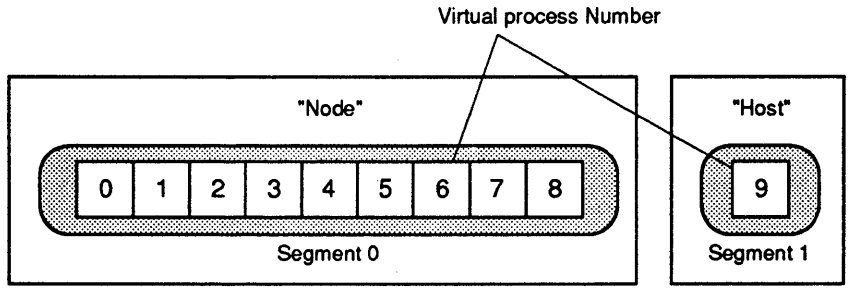
```
#include <ew/ew.h>
```

`libew` provides a parallel programming environment for higher level library implementors and applications programmers who wish to optimise performance. The set of parallel programming constructs it provides does not hide, but augments the basic capabilities of the elan/elite communications network. This frees the user from low level hardware considerations, without sacrificing performance to generality or ease of use.

Process Model

A parallel application is a collection of one or more segments. Each segment consists of a set of processes replicated over a set of network-contiguous processors, one process per processor. All processes in a segment execute the same program.

Figure 1-1 Two Segment Parallel Application.



All processes in all segments of a parallel application start up together. This assigns every process a unique virtual process number in a contiguous range starting from 0, and makes every process's address space accessible to its peers through the network. For more information see `ew_init()`.

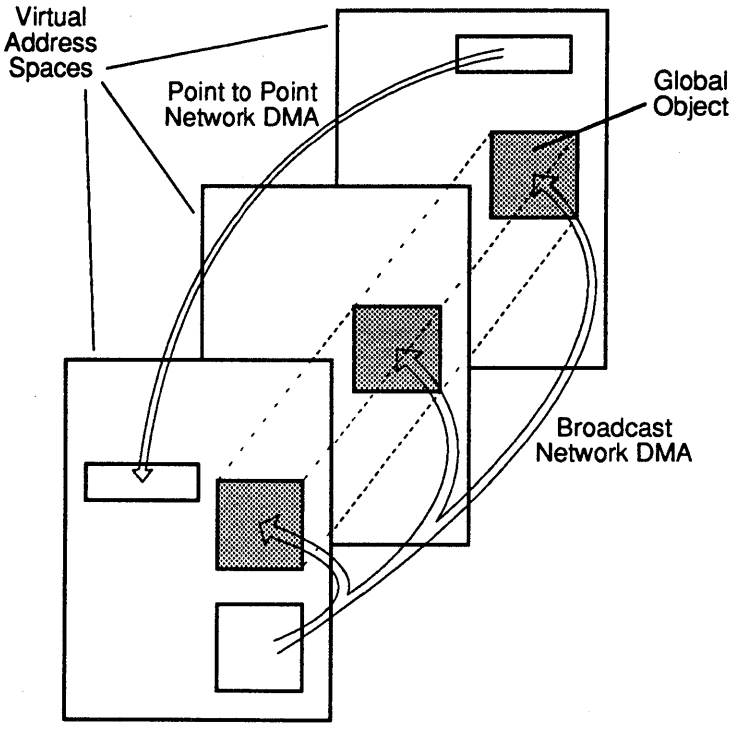
A set of processes with contiguous virtual process numbers and contained within the same segment may be addressed by a single broadcast virtual process number. See `ew_createBcastVp()`. Broadcast virtual process numbers allow an application to exploit hardware broadcast.

System calls are handled locally by default. For example, every process of a p allel application can access the file system independently with standard Unix brary and system calls. A subset of system calls can be redirected to a nominat server to concentrate system calls relating to the standard input, output and er on a single process. See `ew_rsysServerInit()`.

Global Memory

The address spaces of the processes of a parallel application constitute a distr uted global memory. Non-local memory, addressed by a combination of virtu process number and virtual memory location, can be accessed **explicitly** by n work DMA operations. `libew` provides a non-blocking interface to these op ations. See `ew_storeStart()` and `ew_fetchStart()`. Network DM do not require the cooperation of the remote process and they transfer data wi the lowest latency.

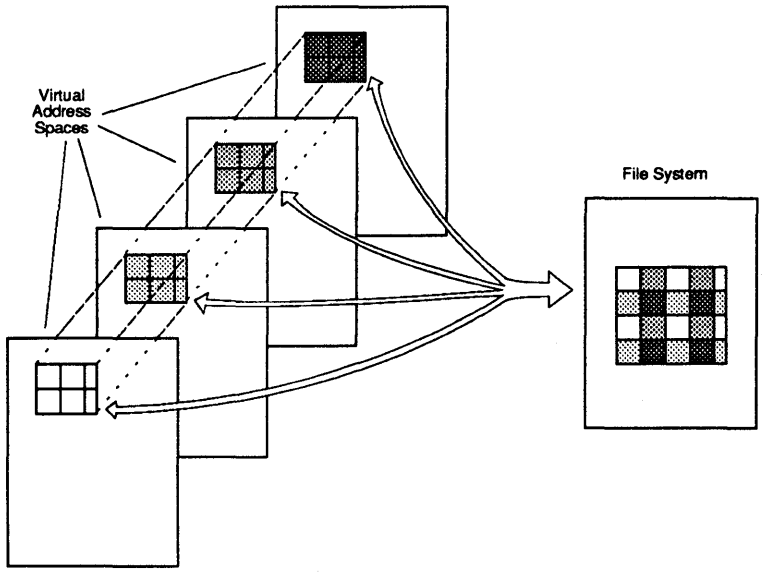
Figure 1-2 Non-local Memory Access.



Global objects are data structures which are distributed over a set of processes, but located at the same virtual address within each process. Each component of a global object is called a slice. If the processes owning a global object are contiguous within a single slice, a single broadcast network DMA may be used to replicate source data in one process to all slices of the global object.

Many `libew` constructs are themselves global objects. They can be created dynamically by their owning processes through synchronised use of the `libew` global heap management procedures. See `ew_allocate()`.

Figure 1-3 Global Object File I/O.

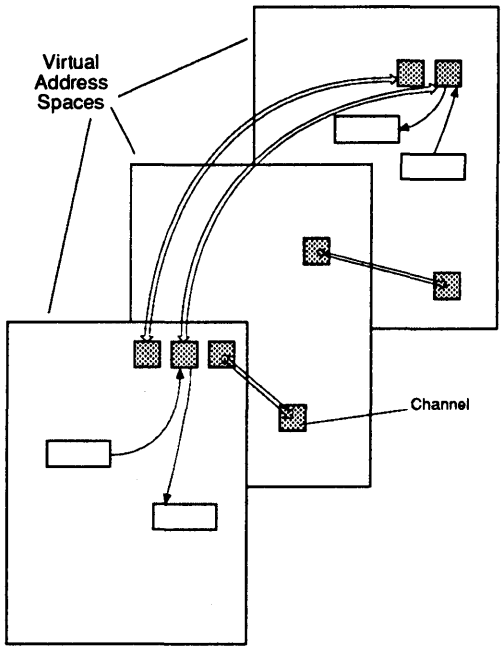


A global object may be read from or written to the file system as a single entity. See EW_PFD. Regular distributions of *n*-dimensional arrays as global objects are supported. They are redistributed as they are read and written to convert from the specific distribution required by the application, to a canonical representation in the file system. This allows sequential applications as well as parallel applications with different process decompositions to share the same data.

Message Passing

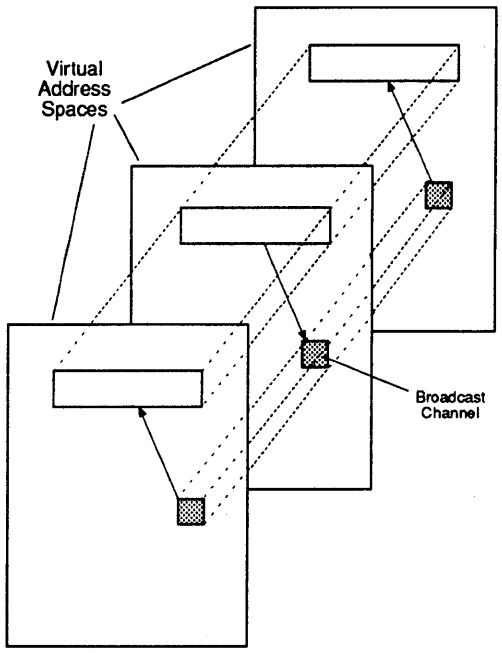
Unlike network DMAs, message passing requires the cooperation of both sen
ing and receiving processes. `libew` supports several types of message passir
as appropriate to different programming models and functionality requiremen

Figure 1-4 Channel Communication.



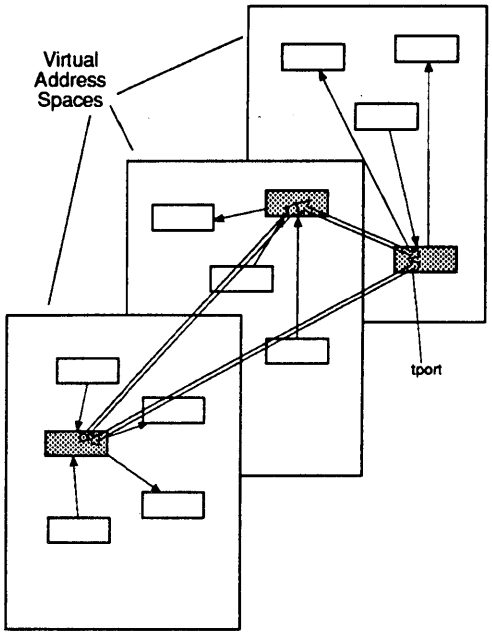
Channels provide the simplest and lowest latency message passing. See `EW_CHAN`. A channel connects a pair of processes. The connection must be estab·lished by both processes before it can be used. Communication is unbuffered. Messages are transferred directly from the sending buffer to the receiving buff therefore when a transmit completes, it guarantees that a receive has been poste Message passing is non-blocking and full duplex. Both processes at the ends a channel may have up to one transmit and one receive outstanding at any tim Multiple channels may be connected between a pair of processes to allow mo non-blocking operations and the passing of non-contiguous messages.

Figure 1-5 Broadcast Channel Communication.



A broadcast channel is a global object. It is used to barrier synchronise and replicate the slices of other global objects. See `EW_BCHAN`. It has a non-blocking interface which allows up to one outstanding broadcast at a time. Multiple broadcast channels may be used by a group of processes to allow more outstanding broadcasts.

Figure 1-6 Tagged Communication.

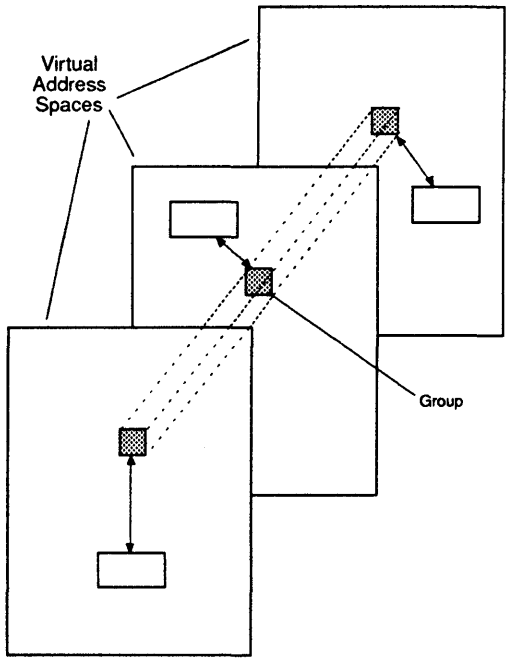


Tagged messages are passed between tagged message passing ports, called tports. See EW_TPORT. They support both buffered and unbuffered message passing, with a non-blocking interface which allows arbitrary numbers of out standing transmits and receives. Messages may be received selectively, both c the sender and on the tag. Given equivalent selection criteria, messages passe from the same source to the same destination remain ordered.

Groups

A group is a global object. Groups are used to define arbitrary subsets of the processes of a parallel application including non-contiguous and irregular sets. Groups number their members in a contiguous range starting from 0. A user-supplied group membership function maps group member number to virtual process number. This implicitly determines the set of processes in the group.

Figure 1-7 Reduction with a group.



Groups support barriers, broadcasts, reduction and global exchange. See `EW_GROUP` and `EW_GEX`. Groups exploit hardware broadcast where appropriate. There is no requirement that they operate only on global objects.

Tracing

`libew` provides an interface for the generation of `ParaGraph` format trace files. See `ew_ptrace()`. Each process being traced may independently enable, disable or flush traces to their own output file. The trace files are merged with `sort -m` before they are displayed.

Exception Handling

`libew` treats errors which undermine its operating assumptions (e.g. that an coming message should be received into a buffer of greater or equal size) as fa It attempts no error recovery, however it provides an exception handling inte face which allows the user to intervene prior to process termination. See `ew_exception()`.

Base Environment

`libew` provides a collection of parts for building higher level programming models. The `libew` base environment contains a minimal set of facilities bu from these parts which are required by the majority of these models. It simplif common initialisation procedures and standardises resource usage so that dif ent models can work in the same application without interfering with each oth This allows an application written in one programming model to exploit paral libraries written with a different model. See `ew_base`.

See Also

`ew_init()`, `ew_base`, `ew_allocate()`, `EW_GROUP`, `EW_DMAPOOI` `EW_CHAN`, `EW_BCHAN`, `EW_TPORT`, `ew_rsysServerInit()`, `ew_ptrace()`, `ew_exception()`.

ew_init — **ew_ctx, ew_state, ew_init, ew_attach — elan widget library initialisation**

Synopsis

```
#include <ew/ew.h>
void* ew_ctx;
EW_STATE ew_state;
void ew_init (void);
void ew_attach (void);
```

Description

`ew_init()` performs preliminary initialisation for `libew` procedures which do not access the network. `ew_attach()` completes `libew` initialisation and connects the calling process's address space to the network. This two-stage process creates the opportunity to perform user initialisations local to a process, before its address space becomes accessible to its peers.

`ew_ctx` is initialised by `ew_init()`. It is the handle on the elan context that the process will use in all its network operations. It should be passed when applications wish to call `libelan` procedures directly.

`ew_init()` generates an exception with code `EW_EINIT` if the application has been linked with an incompatible version of `libelan`, if it can't open `/dev/zero` or if it can't initialise the elan.

`ew_state` packages the remaining `libew` invariants into a single structure. Its components are listed below. Initially, all its components are zero. `ew_init()` initialises `ew_state.version`, and sets `ew_state.initialised`. The remaining components, are set by `ew_attach()` and remain constant thereafter.

