Computing Surface

CS-2 Documentation Set

Volume 2

83–MS048

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

Resource Management User Interface Library

S1002–10M110.01

meiko

Meiko's address in the US is:

Meiko Scientific
Reservoir Place
1601 Trapelo Road
Waltham MA 02154

617 890 7676
Fax: 617 890 5042

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*

Contents

Functions 1

About this Manual

This chapter describes the user interface to the Resource Management System — `librms`. The functions in this library allow user programs to query the resources in the CS-2 and to run parallel programs on those resources. Direct use of this library will allow you to write your own versions of the resource management commands (such as `prun`, `allocate`, and `rinfo`) and to tailor them to the specific requirements of your own applications.

The resource management user interface library also includes a number of system administration functions which are not described in this manual. These functions are used by high level system administration tools, such as Pandora, which offer to the System Administrator a safe environment in which to perform sensitive operations.

Compilation

Function prototypes, data structures, and associated definitions for use with this library are included in the header file `<rmanager/uif.h>` which is distributed in `/opt/MEIKOcs2/include`. You will need to include this header file in your program files and specify it's home directory in your pre-processor's search path (usually with the compiler driver's `-I` option, as shown below).

Applications built upon this library must be linked with `librms` (resource management library), `libew` (Elan Widget library), and `libelan` (Elan library) — all are distributed in `/opt/MEIKOcs2/lib`. You usually identify these libraries and their home directory to the linker by using your compiler driver's `-L` and `-l` options (as shown below).

The resource management user interface library is a dynamic shared library and requires a search path to be passed to the runtime linker; the most convenient way of doing this is to specify a search path using your compiler driver's `-R` option. Failure to specify a search path will result in the following error message whenever you execute your application:

```
ld.so.1: program: fatal: librms.so: can't open file:
errno=2: Killed
```

A typical compiler command line for a resource management program is:

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

rms_allocate() — Allocate resources

Synopsis

```
#include <rmanager/uif.h>
int rms_allocate(rrequest_t *request);
```

Availability

MEIKOcs2 — MKrms

Description

Allocate resources and hold them until the calling program exits or the resource timelimit is exceeded (note that timelimits for resources are specified by the System Administrator in the `defaults`(4) file). You only need to use this function when allocating resource in advance of running your parallel application; normally resource allocation and program execution is handled in one operation by `rms_forkexecvp`().

The required resources are specified by an `rrequest_t` structure, which is usually allocated and initialised by a call to `rms_defaultResourceRequest`() (which reads a resource specification from your environment).

```
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 rid;                   /* resource identifier */
   int flags;                 /* options on request */
   int routeTable;            /* route table to use */
   char partition[NAME_SIZE]; /* partition to use */
} rrequest_t;
```

Unassigned fields in the `rrequest_t` structure (set to `RMS_UNASSIGNED`) are interpreted as 'don't care', with the exception of the partition name which is mandatory. On return from `rms_allocate`() the `rrequest_t.rid` field and the RMS_RESOURCEID environment variable will identify the allocated resource (the value assigned to this variable takes the form *partition.rid*, where *partition* is the name of the partition that the resource is allocated from and *rid* is an integer resource identifier); the environment variable allows the allocation and execution phases to occur in separate processes.

Return values from `rms_allocate`() are a positive integer resource id on success or -1 on failure. `rms_allocate`() will fail if resources are already allocated (either by an earlier explicit call to `rms_allocate`() or by running the program in a shell that has executed the `allocate` command). To run a program on the allocated resource you need to pass the resource id to rms_forkexecvp() by assigning it to the rrequest_t.rid field (note that rms_forkexecvp() will itself call rms_allocate() if this field remains unassigned).

Warning – The accounting system charges for the whole period that the resource is held, whether you use it or not.

Example

The following example uses rms_defaultResourceRequest() to get the resource specification from the environment and then modifies this to suit the specific requirements of this application. If the program is being run by a shell with allocated resource then we must use those resources and must not attempt to allocate resource ourselves; rms_defaultResourceRequest() will return in the rrequest_t.rid field a resource identifier that will identify the shell's resource to rms_forkexecvp().

```
#include <sys/wait.h>
#include <stdio.h>
#include <rmanager/uif.h>

#define NPROCS 2
#define PARTITION "parallel"
#define MYPROGRAM "/opt/MEIKOcs2/example/csn/csn"
#define VERBOSE 1

main(int argc, char** argv)
{
   rrequest_t *rreq;
   int rid, status;
   char buffer[30];

   /* Specify the resources that we require */
   rreq = rms_defaultResourceRequest();
   rreq->nProcs = NPROCS;
   rreq->flags = REQUEST_VERBOSE;
   sprintf(rreq->partition, PARTITION);

   /* Grab the resources, but only if they have not      */
   /* already been allocated to the shell by allocate(1) */
   if(rreq->rid < 0)
      if((rid = rms_allocate(rreq)) < 0) {
         printf("Failed to allocate resources\n");
         exit(-1);
```

```
        } else
           rreq->rid = rid;

   /* We could do some work here whilst holding the resource */

   /* Execute the program on the grabbed resource */
   if(rms_forkexecvp(rreq, MYPROGRAM, argv)) {
      fprintf(stderr, "forkexecvp() failed\n");
      exit(1);
   }

   /* Wait for the parallel application to complete */
   if(rms_waitpid(rms_getgpid(), &status, 0)) exit(1);

   /* return exit status of parallel application */
   return(WEXITSTATUS(status));