Component	Description
`int version`	`libew` version string.
`int initialised`	`TRUE` after `ew_init()` has been called.
`int attached`	`TRUE` after `ew_attach()` has been called.
`ELAN_CAPABILITY *segCaps`	Capabilities of all segment of the application.
`int seg`	Segment number of this process.
`int nseg`	The total number of segments in the application.
`int segbasevp`	Virtual process number of the first process in this segment.

Component	Description
`int segnvp`	Total number of processes in this segment.
`u_int vp`	Virtual process number of this process.
`int nvp`	Total number of processes in the application.

`ew_attach()` initialises the array of segment capabilities from the environ ment and assigns the virtual process numbers from a contiguous range startin with 0 for process 0 of segment 0. Each segment of the parallel application, i cluding the process's own segment is represented by an environment string which encodes the elan capability for the segment.

Environment Variable	Description
`LIBEW_ECAP`	Capability of this process's segment
`LIBEW_ECAP0`	Capability of first segment
`LIBEW_ECAP1`	Capability of second segment
`...`	...
`LIBEW_ECAP`*n-1*	Capability of last segment

It is an error for `LIBEW_ECAP` not to match one of the other capabilities. Ho ever single segment applications may specify only `LIBEW_ECAP`. This is equi alent to specifying `LIBEW_ECAP` and a matching `LIBEW_ECAP0`.

`ew_attach()` causes an exception with code `EW_EINIT` if it fails to extra a consistent set of capabilities from the environment, or if it fails to attach to t network.

See Also

`ew_base`.

ew_base — **ew_base, ew_baseInit — libelan base programming environment**

Synopsis

```
#include <ew/ew.h>
EW_BASE ew_base;
void ew_baseInit (void);
```

Description

The `libew` base environment provides a set of facilities which are commonly required by the implementations of higher-level programming models. These include defaults for performance related constants to be passed to other `libew` constructs, a global allocator, useful groups and a remote system call server.

Note that none of the parameters in the base environment are referenced elsewhere in `libew`. They are provided only so that other libraries, obeying a common set of conventions on how they use `libew`, can be linked into the same application.

`ew_base` is a single structure containing the components listed below. It is initialised by `ew_baseInit()`. Initially all components are zero.

Component	Description
`int init`	`TRUE` after `ew_baseInit()` has been called.
`int waitType`	Default parameter to `elan_waitevent()`. This value should be passed on initialisation of appropriate `libew` constructs.
`int dmaType`	Default network DMA type parameter. This value should be passed on initialisation of appropriate `libew` constructs.
`caddr_t alloc_base`	Base of global dynamic memory.
`int alloc_size`	Size of global dynamic memory.
`EW_ALLOC *alloc`	Global dynamic memory allocator.
`int group_bufsize`	Default group packet size.This value should be passed to `ew_groupInit()`.
`int group_branch`	Default group spanning tree branching ratio. This value is passed to `ew_groupInit()`.

Component	Description
`int group_hwbcast`	`TRUE` if groups composed of processes with contiguous virtual process numbers should use hardware broadcast by default. This value deter mines whether a broadcast virtual process number is passed to `ew_groupInit()`.
`EW_GROUP *allGroup`	The group of all processes. This group is creat ed by `ew_baseInit()`.
`EW_GROUP *segGroup`	The group of processes in this process's seg- ment. This group is created by `ew_ baseInit()`. If the application only has a sin- gle segment it is identical to `allGroup`.
`int tport_nattn`	Default number of attention slots. This value should be passed to `ew_tportInit()`.
`int tport_smallmsg`	Default small message size. This value should be passed to `ew_tportInit()`.
`int rsys_enable`	`TRUE` if remote system call serving is enabled and `ew_baseInit()` has initialised this proc- ess as a remote system call client.
`u_int rsys_server`	The virtual process number of the remote sys- tem call server (if enabled).
`int rsys_bufsize`	The maximum packet size for remote system calls (if enabled).

`ew_baseInit()` sets the constants in `ew_base` to values specified by the e vironment variables listed below. All the environment variables, with the exce tion of those indicated, apply to the `ew_base` component of the same name. no corresponding environment variable is set, a default is chosen. Integer con stants may be specified in decimal or hex, by prepending "0x".

`ew_baseInit()` performs the two stage `ew_init()`, `ew_attach()` ini tialisation, creates the global heap and sets up the "all" and segment groups. I calls `ew_pfInit()`, passing it the default `waitType` and `dmaType`.

If remote system calls are enabled, it spawns a system call server on the nominated process, starts system call redirection on all other processes and sets line buffering on `stdout` and `stderr`.

`ew_baseInit()` causes an exception with code `EW_EINIT` if it has not been called before, but `libew` is already initialised (i.e. `ew_attach()` has been called), or if it fails to create the global heap or allocate its groups.

Environment Variable	Values	Default
LIBEW_WAITTYPE	POLL WAIT integer	ELAN_POLL_EVENT
LIBEW_DMATYPE	NORMAL SECURE integer	TR_TYPE_BYTE (Sets the dma_opCode component of dmaType).
LIBEW_DMACOUNT	integer	1 (Sets the dma_failCount component of dmaType).
LIBEW_GROUP_BUFSIZE	integer	8192
LIBEW_GROUP_BRANCH	integer	2
LIBEW_GROUP_HWBCAST	0,NO 1,YES	TRUE
LIBEW_TPORT_NATTN	integer	4
LIBEW_TPORT_SMALLMSG	integer	4096
LIBEW_ALLOC_BASE	caddr_t	0xe0000000
LIBEW_ALLOC_SIZE	integer	0x10000000
LIBEW_RSYS_ENABLE	0,NO 1,YES	TRUE
LIBEW_RSYS_BUFSIZE	integer	8192
LIBEW_RSYS_SERVER	integer	0 in a single segment application. ew_state.nvp-1 in a multi-segment program.

ew_version

ew_version, ew_checkVersion — libew version checking

Synopsis

```
#include <ew/ew.h>
#define EW_VERSION
char *ew_version (void);
int ew_checkVersion (char *version);
```

Description

`EW_VERSION` is a macro which gives the version string of the particular instance of `libew` against which an application was compiled.

`ew_version()` returns the version string of the particular instance of `libe` with which an application was linked.

`ew_checkVersion()` checks that the version of `libew` against which an a plication was compiled is compatible with the version with which it was linke It returns `TRUE` if `version` is compatible.

Example

```
if (!ew_checkVersion (EW_VERSION))
{
   fprintf (stderr, "libew version error\n");
   fprintf (stderr, "  Compiled with '%s'\n", EW_VERSION);
   fprintf (stderr, "  Linked with   '%s'\n", ew_version ());
   exit (1);
}
```

ew_exception

ew_exception, ew_setExceptionHandler, ew_exceptionStr — exception handling

Synopsis

```
#include <ew/ew.h>
typedef void (*EW_EXH)(int code, char *msg);
void ew_exception (int code, char *format, ...);
EW_EXH ew_setExceptionHandler (EW_EXH handler);
char *ew_exceptionStr (int code);
```

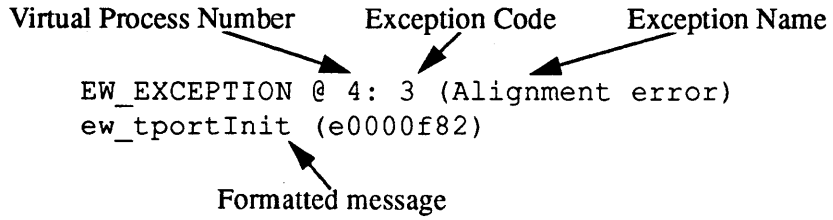
Description

`ew_exception()` initiates a `libew` exception. `code` is a numeric exception code. Any value may be passed, however values below `1000000` are reserved to Meiko. `format` is a `printf()` style format string which is used to produce a formatted message.

Warning – The maximum size of the message after formatting is `256` bytes including the terminating `NULL` character.

`ew_exception()` calls the currently installed exception handler with the exception code and the formatted message. On return from the handler, if the exception code is `EW_DEBUG`, or if the environment variable `LIBEW_TRACE` is set according to the table below, it dumps internal debugging traces. All exception codes other than `EW_DEBUG` result in program termination with an optional core dump, depending on the environment variable `LIBEW_CORE`.

The default exception handler prints a message on `stderr` listing the process's virtual process number, the exception code number, its symbolic representation and the formatted message.



`ew_setExceptionHandler()` allows users to install their own exception handlers. It sets the currently installed exception handler to `handler`, and returns the previously installed handler. Users should save the old handler, so that

it may be called if the user's handler is passed an exception code that it does r understand. This allows different libraries to share a common exception mec nism.

`ew_exceptionStr()` returns a pointer to a static string which is a textual representation of the exception code `code` passed to it. It returns "`Unknowi exception`" if it is passed an exception code which did not originate from `libew`. Current `libew` exception codes are listed below.

Name	Value	String representation
`EW_DEBUG`	1	Debug
`EW_EINTERNAL`	2	Internal error
`EW_EALIGN`	3	Alignment error
`EW_EOVERRUN`	4	Message overrun
`EW_ENOMEM`	5	Memory exhausted
`EW_EINIT`	6	Initialisation error
`EW_EMISMATCH`	7	Tx and Rx size discrepancy
`EW_ERANGE`	8	Value out of range
`EW_ENOTSTARTED`	9	Communication not started
`EW_ETIMEOUTSET`	10	Timeout already set
`EW_EIO`	11	I/O error
`EW_ENOROUTES`	12	No more route tables
`EW_EUSERBASE`	1000000	Unknown exception

`ew_exception()` may be attached to a signal handler by setting the envirc ment variable `LIBEW_DEBUGSIG`. Receipt of the nominated signal then caus an exception with code `EW_DEBUG`. The environment variables which affect e ception handling, and their values are listed below. An exception with code E `EINIT` is generated if the values can't be parsed.

Environment Variable	Values	Description
LIBEW_DEBUGSIG	integer	Initialise handler to set a debugging exception on receipt of a signal. Default: disabled
LIBEW_CORE	1,YES 0,NO	Enable core dump on exception. Default: disabled unless linking with libew_dbg.a
LIBEW_TRACE	1,YES 0,NO	Enable trace dump on exception. Default: disabled unless linking with libew_dbg.a

ew_touchBuf **pre-fault memory**

Synopsis

```
#include <ew/ew.h>
void ew_touchBuf (caddr_t buf, int nob);
```

Description

`ew_touchBuf()` causes a read and a write access, by both the main process and the elan, to every page of the buffer with base `buf` and size `nob` in bytes. This causes a page fault on any non-resident pages on either processor.

This procedure has not been optimised for speed. It has been designed to be called infrequently, typically during start-up initialisations.

Warning – Pages which overlap less than 1 word of `buf` are not touched.

ew_usleep **suspend execution for a period**

Synopsis

```
#include <ew/ew.h>
void ew_usleep (u_int t);
```

Description

The calling process is suspended for the period `t` micro-seconds. The actual period of suspension may be less than that requested as `ew_usleep()` returns immediately if a signal is delivered.

This procedure is implemented using `select()`. It does not interfere with `SIGALRM` handling.

ew_utimeout — **schedule a procedure for execution after an interval**

Synopsis

```
#include <ew/ew.h>
void ew_utimeout (u_int timeout, void (*handler)(int))
```

Description

`ew_utimeout()` uses the interval timer to schedule `SIGALRM`, so that a pr cedure may be executed after an interval specified in micro-seconds. A non-ze timeout is specified to schedule the procedure, and a zero timeout is specified restore the previous timer settings and signal handlers.

Only one procedure at a time may be scheduled with `ew_utimeout()`. At- tempts to schedule more than one procedure at a time cause an exception witl code `EW_ETIMEOUTSET`.

With a non-zero `timeout`, `ew_utimeout()` samples the current interval ti er settings. If the interval timer is set to expire before `timeout`, `timeout` i reduced so that `handler` can be scheduled first. The current interval timer s tings are then adjusted to account for `timeout`, and `handler` is installed a the `SIGALRM` handler. The interval timer is then re-enabled to schedule `SIGA RM`.

Passing a `timeout` of zero restores the old timer and signal settings. This ma be done at any time. Doing this before `timeout` has expired, pre-empts the e ecution of `handler`. Note that any interval timer or `SIGALRM` handler install before the call to `ew_utimeout()` will **not be restored** until `ew_utime- out()` is called with a zero timeout.

Warning – Applications which make sophisticated use of the interval tim and `SIGALRM` may find usage clashes with `ew_utimeout()`.

---

ew_dbg — **print a debugging message**

Synopsis

```
#include <ew/ew.h>
void ew_dbg (char *format, ...);
```

Description

Write a debugging message to the standard output stream `stderr`. `format` is a `printf()` style format string. After a successful `ew_attach()`, `ew_dbg()` prefixes the formatted message with the calling process's virtual process number. If it is called before attaching to the network, `ew_dbg()` prefixes "`----: `".

ew_eventStr — **return event state string**

Synopsis

```
#include <ew/ew.h>
char *ew_eventStr (ELAN_EVENT *e);
```

Description

Return a string containing a textual representation of the elan event `e`. The returned string is a pointer into a static buffer which will eventually be overwritten on subsequent calls to `ew_eventStr()`. Currently up to 64 calls may be made before this space is re-used.

Returned strings have the following formats.

String	Description
`QUEUE`	Event is a queued event.
`READY`	Event is set.
`CLEAR`	Event is clear.
`INT_W 00000004`	Interrupt waiting: Signal number, shifted left 2.
`P___W e0000f80`	Thread waiting: Stack pointer.
`DMA_W e00008ef0`	Dma waiting: Dma descriptor address.
`NUL_W 00000000`	Null waiting.
`LE__W e000eeb0`	Local event waiting: Event address.
`RE__W e000eeb0 @ 3`	Remote event waiting: Event, virtual process number.

ew_rup2 **round up to a power of 2**

Synopsis

```
#include <ew/ew.h>
u_int ew_rup2 (u_int n);
```

Description

Round up n to a power of 2.

ew_gray

ew_gray, ew_ginv — gray code conversions

Synopsis

```
#include <ew/ew.h>
u_int ew_gray (u_int n);
u_int ew_ginv (u_int n);
```

Description

`ew_gray()` returns the gray code of argument `n`. It converts integers which d
fer by 1 to integers which differ by a power of 2. `ew_ginv()` is the inverse
function.

The tables below enumerate the function for small binary integers.

n	ew_gray(n)
0	0
1	1
10	11
11	10
100	110
101	111
110	101
111	100

n	ew_gray(n)
1000	1100
1001	1101
1010	1111
1011	1110
1100	1010
1101	1011
1110	1001
...	...

ew_bitFlip **bitwise mirror**

Synopsis

```
#include <ew/ew.h>
u_int ew_bitFlip (u_int nVals, u_int n);
```

Description

`ew_bitFlip()` performs a bitwise mirror on an integer. All bits in `n` from the least significant, to highest bit set in `nVals`, inclusive, are mirrored.

`ew_bitFlip()` may be used to enumerate a range of integers with the most significant bit varying most rapidly. The tables below show binary values for the function and its argument for small integers.

n	ew_bitFlip(7,n)
000	000
001	100
010	010
011	110
100	001
101	101
110	011

ew_putenv — **change or add to the environment**

Synopsis

```
#include <ew/ew.h>
int ew_putenv (char *str);
```

Description

`ew_putenv()` creates, or assigns to an existing environment variable. `str` must be in the format `"name=value"`. It is similar to `putenv()`, however manages its own pool of environment strings. `ew_putenv()` may therefore called repeatedly with arbitrary strings without the possibility of losing memo or of overwriting environment variables that were set previously.

`ew_putenv()` returns 0 on success and -1 if it fails to allocate sufficient me ory on calling `malloc()`.

ew_getenvCap

ew_getenvCap, ew_putenvCap — extract and set elan capabilities in the environment

Synopsis

```
#include <ew/ew.h>
int ew_getenvCap (ELAN_CAPABILITY *cap, int index);
int ew_putenvCap (ELAN_CAPABILITY *cap, int index);
```

Description

`ew_getenvCap()` searches the environment for a capability for segment `index` of a parallel application. It returns 0 on success, having assigned the value of the capability to `*cap`. On failure, it returns -1.

`ew_putenvCap()` sets the environment variable capability string for segment `index` to `*cap`. On success it returns 0. It may fail if `malloc()` can't allocate sufficient memory, in which case it returns -1.

Both procedures take a non-negative value of `index` to mean a specific segment number. A negative value of `index` is used to retrieve or set the process's own segment capability.

See Also

`ew_init()`.

ew_createBcastVp

ew_createBcastVp, ew_destroyBcastVp — broadcast virtual process numbers

Synopsis

```
#include <ew/ew.h>
u_int ew_createBcastVp (u_int base, int count);
void ew_destroyBcastVp (u_int vp);
```

Description

`ew_createBcastVp()` allows the user to create a virtual process number name a contiguous range of processes. The nominated set includes `count` pr esses, starting with the process with virtual process number `base`. This set m not cross segment boundaries. The return value is the new virtual process number.

Passing illegal values of `base` or `count` causes an `EW_ERANGE` exception. Failure to allocate internal memory structures to administer the broadcast virtu process numbers causes an `EW_ENOMEM` exception. Failure to allocate additio elan route table entries causes an `EW_ENOROUTES` exception.

`ew_destroyBcastVp()` makes broadcast virtual process number `vp` ava ble for re-assignment via `ew_createBcastVp()`. Note that the value of is not checked in this procedure. Behaviour on passing values not previously turned by `ew_createBcastVp()` is **not defined**.

See Also

`ew_init()`, `EW_GROUP`, `EW_BCHAN`, `ew_storeStart()`.

ew_allocate

ew_createAllocator, ew_spawnAllocator, ew_destroyAllocator, ew_allocate, ew_free — global heap management

Synopsis

```
#include <ew/ew.h>
typedef void EW_ALLOC;
EW_ALLOC *ew_createAllocator (caddr_t base, int size);
EW_ALLOC *ew_spawnAllocator (EW_ALLOC *a, int size);
void *ew_allocate (EW_ALLOC *a, int align, int size);
void ew_free (void *ptr);
void ew_destroyAllocator (EW_ALLOC *a);
```

Description

The `libew` allocation procedures perform heap management over given virtual address ranges. Processes that create heaps spanning identical virtual address ranges and that synchronise their calls to these procedures are guaranteed to allocate objects at identical virtual addresses. This allows them to manage heaps of global objects.

`EW_ALLOC` is an opaque data-type. Objects of this type are returned as handles on a particular heap by the heap creation procedures.

`ew_createAllocator()` creates a new heap spanning the range of virtual addresses starting at `base`, rounded up to a page boundary, and ending at `base + size` rounded down to a page boundary, where `size` is in bytes.On success `ew_createAllocator()` returns a non-`NULL` pointer. Otherwise it returns `NULL` and sets `errno` to indicate the error.

Error Number	Description
`EINVAL`	Less than one page in region after rounding.
`ENOMEM`	Failed to `malloc()` data structures.

`ew_spawnAllocator()` creates a new heap by partitioning an existing one. It reserves a region of `size` bytes rounded up to a whole number of pages in the parent allocator `a`. On success, `ew_spawnAllocator()` returns a non-`NULL` pointer. Otherwise it returns `NULL` and sets `errno` according to the table above.

`ew_allocate()` allocates `size` bytes with alignment `align` from allocator `a`. Alignments are restricted to powers of 2. A minimum alignment of 8 bytes is silently enforced. `ew_allocate()` first attempts to allocate from a pool of

mapped memory owned by the specified allocator. If this pool is too small, it tempts to grow the pool by mapping some more memory from `/dev/zero` within the virtual address range owned by the allocator.

On success, `ew_allocate()` returns a pointer to the base of the allocated gion. Otherwise it returns a `NULL` pointer with `errno` set to indicate the type error.

Error Number	Description
`EINVAL`	Alignment not a power of 2.
`ENOMEM`	Heap exhausted.
`EAGAIN`	An attempt to map additional memory in the range owned by the allocator failed due to insufficient swap space.

`ew_free()` returns space previously allocated by `ew_allocate()` to the owning allocator. Behaviour is undefined if a pointer passed to `ew_free()` w not allocated by `ew_allocate()`, if the administrative information stored l low the allocated space has been corrupted, or if the heap from which it was located has been destroyed.

`ew_destroyAllocator()` unmaps any memory which may have been mapped on behalf of allocator `a`. If `a` has a parent, it resumes ownership of th allocator's virtual address range. Any objects previously allocated in `a` may r longer be de-referenced.

EW_GROUP

process group

Synopsis

```
#include <ew/ew.h>
```

Description

An EW_GROUP defines a group of processes that wish to cooperate in barriers, reductions and collective communications. It is a global object i.e it exists at the same virtual address in all its members.

```
typedef struct ew_group
{
   int         g_self;
   int         g_size;
   u_int       g_bcastVp;
   u_int     (*g_lookupFn)(struct ew_group *g, int member);
   long        g_lookupParams[16];
   /* Additional private contents */
}        EW_GROUP;
```

Groups number their members contiguously from 0. g_self gives the member number and g_size gives the total number of processes in the group.

Groups consisting of a contiguous set of processes contained within a single segment, may exploit hardware broadcast and use the virtual process number g_bcastVp to address all processes in the group. If hardware broadcast is not possible or not desired, g_bcastVp is ELAN_INVALID_PROCESS and all group operations are conducted only using point-to-point communications.

The set of processes which actually constitute the group is not explicitly represented. Group membership is determined by a lookup function which maps group member number to virtual process number. Applying it to all integers between 0 and g_size-1 inclusive, enumerates the group's processes.

The group membership function is stored in g_lookupFn. Additional parameters, as required by individual lookup function, may also be stored in the group. This allows for arbitrary irregular or table driven membership functions.

Group operations such as reduction are performed on a spanning tree with a user defined maximum branching ratio. It is balanced, rooted in group member 0 and all sub-trees contain a contiguous range of group members. All group operations are deterministic. Sub-trees are always processed strictly in order and reduction results are distributed from the root so that every member receives identical results.

Performance related parameters are set when a group is initialised. These inclu
the spanning tree branching ratio, the packet size used to pass data over it, whe
er to block or to poll for completion, and how to DMA data through the netwo

A group may only be used once all its members have initialised it. This is m
conveniently done with a barrier on parent group. The first group ever to be c
ated has no parent group, however a special barrier, `ew_sgsync()` is used
perform the start-up synchronisation.

See Also

`ew_init()`, `ew_base`, `ew_createBcastVp()`, `ew_groupInit()`
`ew_groupMember()`, `ew_gsync()`, `ew_bcast()`, `ew_reduce()`
`ew_gprintf()`, `EW_GEX`.

ew_groupInit **ew_groupSize, ew_groupInit, ew_groupFini — group administration**

Synopsis

```
#include <ew/ew.h>
int ew_groupSize (int pktSize, int branch);
void ew_groupInit (EW_GROUP *g,int pktSize,int branch,
                   int waitType, int dmaType,
                   int self, int size, u_int bcastVp,
                   void(*initFn)(EW_GROUP *g,void*va),
                   ...);
void ew_groupFini (EW_GROUP *g);
```

Description

`ew_groupSize()` returns the number of bytes which should be allocated to represent a group which uses packet size `pktSize` and a spanning tree with branching factor `branch`.

`ew_groupInit()` initialises the local slice of a group. All group members must have initialised their slice before the group can be used. With the exception of `self`, all group members must pass identical initialisation parameters. This includes `g`, as group is a global object Behaviour is undefined if any of these rules are broken.

The group `g` must be aligned on an `EW_ALIGN` boundary, otherwise an exception with code `EW_EALIGN` is generated. It is initialised to use a packet size of `pktSize`. `branch` sets the group's spanning tree branching factor. `waitType` determines how group operations should block for completion. `dmaType` controls how data is transferred between group members.

`self` is the calling process's own group member number and `size` is the total number of group members. If `self` is not in the range 0 to `size-1`, an exception with code `EW_ERANGE` is generated.

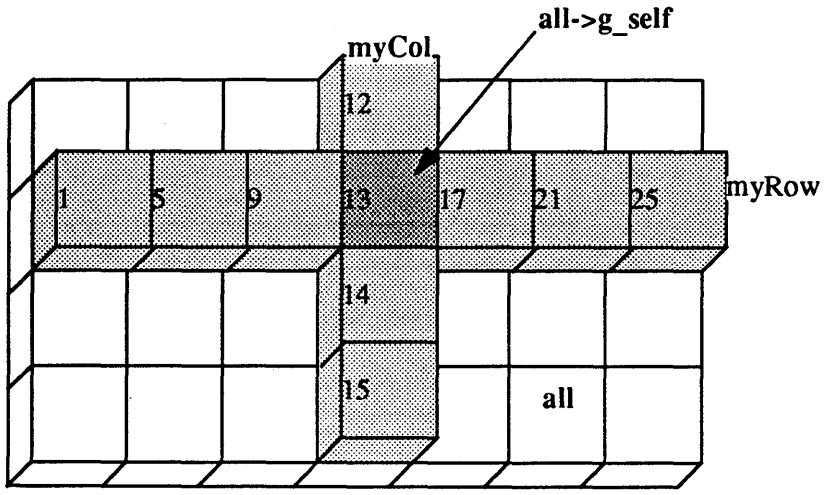
If `ELAN_INVALID_PROCESS` is passed in `bcastVp`, all group operations will be conducted only using point-to-point communications. Otherwise `bcastVp` is assumed to be a broadcast virtual process number which addresses **all the group members, and no others**. Behaviour is undefined if any of these rules are broken.

`initFn` is an initialisation procedure which is used to initialise the group's member lookup function. Note that it is **not** the lookup function itself. It is only ever called once, from `ew_groupInit()`. It is passed a pointer to the group and a `va_list` in `stdarg` format, which points to the additional arguments

passed to `ew_groupInit()`. It must initialise the group's `ew_lookupFn` and it may store any additional arguments required by this function in `g_lookupParams`.

The following example shows the lookup function `lookupId()` which defin groups of regularly strided processes. `ew_groupFn_slice()`, its initialis tion function, is passed to `ew_groupInit()` along with the virtual proces number of group member 0 and the process stride. It is used in the example t describe a 2D process decomposition of a parent group `all`. Each process in t application becomes a member of a column group, in which all members are c tiguous, and a row group, in which all members are strided by `nrow`. Each pr ess is therefore a member of three groups, `all`, `myRow` and `myCol`. Howev from the point of view of the whole application, there are twelve groups. On group which includes all processes, four row groups and seven column grou

Figure 1-8 2D Process Decomposition.



`libew` provides the following pre-defined group initialisation functions.

```
typedef struct
{
   u_int   baseVp;
   int     stride;
}  G1D;

static u_int    lookup1d (EW_GROUP *g, int i)
{
   G1D           *params = (G1D *)g->g_lookupParams;

   return (params->baseVp + i *params->stride);
}

void            ew_groupFn_1d (EW_GROUP *g, va_list ap)
{
   G1D            *params = (G1D *)g->g_lookupParams;

   g->g_lookupFn = lookupSlice;
   params->baseVp = va_arg (ap, int);
   params->stride = va_arg (ap, int);
}

...

int nrow    = 4;
int ncol    = 7;
int me      = all->g_self;
int row     = me % nrow;
int col     = me / nrow;

ew_groupInit (myRow, pktSize, branch, waitType, dmaType,
              col, ncol, ELAN_INVALID_PROCESS,
              ew_groupFn_1d,
              ew_groupMember (all, row), nrow);

ew_groupInit (myCol, pktSize, branch, waitType, dmaType,
              row, nrow, elan_createBcastVp (col * nrow, nrow),
              ew_groupFn_1d,
              ew_groupMember (all, col * nrow), 1);

ew_gsync (all);

/* Group operations on myRow and myCol */
```

Function	Description
`ew_groupFn_all()`	Every process in the application. No additional arguments.
`ew_groupFn_seg()`	Every process in the caller's segment. No additional arguments.
`ew_groupFn_1d()`	An array of regularly strided processes. Additional arguments: `baseVp, stride` `baseVp` is the virtual process number of group member 0 and `stride` is the offset between group members.
`ew_groupFn_table()`	Any irregular group of processes determined by table lookup. Additional arguments: `tablep` `tablep` is a pointer to any array of virtual process numbers indexed by group member number.

`ew_groupFini()` finalises a group. It is a barrier, therefore all group membe must call it before any member can complete. On return, the memory used by t group may be freed or re-used.

See Also `EW_GROUP, ew_groupMember()`

ew_groupMember — **return the virtual process number of a group member**

Synopsis

```
#include <ew/ew.h>
u_int ew_groupMember (EW_GROUP *g, int i);
```

Description

`ew_groupMember()` returns the virtual process number of member `i` of group `g`. It is implemented as a macro which invokes the group's member lookup function, passing it both of its arguments.

`ew_groupMember()` performs no bounds checking on `i`, however this may be done by the group's installed lookup function. The lookup functions provided by `libew` do not perform bounds checking, however they do in the debugging version of the library, `libew_dbg`.

See Also

`EW_GROUP`, `ew_groupInit()`

ew_gsync — **ew_gsync, ew_fgsync, ew_sgsync — barrier synchronise a group**

Synopsis

```
#include <ew/ew.h>
void ew_gsync (EW_GROUP *g);
void ew_fgsync (EW_GROUP *g);
void ew_sgsync (EW_GROUP *g);
```

Description

All these procedures perform a barrier synchronisation, but by different metho

`ew_gsync()` synchronises by reducing "ready" in the group's spanning tre Member 0, at the root of the tree, then signals "ready" back to the group. If t group has a broadcast virtual process number, hardware broadcast is used to s nal the rest of the group, otherwise "ready" is propagated back down the tree.

`ew_fgsync()` requires hardware broadcast, so if the group does not have a broadcast virtual process number, it just calls `ew_gsync()`. Otherwise it sy chronises by reducing "ready" in the network. This is performed using a broa cast network poll which is repeated until all group members have joined the barrier. "Ready" is then broadcast to the group. This method is faster than `ew gsync()` because the reduction is performed at switch latencies rather than whole network latencies, however the polling may have an effect on commun cations that any group members are performing prior to joining the barrier.

`ew_sgsync()` is a special barrier designed for start-up synchronisation. It blocks until all group members have attached to the network and initialised th slice of the group. On all subsequent calls, it just calls `ew_gsync()`. Progra mers using the recommended start-up initialisation `ew_baseInit()` never need to call this procedure.

See Also

`ew_init()`, `ew_base`, `EW_GROUP`.

ew_bcast

ew_bcast, ew_fbcast — broadcast over a group

Synopsis

```
#include <ew/ew.h>
void ew_bcast (EW_GROUP *g, int src,
               caddr_t buf, int size, int global);
void ew_fbcast (EW_GROUP *g, int src,
                caddr_t buf, int size, int global);
```

Description

Both procedures spread `size` bytes in `buf` at group member `src` over all members of the group `g`. `global` should be `TRUE` if `buf` has the same value in all group members (i.e `buf` is a global object). The procedures differ in the manner in which the group members synchronise.

`ew_bcast()` reduces "ready" in the group's spanning tree. When all group members are prepared to receive the data, it distributes it using a network broadcast if the group has a broadcast virtual process number, and via the group's spanning tree, if it does not. If `global` is `TRUE` and hardware broadcast is possible, the data is transferred in a single network DMA directly to `buf`. Otherwise the data transfer is packetised via the group's network buffers. In this case, if hardware broadcast is possible, data is transferred directly from the sender to the rest of the group, otherwise it is spread via the group's spanning tree.

`ew_fbcast()` requires hardware broadcast, so if the group does not have a broadcast virtual process number, it just calls `ew_bcast()`. Otherwise it reduces "ready" in the network by a broadcast network poll (see `ew_fgsync()`). It then distributes the data using a broadcast network DMA. If `global` is `TRUE` all the data is transferred in a single DMA, otherwise, the data transfer is packetised via the group's packet buffers.

See Also

`ew_init()`, `ew_base`, `EW_GROUP`, `ew_gsync()`

ew_reduce — **user defined reduction over a group**

Synopsis

```
#include <ew/ew.h>
void ew_reduce (EW_GROUP *g, void (*userFn)(),
                caddr_t els, int elsz, int nel, int global
void userFn (caddr_t accum, caddr_t part, int *nelp)
```

Description

`ew_reduce()` applies the user defined reduction function `userFn()` ove members of group `g`, to an array of `nel` elements of a user-define datatype c size `elsz` referenced by `els`. The results are stored back to `els` in all grou members. An exception with code `EW_ERANGE` is generated if the element s is greater than the group's packet size.

`global` may be set to `TRUE` if all group members have identical values of e (i.e. `els` is a global object) to eliminate unnecessary copying. If the array of ements is larger than the group's packet size, the reduction is packetised ove whole numbers of elements. If the element size is larger than the group's pac size, an exception is caused.

The reduction function `userFn()` is called with `accum` and `part` which r erence the reduction results from two contiguous sets of contiguous group me bers. It must reduce `*nelp` elements of the user's datatype into `accum`.

Warning – `*nelp` must not be updated by `userFn()`, otherwise behavio is undefined.

For example if `accum` is the reduction result from group members 7 to 13 a `part` is the reduction result from group members 14 to 20, then when `userFn()` returns, `accum` contains the reduction result of group members 7 20.

The following example shows the `libew` implementation of the familiar reduction function `gdsum()`.

```
static void     dsum (double *accum, double *partial, int *nel)
{
   int              n = *nel;

   while (n-- > 0)
      *accum++ += *partial++;
}

void            gdsum (double *x, int nelem, double *work)
{
   ew_reduce (ew_base.segGroup, dsum,
              (caddr_t) x, sizeof (double), nelem, 0);
}
```

See Also

`EW_GROUP, ew_groupInit(), ew_base.`

ew_gprintf

ew_gprintf, ew_gvprintf — print formatted output in a group

Synopsis

```
#include <ew/ew.h>
void ew_gprintf (EW_GROUP *g, int meToo,
                 char *format, ...);
void ew_gvprintf (EW_GROUP *g, int meToo,
                  char *format, void *ap);
```

Description

Both procedures synchronise the printing of `printf()` style formatted mess
es by members of group `g`. They are both an implicit barrier, however group members with no message to print may pass `FALSE` in `meToo`.

`ew_gprintf()` takes a format string and any additional arguments. `ew_gvprintf()` takes a format string and a `stdarg` variable argument list.

Each group member formats the message into a local static buffer, and then sen
this message to group member 0, who prints it on `stdout`. Messages are print
in order of group member number.

**Warning – The maximum size of the message after formatting is `1024` byt
including the terminating `NULL` character.**

See Also

`EW_GROUP`, `ew_groupInit()`

EW_GEX

EW_GEX, ew_gexSize, ew_gexInit — global exchange

Synopsis

```
#include <ew/ew.h>
typedef void EW_GEX;
int ew_gexSize (EW_GROUP *group);
void ew_gexInit (EW_GEX *gex, EW_GROUP *group,
                 iovec_t *txIov, iovec_t *rxIov,
                 int waitType, int dmaType);
```

Description

An `EW_GEX` provides an all-to-all, global exchange. It is a global object i.e. it is at the same virtual address in all its participating processes. A non-blocking interface is supported which allows a single outstanding exchange. It is initialised with local source and destination buffers, and the group over which it will operate. The buffers may be irregularly placed and sized. The only constraints are that there must be one source and one destination buffer for every group member, and that the source buffer size of the sending process matches the destination buffer size of the receiving process. This information is "compiled" into a DMA list which is activated on subsequent calls to perform the exchange.

`ew_gexSize()` returns the size of a gex in bytes.

`ew_gexInit()` is an implicit barrier. It initialises `gex` to operate over the members of group `group`. `gex` must be aligned to an `EW_ALIGN` boundary otherwise an exception is generated with code `EW_EALIGN`.

`txIov` and `rxIov` point to a pair of arrays of buffer descriptors, each having one entry for every member of the group including the caller. `txIov[i]` gives the location and size of the data to be sent to group member `i`. `rxIov[i]` gives the location and size of the data to be received from group member `i`. An exception with code `EW_EMISMATCH` arises if group member `i`'s `txIov[j].iov_len` is not equal to group member `j`'s `rxIov[i].iov_len`.

`waitType` determines how to block for completion and `dmaType` determines how to transfer data for all subsequent gex operations.

See Also

`EW_GROUP`, `ew_gexStart()`

ew_gexStart

ew_gexStart, ew_gexDone, ew_gexWait — perform a global exchange

Synopsis

```
#include <ew/ew.h>
void ew_gexStart (EW_GEX *gex);
int ew_gexDone (EW_GEX *gex);
void ew_gexWait (EW_GEX *gex);
```

Description

`ew_gexStart()` initiates a non-blocking global exchange, previously initi ised in `gex`. The next global exchange on `gex` may be initiated only after calli `ew_gexWait()`.

Warning – No initial synchronisation is performed by `ew_gexStart()` The caller must ensure that all other group members are ready to partic pate in the exchange before calling it.

`ew_gexDone()` returns `TRUE` if the global exchange outstanding on `c` has completed and `FALSE` if it has not. Behaviour is undefined if no exchange is o standing on `gex`.

`ew_gexWait()` blocks until the global exchange outstanding on `gex` has co pleted i.e. that all the caller's source buffers have been transmitted and all its d tination buffers have been filled. On return, the caller may make a further call `ew_gexStart()`.

See Also

`EW_GEX`

EW_DMAPOOL

EW_DMAPOOL, ew_dmaPoolCreate, ew_dmaPoolDestroy — network DMA pool

Synopsis

```
#include <ew/ew/h>
typedef void EW_DMAPOOL;
EW_DMAPOOL *ew_dmaPoolCreate (int waitType,
                              int dmaType);
void ew_dmaPoolDestroy (EW_DMAPOOL *dp);
```

Description

An `EW_DMAPOOL` is an opaque data structure used to administer network DMA descriptors. It is passed to `libew` procedures which initiate direct access to non-local memory.

`ew_dmaPoolCreate()` creates a DMA pool and returns its handle. It uses `malloc()` to allocate the space and generates an exception with code `EW_ENOMEM` if it fails. `waitType` determines how to block for completion and `dmaType` determines how to transfer data for all DMAs initiated from this pool.

`ew_dmaPoolDestroy()` deallocates the DMA pool `dp` and all DMA descriptors that it owns. Calling this procedure while dmas are active causes an exception with code `EW_EBUSY`.

See Also

`ew_storeStart()`, `ew_fetchStart()`.

ew_storeStart **ew_storeStart, ew_storeDone, ew_storeWait — store to non-local memo**

Synopsis

```
#include <ew/ew.h>
ELAN_EVENT *ew_storeStart (EW_DMAPOOL *dp,
                           caddr_t buf, int nob,
                           u_int destProc, caddr_t destBu
                           ELAN_EVENT *destEvent);
int ew_storeDone (ELAN_EVENT *e);
void ew_storeWait (ELAN_EVENT *e);
```

Description

`ew_storeStart()` initiates a store to a remote memory location. It alloca a DMA descriptor from pool `dp` to copy `nob` bytes of data at `buf` in local me ory, to `destBuf` in the address space of the process with virtual process numl `destProc`. If `destEvent` is `NULL` it is ignored, otherwise it specifies an ev to set at the destination on completion of the DMA. It returns an event pointer the handle on the DMA it has initiated . This should be passed to the polling a completion procedures. Note that completion means that the store at the desti tion has been updated and that the destination event, if specified, has been se

`destProc` may be a broadcast virtual process number. If it is, `destBuf` is written and `destEvent`, if it is not `NULL`, is set in all destination processes

Warning – If `destProc` is a broadcast virtual process number, `destEve` may be set more than once in some destination processes.

`ew_storeStart()` grows the pool of DMA descriptors with `malloc()` all descriptors in `dp` are currently in use. If this fails, it causes an exception w code `EW_ENOMEM`.

`ew_storeDone()` is used to test for completion of a remote store. It neve blocks, but returns `TRUE` if the DMA with handle `e` has completed. Otherwis returns `FALSE`.

`ew_storeWait()` blocks until the DMA with handle `e` has completed. It turns the descriptor back to its owning pool.

The following example shows `ew_storeStart()` being used to do an edge exchange between an arbitrary number of neighbouring processes.

```
#define NOB 1024
#define NNBR 6
extern u_int nbrVp[NNBR];          /* my neighbours */
char  exportBufs[NNBR][NOB];
char *importBufs[NNBR];
ELAN_EVENT *e[NNBR];
int i;
EW_DMAPOOL *dp;

/* create dma pool */
dp = ew_dmaPoolCreate (ew_base.waitType, ew_base.dmaType);

/* allocate importBufs at globally consistent addresses */
for (i = 0; i < NNBR; i++)
   importBufs[i] = ew_allocate (ew_base.alloc, EW_ALIGN, NOB);
...
ew_fgsync (ew_base.segGroup);     /* wait for neighbours ready */

for (i = 0; i < NNBR; i++)        /* ship the data */
   e[i] = ew_storeStart (dp, exportBufs[i], NOB,
                         nbrVp [i], importBufs[i], NULL);

for (i = 0; i < NNBR; i++)        /* block until shipped */
   ew_storeWait (e[i]);

ew_fgsync (ew_base.segGroup);     /* Exchange complete */
```

See Also

`EW_DMAPOOL`, `ew_createBcastVp()`.

ew_fetchStart

ew_fetchStart, ew_fetchDone, ew_fetchWait — fetch non-local memory

Synopsis

```
#include <ew/ew.h>
ELAN_EVENT *ew_fetchStart (EW_DMAPOOL *dp,
                           caddr_t buf, int nob,
                           u_int srcProc, caddr_t, srcBuf)
int ew_fetchDone (ELAN_EVENT *e);
void ew_fetchWait (ELAN_EVENT *e);
```

Description

`ew_fetchStart()` initiates a fetch from a remote memory location. It all cates a DMA descriptor from pool `dp` to copy `nob` bytes of data at `srcBuf` the address space of the process with virtual process number `srcProc` to bu in local memory. It returns an event pointer as the handle on the DMA it has itiated. This should be passed to the polling and completion procedures.

Warning – Behaviour is undefined if `srcProc` is a broadcast virtual proce number.

`ew_fetchStart()` grows the pool of DMA descriptors with `malloc()` all descriptors in `dp` are currently in use. If this fails, it causes an exception wi code `EW_ENOMEM`.

`ew_fetchDone()` is used to test for completion of a remote fetch. It never blocks, but returns `TRUE` if the DMA with handle `e` has completed. Otherwise returns `FALSE`.

`ew_fetchWait()` blocks until the DMA with handle `e` has completed. It r turns the descriptor back to its owning pool.

Note that remote fetch requires an additional network latency compared with mote store.

The following example shows `ew_fetchStart()` being used to do an edge exchange between an arbitrary number of neighbouring processes.

```
#define NOB 1024
#define NNBR 6
extern u_int nbrVp[NNBR];          /* my neighbours */
char  importBufs[NNBR][NOB];
char *exportBufs[NNBR];
ELAN_EVENT *e[NNBR];
int i;
EW_DMAPOOL *dp;

/* create dma pool */
dp = ew_dmaPoolCreate (ew_base.waitType, ew_base.dmaType);

/* allocate exportBufs at known addresses */
for (i = 0; i < NNBR; i++)
   exportBufs[i] = ew_allocate (ew_base.alloc, EW_ALIGN, NOB);
...
ew_fgsync (ew_base.segGroup);     /* wait for neighbours ready */

for (i = 0; i < NNBR; i++)        /* grab the data */
   e[i] = ew_fetchStart (dp, importBufs[i], NOB,
                         nbrVp[i], exportBufs[i]);

for (i = 0; i < NNBR; i++)        /* block until grabbed */
   ew_fetchWait (e[i]);

ew_fgsync (ew_base.segGroup);     /* Exchange complete */
```

See Also

`EW_DMAPOOL`.

EW_PFD

EW_PFD, ew_pfInit, ew_pfopen, ew_pfclose — parallel file I/O

Synopsis

```
#include <ew/ew.h>
typedef void EW_PFD;
void ew_pfInit (int waitType, int dmaType);
EW_PFD *ew_pfopen (EW_GROUP *g, char *path, int flags
                   /* int mode */ ...);
int ew_pfclose (EW_PFD *pfd);
```

Description

An `EW_PFD` is an opaque data type used to represent a file that has been open by a group of processes for reading or writing global objects. These processes operate to read and write the file in parallel, according to the file's filesystem type, and convert from a canonical representation in the file system, to the distribution required by the application.

Note that an `EW_PFD` is not a global object. It is only meaningful in the proce that called `ew_pfopen()`.

`ew_pfInit()` initialises the dma pool which is used to redistribute global objects on their way to and from the file system. `waitType` controls how parall file operations will block on remote fetch and store, and `dmaType` determine how they will transfer the data.

`ew_pfopen()` opens file `path` for parallel file I/O in all members of group Group member 0 sets the file's streaming factor, `f`, and broadcasts it to the oth members. `f` determines the number of parallel streams in which the file is rea and written. If the file does not reside on a `PFS` file system, `f` is set to `1`, othe wise it is set to the minimum of the number of underlying data file systems a the number of group members. The first `f` group members then open the file wi the Unix `open()` system call, passing it `path`, `flags` and `mode`.

Warning – `ew_pfopen()`, and all other operations on the parallel file d scriptor returned by it perform a barrier on the members of group `g`. Th group must remain in existence until after the file has been closed.

`ew_pfopen()` causes an exception with code `EW_EINIT` if it is called befc `ew_pfInit()`. If it fails to open the file successfully, it returns `NULL`, with `errno` set to indicate the error. Otherwise it returns the parallel file descript

`ew_pfclose()` closes the file associated with parallel file descriptor `pfd`. It is an implicit barrier on the group with which the file was opened.

See Also

`EW_DST`, `ew_pfseek()`, `ew_pfread()`, `ew_pfwrite()`

EW_DST

EW_DST, ew_dstCreate, ew_dstDestroy — describe a regular matrix distribution

Synopsis

```
#include <ew/ew.h>
typedef struct ew_dst EW_DST;
EW_DST *ew_dstCreate (int n,
                      /* int nAtom, int nProc, int blk, */
                      ...);
void ew_dstDestroy (EW_DST *d);
```

Description

An EW_DST describes a distribution of an *n*-dimensional matrix over a group Each dimension of the matrix is divided into blocks which are cyclically assign to one or more processors in the group. The distribution is regular both in gro member number and virtual address.

ew_dstCreate() creates a distribution description. n specifies the number dimensions in the matrix. For each dimension, nAtom is the number of elemer in that dimension of the matrix, nProc is the number of processors in that di mension of the process array, and blk is the number of contiguous atoms to sign to each processor.

Warning – Note that nAtom and blk must be specified in units of bytes f the most rapidly varying dimension (the last triplet of parameters).

ew_dstCreate() allocates an EW_DST with malloc(), computes the di tribution information for all dimensions and returns a pointer to it. If the allo tion fails, it returns NULL.

ew_dstDestroy() frees the memory allocated for the descriptor d.

The descriptor returned by ew_dstCreate() contains an array of entries f each dimension of the distribution, plus an additional entry for summary info mation. Each entry contains count and stride information for the process arra the slices of the matrix and the matrix itself.

EW_DST component	Description
int d_ndim	Number of dimensions in the matrix.
EW_DSTDIM d_dim[]	Dimension information (d_ndim+1 entries).

EW_DSTDIM component	Description
`long d_blocksize`	Number of atoms per block.
`long d_ngatom`	Number of atoms in this dimension of the matrix.
`long d_gstride`	The stride, in bytes, between atoms in this dimension of the matrix.
`long d_nsatom`	Maximum number of atoms in any slice of this dimension of the matrix. Note that some processes have fewer atoms if the number of atoms is not exactly divisible by the number of processes in this dimension.
`long d_sstride`	The stride in bytes between atoms, in all slices of this dimension of the matrix.
`long d_nproc`	Number of processes in this dimension of the process array.
`long d_pstride`	The stride between processes in this dimension of the process array.

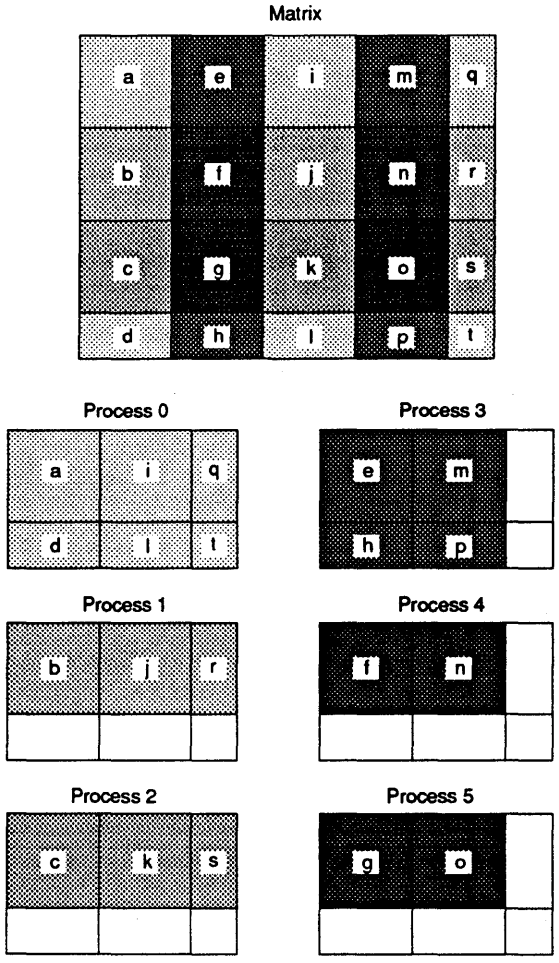
Example

The method of representing the distribution can be more easily described with an example. Consider a matrix of double precision floating point numbers, with dimensions [18][35], distributed over a process array with dimensions [2][3], with blocking factors [4][10].

```
double M[18][35];
EW_DST *d = ew_dstCreate(2,
              sizeof (M)/sizeof(M[0]), 2,                    4,
                       sizeof (M[0]), 3, sizeof (M[0][0])*10);
```

This results in a [5][4] array of blocks assigned cyclically to the [2][3] proces
array. Note that the last block in each dimension contains an odd number of el
ments.

Figure 1-9 Matrix Distribution



1

d_dim	blocksize	ngatom	gstride	nsatom	sstride	nproc	pstride
[0]	1	1	5040	1	1200	1	6
[1]	4	18	280	10	120	2	3
[2]	80	280	1	120	1	3	1

ew_pfseek — **parallel file seek.**

Synopsis

```
#include <ew/ew.h>
long ew_pfseek (EW_PFD *pfd, long delta, int whence);
```

Description

`ew_pfseek()` seeks to the specified offset in the file with parallel file descr tor `pfd`. `delta` specifies an offset. If `whence` is `SEEK_SET`, the offset is a plied from the start of the file, if it is `SEEK_END`, the offset is applied from tl end of the file and if it is `SEEK_CUR`, the offset is applied from the current fi position.

`ew_pfseek()` is an implicit barrier on all processes in the group that opene the file. All group members must supply identical parameters otherwise beha iour is undefined.

`ew_pfseek()` returns the new file position on success. If it fails, it returns and sets `errno` to indicate the error.

See Also

`EW_PFD`, `ew_pfread()`, `ew_pfwrite()`

ew_pfread

ew_pfread, ew_pfbread — parallel file read

Synopsis

```
#include <ew/ew.h>
int ew_pfread (EW_PFD *pfd, caddr_t obj, EW_DST *dst);
int ew_pfbread (EW_PFD *pfd, caddr_t b, long nob);
```

Description

`ew_pfread()` reads the global object referenced by `obj`, with distribution `dst`, from the file with parallel file descriptor `pfd`. This call is an implicit barrier on all members of the group that opened the file. All group members must supply consistent parallel file and distribution descriptors, and identical `obj` pointers.

`ew_pfbread()` broadcasts `nob` bytes from the file with parallel file descriptor `pfd` to all group members. This call is an implicit barrier. All members of the group must supply a consistent parallel file descriptor and the same value of `nob`. However `b` does not have to be at the same address in all group members.

On success, `ew_pfread()` and `ew_pfbread()` return the number of bytes read. Note that for `ew_pfread()`, this is the number of bytes in the whole global object, not just the number of bytes in the local slice.

On failure, `ew_pfread()` and `ew_pfbread()` return `(-1)` and set `errno` to indicate the error.

See Also

`EW_PFD, EW_DST.`

ew_pfwrite — **ew_pfwrite, ew_pfbwrite — parallel file write**

Synopsis

```
#include <ew/ew.h>
int ew_pfwrite(EW_PFD *pfd, caddr_t obj, EW_DST *dst
int ew_pfbwrite (EW_PFD *pfd, caddr_t b, long nob);
```

Description

`ew_pfwrite()` writes the global object referenced by `obj`, with distributi `dst`, to the file with parallel file descriptor `pfd`. This call is an implicit barri on all members of the group that opened the file. All group members must supp consistent parallel file and distribution descriptors, and identical `obj` pointe

`ew_pfbread()` writes `nob` bytes from buffer `b` in one group member, to t file with parallel file descriptor `pfd`. This call is an implicit barrier. All memb of the group must supply a consistent parallel file descriptor and the same val of `nob`. `b` does not have to be at the same address in all group members, but contents must be the same because the actual source of the data is not define

On success, `ew_pfwrite()` and `ew_pfbwrite()` return the number of bytes written. Note that for `ew_pfwrite()`, this is the number of bytes in t whole global object, not just the number of bytes in the local slice.

On failure, `ew_pfwrite()` and `ew_pfbwrite()` return `(-1)` and sets `errno` to indicate the error.

See Also

`EW_PFD, EW_DST`

EW_CHAN

EW_CHAN, ew_chanSize, ew_chanInit — channel communication

Synopsis

```
#include <ew/ew.h>
typedef void EW_CHAN;
int ew_chanSize (void);
void ew_chanInit (EW_CHAN *chan,
                  u_int peerProc, EW_CHAN *peerChan,
                  int waitType, int dmaType);
```

Description

An `EW_CHAN` is a one-to-one message passing port. It provides unbuffered communications i.e. a transmit does not complete until a corresponding receive has been posted and the data has been transferred. Full-duplex, non-blocking communication is supported. Up to one transmit and one receive may be posted on a channel at any time.

`ew_chanSize()` returns the size of a channel in bytes.

`ew_chanInit()` initialises channel `chan` and connects it to its peer `peerChan` in the address space of the process with virtual process number `peerProc`. `chan` must be aligned on an EW_ALIGN boundary, otherwise an exception is generated with code `EW_EALIGN`. `waitType` determines how to block for completion and `dmaType` determines how to transfer data for all messages passed on `c`.

Initialisation occurs locally and requires no communication, however both ends of the channel must be initialised before any communication may be initiated, otherwise behaviour is undefined. Typically this occurs by barrier synchronising processes after channel initialisation but before the first communication.

A channel may be re-initialised to connect to a different peer, however behaviour is undefined if any communications are outstanding at this time.

The following example shows channels being used to do an edge exchange b tween an arbitrary number of neighbouring processes.

```
#define NOB 1024
#define NNBR 6
extern u_int nbrVp[NNBR];          /* my neighbours */
extern u_int nbrNum[NNBR];         /* neighbours' index for me */
char exportBufs[NNBR][NOB];
char importBufs[NNBR][NOB];
int i;
EW_CHAN *chans[NNBR];

/* create chans at globally consistent addresses */
for (i = 0; i < NNBR; i++)
   chans[i] = (EW_CHAN *) ew_allocate (ew_base.alloc, EW_ALIGN,
                                       ew_chanSize ());

for (i = 0; i < NNBR; i++)
   ew_chanInit (chans[i], nbrVp[i], chans[nbrNum[i]],
                ew_base.waitType, ew_base.dmaType);

ew_fgsync (ew_base.segGroup);    /* block until all initialised *

...

for (i = 0; i < NNBR; i++)         /* start exchange */
{
   ew_chanTxStart (chan[i], exportBufs[i], NOB);
   ew_chanRxStart (chan[i], importBufs[i], NOB);
}

for (i = 0; i < NNBR; i++)       /* block until exchange complete *
{
   ew_chanTxWait (chan[i]);
   ew_chanRxWait (chan[i]);
}
```

See Also

`ew_chanRxStart()`, `ew_chanTxStart()`.

ew_chanRxStart **ew_chanRxStart, ew_chanRxDone, ew_chanRxWait — channel receive**

Synopsis

```
#include <ew/ew.h>
void ew_chanRxStart(EW_CHAN *c, caddr_t buf, int nob);
int ew_chanRxDone (EW_CHAN *c);
void ew_chanRxWait (EW_CHAN *c);
```

Description

`ew_chanRxStart()` initiates a receive on channel `c` into buffer `buf` which is of length `nob` bytes. The receive completes after a transmit on the channel's peer has been initiated and the message has been copied into `buf`. The next receive may be initiated only after a call to `ew_chanRxWait()`, otherwise behaviour is undefined.

The number of bytes actually transferred is determined by the transmitter. If the transmitter sends a message which is longer than `nob` bytes, memory beyond the end of the receiver's buffer will be overwritten. This error is **not** detected in the standard `libew` version. However an application linked with `libew_dbg`, detects this error at the **transmitter** and causes an exception with code `EW_EOVERRUN`.

`ew_chanRxDone()` returns `TRUE` if the receive outstanding on channel `c` has completed and `FALSE` if it has not. Behaviour is undefined if no receive is outstanding.

`ew_chanRxWait()` blocks until the receive outstanding on channel `c` has completed. On return, the caller may initiate the next receive.

See Also

`EW_CHAN`, `ew_chanTxStart()`.

ew_chanTxStart **ew_chanTxStart, ew_chanTxDone, ew_chanTxWait — channel transmit**

Synopsis

```
#include <ew/ew.h>
void ew_chanTxStart(EW_CHAN *c, caddr_t buf, int nob)
int ew_chanTxDone (EW_CHAN *c);
void ew_chanTxWait (EW_CHAN *c);
```

Description

`ew_chanTxStart()` initiates a transmit on channel `c` from buffer `buf` of `nob` bytes. The transmit completes after a receive on the channel's peer has be initiated and the message has been copied to the receiver's buffer. The next tra mit may be intiated only after a call to `ew_chanTxWait()`, otherwise beha iour is undefined.

If the transmitter sends a message which is longer than the receiver's buffer, memory beyond the end of the receiver's buffer will be overwritten. This error **not** detected in the standard `libew` version. However an application linked wi `libew_dbg`, detects this error at the **transmitter** and causes an exception wi code `EW_EOVERRUN`.

`ew_chanTxDone()` return `TRUE` if the transmit outstanding on channel `c` h completed and `FALSE` if it has not. Behaviour is undefined if no transmit is o standing.

`ew_chanTxWait()` blocks until the transmit outstanding on channel `c` has completed. On return, the caller may initiate the next transmit.

See Also

`EW_CHAN`, `ew_chanRxStart()`.

EW_BCHAN

EW_BCHAN, ew_bchanSize, ew_bchanInit — broadcast channel communication

Synopsis

```
#include <ew/ew.h>
typedef void EW_BCHAN;
int ew_bchanSize (void);
void ew_bchanInit (EW_BCHAN *chan, u_int bcastVp,
                   int waitType, int dmaType);
```

Description

An `EW_BCHAN` provides a combined barrier and broadcast. A non-blocking interface is supported which allows a single outstanding broadcast. It is a global object i.e it exists at the same virtual address in all processes participating in it.

In any broadcast, one process flags itself as the sender, and all others must flag themselves as receivers. The sender uses a broadcast network poll to determine when all processes have become ready. It then initiates a broadcast DMA to distribute the data. This use of hardware broadcast requires a broadcast channel to span a contiguous range of processes.

`ew_bchanSize()` returns the size of a broadcast channel in bytes.

`ew_bchanInit()` initialises the broadcast channel `chan` which spans the processes addressed by broadcast virtual process number `bcastVp`. `chan` must be aligned on an `EW_ALIGN` boundary, otherwise an exception is generated with code `EW_EALIGN`. `waitType` determines how to block for completion and `dmaType` determines how to transfer data on all broadcasts on `chan`. The broadcast channel must be initialised in all its processes before it is used by any one of them. This is most conveniently done with a barrier on a parent group.

Warning – The memory occupied by a broadcast channel may not be reused until all participating process have synchronised using some other means.

See Also

`ew_createBcastVp()`, `ew_bchanStart()`.

ew_bchanStart

ew_bchanStart, ew_bchanDone, ew_bchanWait — broadcast channel transmit

Synopsis

```
#include <ew/ew.h>
void ew_bchanStart (EW_BCHAN *c, int tx,
                    caddr_t data, int nob);
int ew_bchanDone (EW_BCHAN *c);
void ew_bchanWait (EW_BCHAN *c);
```

Description

`ew_bchanStart()` initiates a broadcast on broadcast channel `c`. `tx` is TRU on the sending process. It must be `FALSE` on all others. `data` specifies a sour destination buffer of size `nob` bytes. It must be at the same virtual address in a participating processes, including the sender. The next broadcast on `c` may be itiated only after calling `ew_bchanWait()`. Behaviour is undefined if any these rules is broken.

`ew_bchanDone()` returns `TRUE` if the broadcast outstanding on `c` has com pleted and `FALSE` if it has not. Behaviour is undefined if no broadcast is outstanding on `c`.

`ew_bchanWait()` blocks until the broadcast outstanding on `c` has complete On return, the caller may make a further call to `ew_bchanStart()`.

See Also

`EW_BCHAN`.

EW_TPORT **tagged message passing**

Synopsis

```
#include <ew/ew.h>
```

Description

An EW_TPORT is a point-to-point message passing port which supports the following features.

- Tag and sender selection.
- Non-blocking transmit and receive.
- Messages from the same source to the same destination, arrive in the order sent (given equal selection criteria).
- Buffered or unbuffered message passing determined by sender.

Every tport is initialised with a sender id. This is a single word value which is passed in the envelope information of message. It is used to identify and select the message sender. It must be unique within the community of tports with which the sender communicates.

A message is queued for sending on a tport by passing a flag, a tag, the destination virtual process number and tport, and the message buffer. The tag and the sending tport's sender id are inserted in the message's envelope information. The flag determines whether the message may be buffered.

The return value is a handle on the transmit which is passed to test or block for completion. The maximum number of outstanding transmits is limited only by memory availability.

Transmission completes after the message has been copied. If transmission is unbuffered, this must occur directly. i.e. after a matching receive has been queued. Otherwise the message may be buffered so that transmission can complete without blocking for a matching receive. If a buffered transmit is posted before a matching receive has been queued, a buffer is allocated at the receiving tport and the message is copied to it. When a matching receive is subsequently posted, the message is copied again and the buffer is freed. Failure to allocate a buffer causes an exception.

A receive is queued on a tport by passing tag and sender selection parameters flag, and a buffer. The selection parameters consist of a value and mask pair. T mask determines the significant bits in the value.(e.g. a mask of zero means "match all", all bits set means "exact match"). The flag determines whether ju to probe for a selection match, or to actually consume the incoming message.

The return value is a handle on the receive which is passed to test or block fo completion. The maximum number of outstanding receives is limited only by memory availability.

Performance related parameters are set when a tport is initialised. These inclu the minimum buffer size, the number of attention slots, whether to poll or blo for completion, and how to DMA data through the network.

The minimum buffer size sets the smallest size of buffer which will be allocat for incoming buffered transmits which match no receive. Increasing this size duces fragmentation, at the expense of wasted memory if it is larger than the n jority of messages.

The number of attention slots determines the number of elan threads which c duct the non-blocking transmit and receive operations on behalf of the user. Ea attention slot has one sending and one receiving thread. Increasing the number attention slots increases the tport's memory requirements. However it reduce destination conflicts when several processes attempt to transmit to the same d tination tport and it increases transmit concurrency on a sending tport when m sages are queued to many different destinations. This parameter must be identi in all tports within a community.

The following example shows a tport being used to do an edge exchange between an arbitrary number of neighbouring processes.

```
#define NOB 1024
#define NNBR 6
extern u_int nbrVp[NNBR];          /* my neighbours */
ELAN_EVENT *rxd[NNBR];
ELAN_EVENT *txd[NNBR];
char exportBufs[NNBR][NOB];
char importBufs[NNBR][NOB];
int i;
EW_TPORT *p;

/* allocate tport */
p = (EW_TPORT *) ew_allocate (ew_base.alloc, EW_ALIGN,
                              ew_tportSize (ew_base.tport_nattn));
/* initialise tport */
ew_tportInit (p, ew_base.tport_nattn, ew_state.vp,
              ew_base.tport_smallmsg,
              ew_base.waitType, ew_base.dmaType);

ew_fgsync (ew_base.segGroup);    /* block until all initialised */

...

for (i = 0; i < NNBR; i++)         /* start exchange */
{
   rxd[i] = ew_tportRxStart (p, 0, nbrVp[i], -1, 0, 0,
                             importBufs[i], NOB);
   txd[i] = ew_tportTxStart (p, EW_TPORT_TXSYNC, nbrVp[i], p, 0,
                             exportBufs[i], NOB);
}

for (i = 0; i < NNBR; i++)       /* block until exchange complete */
{
   ew_tportRxWait (rxd[i], NULL, NULL, NULL);
   ew_tportTxWait (txd[i], NULL, NULL, NULL);
}
```

See Also

`ew_init()`, `ew_base`, `ew_gsync()`, `ew_tportInit()`, `ew_tport-TxStart()`, `ew_tportRxStart()`.

ew_tportInit **ew_tportSize, ew_tportInit — tagged message port initialisation**

Synopsis

```
#include <ew/ew.h>
int ew_tportSize (int nAttn);
void ew_tportInit (EW_TPORT *p,
                   int nAttn, int id, int minBufSize,
                   int waitType, int dmaType);
```

Description

`ew_tportSize()` returns the number of bytes which should be allocated t represent a tport with `nAttn` attention slots.

`ew_tportInit()` initialises `p` to be a tagged message passing port with se er id `id`. `p` must be aligned on an `EW_ALIGN` boundary, otherwise an excepti with code `EW_EALIGN` is generated. It is given `nAttn` attention slots, wher each attention slot has a sending and a receiving elan thread. `minBufSize` s the minimum size of buffer to allocate to an incoming messages with no mat ing receive. `waitType` determines how message passing on `p` should block completion and `dmaType` controls how data is transferred.

Warning – The sender id of any tport within any group transmitting to t same destination tport must be unique. Also the value of `nAttn` must be t same in any pair of communicating tports.

The following sender ids are reserved.

Symbolic name	Value
`EW_TPORT_NULLID`	`0x80000000`
`EW_TPORT_ATTNID`	`0xc0000000`

See Also

`EW_TPORT, ew_tportTxStart(), ew_tportRxStart().`

ew_tportTxStart

ew_tportTxStart, ew_tportTxDone, ew_tportTxWait — tagged message transmit

Synopsis

```
#include <ew/ew.h>
ELAN_EVENT *ew_tportTxStart (EW_TPORT *srcp, int flag,
                             u_int dest, EW_TPORT *destp,
                             int tag, caddr_t base, int size);
int ew_tportTxDone (ELAN_EVENT *t);
void ew_tportTxWait (ELAN_EVENT *t);
```

Description

`ew_tportTxStart()` initiates a tagged message transmit from tport `srcp` to tport `destp` in the address space of the process with virtual process number `dest`. The message, of `size` bytes, is in the buffer located at `base`. It is tagged with `tag` and `srcp`'s sender id. Return is immediate. Meanwhile the transmit proceeds according to `flag` as follows.

Flag	Description
0	Buffered transmit. The message is buffered at the destination if no matching receive has been posted. Transmit completes when the message has been copied and may be overwritten.
EW_TPORT_TXSYNC	Unbuffered transmit. Transmit completes when the message has been consumed by a matching receive at the destination.

Note that a buffered transmit consumes memory at the destination until a matching receive is posted. It causes a buffer to be allocated from a pool which is grown on demand by mapping more memory. An exception with code EW_ENOMEM, is caused at the destination if this fails.

When a matching receive is found at the destination, the size of the receiver's buffer is compared with the size of the message. It the receiver's buffer is too small, an exception is generated with code EW_EOVERRUN.

`ew_tportTxStart()` allocates a transmit descriptor from a pool of descriptors associated with `srcp`. If all descriptors are in use, it grows the pool by calling `malloc()`. Failure causes an exception with code EW_ENOMEM. It returns the descriptor, cast to an event, as the handle on the transmit. This should be passed to the polling and completion procedures.

`ew_tportTxDone()` returns `TRUE` if the transmit with handle `t` has compl
ed. Otherwise it returns `FALSE`.

`ew_tportTxWait()` blocks until the transmit with handle `t` has complete
It returns the associated transmit descriptor back to its owning tport.

See Also

`EW_TPORT`, `ew_tportInit()`, `ew_tportRxStart()`.

ew_tportRxStart

ew_tportRxPoll, ew_tportRxStart, ew_tportRxDone, ew_tportRxWait, ew_tportBufFree — tagged message receive

Synopsis

```
#include <ew/ew.h>

extern int ew_tportRxPoll (EW_TPORT *p,
                           int senderMask, int senderSel,
                           int tagMask, int tagSel,
                           int *sender, int *tag, int *size);

ELAN_EVENT *ew_tportRxStart (EW_TPORT *p, int flag,
                             int senderMask, int senderSel,
                             int tagMask, int tagSel,
                             caddr_t base, int size);

int ew_tportRxDone (ELAN_EVENT *r);

caddr_t ew_tportRxWait (ELAN_EVENT *r,
                        int *sender, int *tag, int *size);

void ew_tportBufFree (caddr_t buf);
```

Description

Tagged message receive supports selection on sender and tag. Each selection parameter is specified by a mask and value pair. The mask determines the significant bits of the value.

```
match = ((msg->tag & tagMask) == (tagSel & tagMask)) &&
        (msg->sender & senderMask) == (senderSel & senderMask));
```

A receive posted with a user buffer which is too small to contain the message it matches, generates an exception with code `EW_EOVERRUN`.

`ew_tportRxPoll()` checks if a message matching the given selection parameters has arrived at tport `p`. If no matching message has arrived, it returns `FALSE`. Otherwise it returns `TRUE` and the message's envelope information is returned in `*sender`, `*tag` and `*size`. The caller may pass a `NULL` pointer for any component of the envelope information that is not of interest.

`ew_tportRxStart()` initiates a receive of a message matching the given lection parameters on tport `p`. It returns immediately. Meanwhile the receive p ceeds according to `flag` as follows.

Flag	Description
0	Receive completes when a matching message has been received into the buffer at `base`. `size` spec ified the length in bytes of the buffer.
`EW_TPORT_RXBUF`	Receive completes when a matching message has been received into a tport buffer. This buffer will be returned by `ew_tportRxWait().base` and `size` are ignored.
`EW_TPORT_RXPROBE`	Probe completes when a matching message could be received without blocking. The matching mes· sage is not consumed, but remains awaiting a "real" receive. If the message was transmitted with `EW_TPORT_TXSYNC`, the transmit remains uncompleted. `base` and `size` are ignored.

`ew_tportRxStart()` allocates a receive descriptor from a pool of descri tors associated with `p`. If all descriptors are in use, it grows the pool by callin `malloc()`. Failure causes an exception with code `EW_ENOMEM`. It returns t descriptor, cast to an event, as the handle on the receive. This should be passe to the polling and completion procedures.

`ew_tportRxDone()` returns `TRUE` if the receive with handle `r` has complete Otherwise it returns `FALSE`.

`ew_tportRxWait()` blocks until the receive with handle `t` has completed. frees the associated receive descriptor and returns envelope information in `*sender`, `*tag`, and `*size`. The caller may pass a `NULL` pointer for any co ponent of the envelope information that is not of interest.

If the receive was posted with `EW_TPORT_RXBUF`, `ew_tportRxWait()` turns a pointer to a buffer in the tport's buffer pool, containing the message. Th buffer must be returned to the tport after its contents have been used.

`ew_tportBufFree()` returns a buffer to its owning tport's buffer pool. `buf` must have been previously returned by `ew_tportRxWait()`, on completion of a receive posted with `EW_TPORT_RXBUF`.

See Also

`EW_TPORT`, `ew_tportInit()`, `ew_tportTxStart()`.

ew_rsysServerInit

ew_rsysServerInit, ew_rsysClientInit — remote system call

Synopsis

```
#include <ew/ew.h>
EW_TPORT *ew_rsysServerInit (int bufSize, int nAttn,
                             int waitType, int dmaType)
void ew_rsysClientInit (u_int server,
                        EW_TPORT *tport, int nAttn,
                        int waitType, int dmaType);
```

Description

`ew_rsysServerInit()` spawns a light weight process to act as a remote sy tem call server. This process receives system call requests by tagged message passing, however it fetches and stores larger arguments and returns the systen call result by network DMA to the requester. System calls are served strictly i requesting sequence.

`bufsize` is the maximum buffer size that it may use to packetise large arguments. `nAttn` is the number of attention slots in the server's tport. `waitTyp` controls how the server process blocks for completion and `dmaType` determin how data will be transferred. It returns a pointer to the tport which should be passed to `ew_rsysClientInit()`.

`ew_rsysServerInit()` generates an exception with code `EW_ENOMEM` if fails to allocate space for its packet buffer, tport and light weight process stac It causes an exception with code `EW_EINIT` if it fails to spawn the server lig weight process.

`ew_rsysClientInit()` starts system call redirection to the process with v tual process number `server`. `tport` is the address of the tport that was creat on the server, and `nAttn` must be the number of attention slots it has, otherwi behaviour is undefined. `waitType` controls how to block for system call co pletion and `dmaType` determines how to send system call requests.

`ew_rsysClientInit()` causes an exception with code `EW_ENOMEM` if it fails to allocate space for its tport.

`ew_rsysClientInit()` initiates interception of the following system calls.

System Call	Description
`read()`	Read is intercepted only on file descriptor `0`, corresponding to `stdin`.
`write()`	Write is intercepted only on file descriptors `1` and `2`, corresponding to `stdout` and `stderr`.

Buffering on the standard input, output and error streams must be understood for the desired effect to occur when the underlying `read()` and `write()` system calls are redirected. Also processes must synchronise on reading the standard input if results are to be deterministic.

See Also

`ew_init()`, `ew_base`, `EW_DMAPOOL`, `ew_tportInit()`.

ew_ptrace

ew_ptraceInit, ew_ptraceStart, ew_ptraceStop, ew_ptraceFlush, ew_ptra
— generate ParaGraph trace files

Synopsis

```
#include <ew/ew.h>
void ew_ptraceInit (EW_GROUP *g, char *fname,
                    int ne, int pid);
extern void ew_ptraceFini (void);
void ew_ptraceStart (void);
void ew_ptraceStop (void);
void ew_ptraceFlush (void);
void ew_ptrace (EW_PTR rtype, EW_PTE etype, int n, ...)
```

Description

This set of procedures enables a set of processes to generate ParaGraph form trace information. The information is recorded in a buffer of user determined size, which is periodically flushed to a trace file. Every process generates a se arate trace file. The set of traces may be merged by the following `sort` command.

```
user@cs2-0: sort -m +2n -o fname fname.*
```

`ew_ptraceInit()` initiates tracing for all members of group `g`. `pid` ident fies the caller for tracing purposes. It is used to tag all trace records produced t the calling process. It is also used by other processes when they produce a tra record which identifies this process as the source or destination of a message.

`ew_ptraceInit()` executes a barrier on `g` to synchronise process clocks a ensure consistent time stamps. The group is not used after this initial synchro sation. Each group member creates a private trace file with a name constructe from `fname` and `pid`.

```
sprintf (traceFname, "%s.%d", fname, pid);
```

`ne` sets the size of the event buffer. This buffer is automatically flushed to the fi system when it is full.

`ew_ptraceInit()` generates an exception with `EW_EINIT` if it is called b fore `libew` has been initialised, it can't open the trace file, or if it can't alloca the trace buffer.

`ew_ptraceStart()` enables tracing and records a "start of tracing" event.

`ew_ptraceFlush()` flushes the event buffer to the file system. It records a "start of flushing" event when it begins, and an "end of flushing" event on completion. It generates an exception with code `EW_EIO` if it fails to write to the trace file.

`ew_ptraceStop()` disables tracing, records an "end of tracing" event and calls `ew_ptraceFlush()`. Note that `ew_ptraceStop()` and `ew_ptraceStart()` may be called repeatedly to record snapshots of a program's behaviour

`ew_ptraceFini()` calls `ew_ptraceStop()` to disable tracing and flush any buffered events. It then closes the trace file and frees the event buffer. Note that `ew_ptraceInit()` calls `ew_ptraceFini()` if it is called while an existing trace file is open

`ew_ptrace()` does nothing when tracing is not enabled. Otherwise it records an event by adding to the event buffer. If the buffer becomes full, it calls `ew_ptraceFlush()`. `rtype` is the ParaGraph record type, `etype` is the ParaGraph event type and `n` is the number of data items to follow. If `n` is zero, no further parameters are parsed, otherwise the remaining parameters must be a ParaGraph data type followed by `n` values of that type. An exception with code `EW_ERANGE` is generated if `n` is greater than `8`, or if the data type is not recognised.

The following examples illustrate the use of the tracing procedures in a highe level programming model.

```
int             csend (int mtype, void *mbuf,
                       int len, int node, int pid)
{
   int             rc;

   ew_ptrace (EW_PTR_EVENTENTRY, EW_PTE_SEND, 4, EW_PTD_INT,
              len, mtype, node, pid);
   ...
   ew_ptrace (EW_PTR_EVENTEXIT, EW_PTE_SEND, 0);

   return (rc);
}
```

```
int             crecv (int mtype, void *mbuf, int len)
{
   int             rc;

  ew_ptrace (EW_PTR_EVENTENTRY, EW_PTE_RECVBLOCK, 3, EW_PTD_INT,
             mtype, 0, 0);
   ...
   ew_ptrace (EW_PTR_EVENTEXIT, EW_PTE_RECVBLOCK, 4, EW_PTD_INT,
             mpsc_state.infocount, mpsc_state.infotype,
             mpsc_state.infonode, 0);

   return (rc);
}
```

Error Messages

Message Format

Errors within Widget library programs are reported by the Widget library exce
tion handler. This writes diagnostic messages to the standard error device and
kills the application.

The format of the diagnostic messages is as follows:

```
EW_EXCEPTION @ process: error_code (error_text)
error message string
```

The *error message strings* are listed later in this chapter. The *process* is the vi
tual process number of the process that detected the error; if the exception occu
before the process has attached to the network then this is shown as ----. Tl
error code (and its textual equivalent the *error text*) are one of:

Error Code	Error Text
0	OK
1	Debug
2	Internal error
3	Alignment error
4	Message overrun

Error Code	Error Text
5	Memory exhausted
6	Initialisation error
7	Tx and Rx size discrepancy
8	Value out of range
9	Communication not started
10	Timeout already set
11	I/O error
12	No more route tables
13	Widget busy

The precise meaning of each error is listed later, but in general the errors have the following meanings:

Error Code	Typical meaning
0	Program aborted without error.
1	Program aborted for debugging.
2	Implementation error; please report error to Meiko.
3	Structure not aligned on EW_ALIGN boundary.
4	Incoming message larger than receiving buffer.
5	Insufficient memory available; call to malloc(3c) etc. failed.
6	Did you call the appropriate Init() function?
7	Size of send and receive buffers do not match.
8	Value outside permitted range.
9	Trying to test a communication that had not been started.
10	ew_timeout() has been used twice concurrently.
11	Could not open or write to file.
12	Could not create broadcast routes; insufficient space.
13	Trying to remove data structure that is still in use.

Thread Process Exceptions

Some Widget library functions spawn processes on the Elan Thread Processo
Exceptions in thread process code are similar to those described above:

```
EW_T_EXCEPTION @ process: error_code (error_text)
error message string
```

Other Widget Exception Messages

Message passing libraries implemented above the Widget library (e.g. the CSI
PVM etc. libraries) may also report errors using the Widget library exception
handler. These error messages are described in the documentation for each li-
brary.

Internal Errors

Messages of type 2 (internal error) indicate errors in the implementation of th
library. Please report these errors to Meiko, giving as much information about t
circumstances that caused the error. This should include details of the hardwa
configuration that you were using, and (ideally) minimised code fragments th
will allow Meiko to reproduce the error.

Error Messages

In the following list italicised text represents context specific text or values.

'*ew_version*' incompatible with '*elan_versA*' ('*elan_versB*' expected)
Error type is 6 (initialisation error).

Occurs in the Widget library function `ew_init()`; Elan library version in-
compatibility. An Elan library with version *elan_versA* was found when *ela*
versB was expected.

Attempt to map *size* bytes in allocator type *type*
Error type 2 (Internal error). Please report to Meiko.

Occurs in the Widget library function `ew_allocate()`; searching an internal list of unmapped memory regions and found a reference to memory of type ALLOCED or BSS (when MAPPED was expected).

Bad BCHAN struct spec
Error type 2 (Internal error). Please report to Meiko.

Occurs in the Widget library function `ew_bchanInit()`; the EW_BCHAN structure has not been correctly defined.

Bad CHAN struct spec
Error type 2 (Internal error). Please report to Meiko.

Occurs in the Widget library function `ew_chanInit()`; the EW_CHAN structure has not been correctly defined.

Bad DMA struct spec
Error type 2 (Internal Error). Please report to Meiko.

Occurs in the Widget library function `ew_dmaPoolCreate()`; the EW_DMAPOOL structure has not been correctly defined.

Bad free Rx state
Error type is 2 (Internal error). Please report to Meiko.

Occurs in the Widget library function `ew_tportRxStart()`; internal data structures incorrectly initialised.

Bad GEX struct spec
Error type is 2 (Internal error). Please report to Meiko.

Occurs in Widget library function `ew_gexInit()`; the EW_GEX structure has not been correctly defined.

Bad group member *self* in group size *size*
Error type is 8 (value out of range).

Occurs in the Widget library function `ew_groupInit()`; the function was called with the `self` argument set to less than 0 or greater than the number of group members.

Bad GROUP struct spec
Error type is 2 (Internal error). Please report to Meiko.

Occurs in the Widget library function `ew_groupInit()`; the EW_GROU structure has been incorrectly defined.

Bad size *size* in ew_growBitmap
Error type is 2 (Internal error). Please report to Meiko.

Occurs in the Widget library function `ew_growBitmap()`; this is an intern function that is used by `ew_createBcastVp()`. The specified size was n a multiple of 8 integers.

Bad tport ATTN struct spec
Error type is 2 (Internal error). Please report to Meiko.

Occurs in the Widget library function `ew_tportInit()`; incorrect definitic of an internal data structure.

Bad tport BUF struct spec
Error type is 2 (Internal error). Please report to Meiko.

Occurs in the Widget library function `ew_tportInit()`; incorrect definitic of an internal data structure.

Bad tport HDR struct spec
Error type is 2 (Internal error). Please report to Meiko.

Occurs in the Widget library function `ew_tportInit()`; incorrect definitic of an internal data structure.

Bad tport RXD struct spec
Error type is 2 (Internal error). Please report to Meiko.

Occurs in the Widget library function `ew_tportInit()`; incorrect definitic of an internal data structure.

Bad tport SENDER struct spec
Error type is 2 (Internal error). Please report to Meiko.

Occurs in the Widget library function `ew_tportInit()`; incorrect definitic of an internal data structure.

Bad TPORT struct spec
Error type is 2 (Internal error). Please report to Meiko.

Occurs in the Widget library function ew_tportInit(); incorrect definition of an internal data structure.

Bad tport TXD struct spec
Error type is 2 (Internal error). Please report to Meiko.

Occurs in the Widget library function ew_tportInit(); incorrect definition of an internal data structure.

Can't allocate trace
Error type is 5 (memory exhausted).

Occurs in the Widget library internal function ew_traceAlloc(), which is used by ew_tportInit(), ew_tportTxStart(), ew_tportRx-Start(), and ew_rsysServerInit(). A call to calloc(3c) failed due to insufficient memory.

Can't allocate trace lock
Error type is 5 (memory exhausted).

Occurs in the Widget library internal function ew_traceAlloc(), which is used by ew_tportInit(), ew_tportTxStart(), ew_tportRx-Start(), and ew_rsysServerInit(). A call to calloc(3c) failed due to insufficient memory.

Can't check stack *message*
Error type is 5 (memory exhausted).

Occurs in the Widget library functions ew_groupInit(), ew_tportIn-it(), ew_bchanInit(), ew_dmaPoolCreate(), and ew_touchBuf() (the message identifies which). A call to malloc(3c) failed to allocate an internal data structure.

Can't find own elan capability
Error type is 6 (initialisation error).

Occurs in the Widget library function ew_attach(); could not extract a capability from the environment. The capability should be passed to the application by the resource management system.

This error occurs if you run a parallel program without using the RMS; did you try to execute a parallel application without using prun(1)?

Can't open /dev/zero: *errno* (*message*)
Error type is 6 (initialisation error).

Occurs in the Widget library function `ew_init()`; cannot open `/dev/zer` a call to `open`(2) failed and set `errno` as indicated in the exception messa

channel tx *chan* (**tx** *size* **rx** *size*)
Error type 4 (message overrun).

Occurs in the Widget library function `ew_chanTxDone()` (with debug checking enabled); the transmission has overwritten the recipients data buff (the transmission was too large). *chan* is a pointer to the EW_CHAN structu *tx size* is the amount of data sent, and *rx size* is the amount expected (both bytes).

dma descriptor: dma pool *dmapool*
Error type is 5 (memory exhausted).

Occurs in the Widget library functions `ew_storeStart()` and `ew_fetchStart()`; could not allocate memory for additional DMA descripto (a call to `memalign`(3c) failed). *dmapool* is a pointer to the EW_DMAPOC structure.

DMA has 0 type
Error type is 2 (Internal error). Please report to Meiko.

Occurs in the Widget library function `ew_bcast()`; DMA type has been s to 0.

ew_baseInit() called AFTER ew_attach()
Error type 6 (initialisation error).

Occurs in Widget library function `ew_baseInit()`; the `ew_state.at-tached` field is already initialised indicating that `ew_attach()` has alrea been called.

ew_bchanDone (*bchan*)
Error type is 9 (communication not started).

Occurs in the Widget library function `ew_bchanDone()` (with debuggin enabled); `ew_bchanDone()` was called when there was no outstanding broadcast on that channel. *bchan* is a pointer to the EW_BCHAN structure

ew_bchanInit (*bchan*)
Error type 3 (alignment error).

Occurred in the Widget library function `ew_bchanInit()`; the EW_BCHAN structure that was passed as an argument was not aligned on an EW_ALIGN boundary. *bchan* is a pointer to the structure.

ew_bchanWait (*bchan*)
Error type is 9 (communication not started).

Occurs in the Widget library function `ew_bchanWait()` (with debugging enabled); `ew_bchanWait()` was called when there was no outstanding broadcast on that channel. *bchan* is a pointer to the EW_BCHAN structure.

ew_chanInit (*chan*)
Error type 3 (alignment error).

Occurred in the Widget library function `ew_chanInit()`; the EW_CHAN structure that was passed as an argument was not aligned on an EW_ALIGN boundary. *chan* is a pointer to the structure.

ew_checkVersion(self)
Error type is 2 (Internal error). Please report to Meiko.

Occurs in the Widget library function `ew_init()`; internal incompatibility of library source files.

ew_createBcastVp(): alloc failed after growing tables
Error type is 2 (Internal error). Please report to Meiko.

Occurs in the Widget library function `ew_createBcastVp()`; an internal data structure should be large enough to meet requirements but was found to be insufficient.

ew_createBcastVp (*base, count*) **base out of range**
Error type is 8 (value out of range).

Occurs in the Widget library function `ew_createBcastVp()`; the specified *base* does not identify a process in any of this application's segments.

ew_createBcastVp (*base, count*) **count out of range in seg** *segment* **base** *segbase* **size** *segcount*
Error type is 8 (value out of range).

Occurs in the Widget library function `ew_createBcastVp()`; the specifi range of processes (*base*, *count*) did not fit within a segment. The error me sage identifies the segment, the base process id within the segment (*segbas* and the number of processes in that segment (*segcount*).

ew_createBcastVp (*base*, *count*) **invalid count**
Error type is 8 (value out of range).

Occurs in the Widget library function `ew_createBcastVp()`; the count gument was less than or equal to 0.

ew_dmaPoolDestroy (*dmapool*)
Error type 13 (widget busy).

Occurs in the Widget library function `ew_dmaPoolDestroy()`; attempt destroy an EW_DMAPOOL while DMAs are still active. *dmapool* is a poin to the EW_DMAPOOL structure.

ew_gexInit (*gex*)
Error type is 3 (alignment error).

Occurs in the Widget library function `ew_gexInit()`; the EW_GEX stru ture that was passed as an argument is not aligned on an EW_ALIGN bou ary. *gex* is a pointer to the EW_GEX structure.

ew_gexInit() element *index*, **tx size** = *size*, **rx size** = *size*
Error type is 7 (Tx and Rx size discrepancy).

Occurs in the Widget library function `ew_gexInit()`; a mismatch in size curred between send and receive buffers. This exception occurs if group me ber *i*'s `txIov[j].iov_len` is not equal to member *j*'s `rxIov[i].io len`.

ew_groupInit (*group*)
Error type is 3 (alignment error).

Occurs in the Widget library function `ew_groupInit()`; the EW_GROU structure that was passed as an argument is not aligned on an EW_ALIGN boundary. *group* is a pointer to the EW_GROUP structure.

ew_groupMember (*group*, *member*)**: group size** *size*
Error type is 8 (value out of range).

Occurs in the Widget library functions `lookupId()` and `lookupTable()`, which are installed as the group member lookup functions by specifying any one of `ew_groupFn_id()`, `ew_groupFn_table()`, `ew_groupFn_seg()`, or `ew_groupFn_all()` to the group initialisation function `ew_groupInit()`.

This error message may be seen when `ew_groupMember()` is called (because it is implemented as a call to these group lookup functions). It occurs because the specified group member is less than 0 or greater than the number of group members.

group is a pointer to the EW_GROUP structure, *member* is the group member, and `size` is the group's size (number of members).

ew_pfopen(): parallel file interface not initialised
Error type is 6 (initialisation error).

Occurs in `ew_pfopen()`; `ew_pfInit()` has not been previously used to initialise the parallel file interface.

ew_prefix() not yet implemented
Error type is 2 (Internal error).

Occurs in the Widget library function `ew_prefix()`; this function is not implement in your library.

ew_ptrace() too many data items *count*
Error type is 8 (value out of range).

Occurs in the Widget library function `ew_ptrace()`; number of data values passed to `ew_ptrace()` exceeds 8.

ew_ptrace() bad data type *type*
ew_ptraceFlush() bad data type *type*
Error type 8 (value out of range).

Occurs in the Widget library functions `ew_ptraceFlush()` and `ew_ptrace()`; the ParaGraph data type was not recognised; expecting one of EW_PTD_CHAR, EW_PTD_STRING, EW_PTD_INT, EW_PTD_LONG, EW_PTD_FLOAT, or EW_PTD_DOUBLE.

ew_ptraceFlush(*filename***) failed to output trace data:** *message*
Error type is 11 (I/O error).

Occurs in the Widget library function `ew_ptraceFlush()`; could not wri to tracefile, an internal call to `fprintf(3S)` failed. Check file permission and ensure sufficient space on filesystem.

ew_ptraceInit(*filename*, *ne*) called before ew_attach()
Error type is 6 (initialisation error).

Occurs in the Widget library function `ew_ptraceInit()`; `ew_ptraceInit()` was called before `ew_attach()`.

filename and *ne* are the filename and number of events arguments that wer passed to `ew_ptraceInit()`.

ew_ptraceInit(*filename*, *ne*) can't allocate trace buffer: *message*
Error type is 6 (initialisation error).

Occurs in the Widget library function `ew_ptraceInit()`; cannot allocat memory for internal trace buffer; a call to `malloc(3c)` failed with errno s as described in the exception message.

filename and *ne* are the filename and number of events arguments that wer passed to `ew_ptraceInit()`.

ew_ptraceInit(*filename*, *ne*) can't open trace file: *message*
Error type is 6 (initialisation error).

Occurs in the Widget library function `ew_ptraceInit()`; the specified tracefile could not be opened. Check that you have write permission to the fi system. The `open(2)` system call failed with errno set as described by *mes sage*.

filename and *ne* are the filename and number of events arguments that wer passed to `ew_ptraceInit()`.

ew_ptraceInit(*filename*, *ne*) filename too long
Error type is 6 (initialisation error).

Occurs in the Widget library function `ew_ptraceInit()`; the filename a gument exceeded the internal length limit of (currently) 64 characters.

filename and *ne* are the filename and number of events arguments that wer passed to `ew_ptraceInit()`.

ew_reduce() elsize *size* **pktsize** *size*
Error type is 8 (value out of range).

Occurs in the Widget library function `ew_reduce()`; the specified element size (elsize) is larger than the group's packet size (pktsize). (The group packet size is defined with `ew_groupInit()`.)

ew_rsysInit(): allocating client Tport (*size*)
Error type is 5 (memory exhausted).

Occurs in the Widget library functions `ew_rsysClientInit()` and `ew_rsysServerInit()`; a call to `memalign`(3c) failed while trying to allocate memory of *size* = `ew_tportSize(nAttn)`.

ew_rsysInit(): allocating server buffer *size*
Error type is 5 (memory exhausted).

Occurs in the Widget library function `ew_rsysServerInit()`; a call to `ew_rsysServerInit()` with the specified buffer size argument failed because `malloc`(3c) was unable to allocate the buffer.

ew_rsysInit(): allocating stack *size*
Error type is 5 (memory exhausted).

Occurs in the Widget library function `ew_rsysServerInit()`; a call to `memalign`(3c) failed to allocate stack for the lightweight server process.

ew_rsysInit(): Can't create system call server
Error type is 6 (initialisation error).

Occurs in the Widget library function `ew_rsysServerInit()`; failed to spawn the lightweight server process. Maybe the system limit for LWPs has been exceeded for this user.

ew_t_touchBuf (*bottom–top*)**: continued after waitevent**
Error type is 2 (internal error). Please report to Meiko.

Occurs in the Widget library internal function `ew_t_touchbuf()`, which is used by `ew_touchbuf()`.

bottom and *top* refer to first and last memory pages; the function tries to make memory accesses in each page to generate page faults, and thus preload the pages in advance of their use.

ew_tportInit (*tport*)
Error type is 3 (alignment error).

Occurs in the Widget library function `ew_tportInit()`; the EW_TPOR structure that was specified as an argument was not aligned on an EW_ALIC boundary. *tport* is a pointer to the EW_TPORT structure.

ew_tportTxStart: t_done set
Error type is 2 (Internal error). Please report to Meiko.

Occurs in the Widget library function `ew_tportTxStart()`; an internal event was ready sooner than expected.

ew_utimeout (*timeout, handler*)
Error type is 10 (timeout already set).

Occurs in the Widget library function `ew_utimeout()`; the function has been called while a previous call is still pending. Only one procedure may scheduled at any one time. *timeout* and *handler* are the arguments that wer passed to the failed function call.

Failed to allocate rsys save descriptors
Error type is 5 (memory exhausted).

Occurs in the Widget library internal function `ew_rsysSaveFds()`; a call `malloc`(3c) failed.

Failed base createAllocator (*base, size*) *errno: message*
Error type 6 (initialisation error).

Occurs in Widget library function `ew_baseInit()`; a call to `ew_createAllocator()` failed; the `ew_base.alloc` field could not be initia ised because a request for memory failed. Errno was set by the allocating functions as reported by this error message.

Failed base group allocation
Error type is 6 (initialisation error).

Occurs in Widget library function `ew_baseInit()`; a call to `ew_allocate()` failed; could not allocate the EW_GROUP structure to initialise th `.ew_base.allGroup` field. Probable cause is insufficient memory.

Failed elan_addvp (*segment* @ *process* **for** *count*) *errno*: *message*
Error type is 6 (initialisation error).

Occurs in the Widget library function `ew_attach()`; a call to `elan_addvp()` failed and set errno to the specified value. `elan_addvp()` is used to define the virtual process number for the members of all the segments in the application.

segment is the segment id, *process* is the process within the segment that failed to call `elan_addvp()`. *count* is the number of processes in the segment.

Failed elan_attach() *errno*: *message*
Error type is 6 (initialisation error).

Occurs in the Widget library function `ew_attach()`; a call to `elan_attach()` failed and errno was set to the value reported by this exception.

`elan_attach()` may fail because the process has already called `elan_attach()`, or because the capability has been corrupted (maybe the LIB_EWCAP environment variables have been corrupted).

Failed elan_init() *errno*: *message*
Error type is 6 (initialisation error).

Occurs in the Widget library function `ew_init()`; a call to `elan_init()` failed. This may occur because your machine is equipped with the wrong revision Elan device, there are too many processes currently using the Elan, there is no virtual address space left to map-in the Elan device, or you are running the program on a processor with no attached Elan device.

When `elan_init()` failed it set errno to the value reported in the exception message.

Failed segment group allocation
Error type 6 (initialisation error).

Occurs in the Widget library function `ew_baseInit()`; a call to `ew_allocate()` failed; could not allocate the EW_GROUP structure to initialise the `ew_base.segGroup` field. Probable cause is insufficient memory.

Failed to allocate dma pool
Error type 5 (memory exhausted).

Occurs in the Widget library function `ew_dmaPoolCreate()`; could not locate memory for the EW_DMAPOOL structure (a call to `memalign(3c` failed).

Failed to allocate grow route table *process+count*
Error type is 12 (no more route tables).

Occurs in the Widget library function `ew_createBcastVp()`; a call to `elan_addrt()` failed possibly because there was insufficient space in the Elan route tables to create a broadcast virtual process id for this group of processes.

process is the first process in the broadcast group, *count* is the number of processes.

Failed to create gex dmapool
Error type is 6 (initialisation error).

Occurs in the Widget library function `ew_gexInit()`; failed call to `ew_dmaPoolCreate()`; could not create EW_DMAPOOL structure. Possible memory shortage.

Failed to grow broadcast vp bitmap *process+count*
Error type is 5 (memory exhausted).

Occurs in the Widget library function `ew_createBcastVp()`; failed to grow internal structures to accommodate processes *number* to (*number+count*). *count* is the required number of routes (a multiple of 8).

Failed to parse env *name=value*
Error type is 6 (initialisation error).

Occurs in the Widget library function `ew_parseEnvVar()`; this is an internal function that is used by `ew_baseInit()`, `ew_init()`, `ew_attach(` and `ew_rsysServerInit()` to parse environment variables. This may indicate that the wrong type of value was assigned to a variable (an integer where a name was expected). *name* is the name of the variable, *value* is its current incorrect value.

Failed to set broadcast route *base* **for** *count*
Error type is 2 (Internal error). Please report to Meiko.

Occurs in the Widget library function `ew_createBcastVp()`; a call to the Elan library function `elan_setrt()` unexpectedly failed.

Failed realloc(capVec = *count*) *errno*: *message*
Error type is 6 (initialisation error).

Occurs in the Widget library function `ew_attach()`; a call to `realloc`(3c) failed while trying to extend the internal buffers to store segment capabilities. The *count* is the number of capabilities that we want to create space for (which will be a multiple of 2 and may be more than the application really needs).

errno is the result of the failed `realloc()`, and the *message* is a textual explanation of the error.

Non-uniform library utilisation: '*versionA***' (***member***) '***versionB***' (self)",**
Error type is 6 (initialisation error).

Occurs in the Widget library function `ew_sgsync()`; group members were compiled with different library versions; the process that detected the error (self) was compiled with *versionB*, whereas group member *member* was compiled with *versionA*.

Self not in route capabilities
Error type is 6 (initialisation error).

Occurs in the Widget library function `ew_attach()`. `ew_attach()` extracts the capability for each segment from environment variables. LIBEW_ECAP is the capability for this segment, and LIBEW_ECAP*n* is the capability for segment n. The exception is generated if there was no value of *n* for which LIBEW_ECAP = LIBEW_ECAP*n*.

tagged message buffer: port *tport* **index** *number* **size** *size* **total** *size*
Error type is 5 (memory exhausted).

Occurs in the Widget library internal function `ew_t_tportNewBuf()`, which is used by `ew_tportRxStart()`; failed to allocate memory for an internal data buffer (for a non-blocking communication).

tagged msg rx: port *port*
Error type is 5 (memory exhausted).

Occurs in the Widget library internal function `newRxDesc()`, which is us by both `ew_tportRxStart()` and `ew_rsysServerInit()`. A call to `memalign`(3c) failed. *port* is a pointer to a EW_TPORT structure

tagged msg rx: tag *rxtag* **(***tag***) sender** *rxsender* **(***sender***) size** *rxsize* **(***size***) po** *tport*

Error type is 4 (message overrun).

Occurs in the Widget library functions `ew_tportRxStart()` and `ew_tportRxWait()`; receive buffer is too small to contain the message that i matches.

The exception message displays the following "received (expected)" pairs

rxtag	Tag on incoming message.
tag	`tagSel` argument as specified to `ew_tportRxStart()`.
rxsender	Sender's id.
sender	`senderSel` argument to `ew_tportRxStart()`.
rxsize	Incoming message size.
size	Size of receiver's buffer; this may be the system default size if either EW_TPORT_RXBUF or EW_TPORT_RXPROBE flags were set.
tport	Pointer to the receiving EW_TPORT (argument to `ew_tportRxStart()`).

tagged msg tx: port *port*

Error type is 5 (memory exhausted).

Occurs in the Widget library internal function `newTxDesc()`, which is us by both `ew_tportTxStart()` and `ew_rsysServerInit()`. A call to `memalign`(3c) failed. *port* is a pointer to the EW_TPORT structure that w passed to the failed function.

version string too long (<version>)

Error type is 2 (Internal error). Please report to Meiko.

Occurs in the Widget library function `ew_sgsync()`; the version string re turned by `ew_version()` does not match the internal 64 byte limit. *versi* is the string that `ew_version()` returned.

Index

F

G

64630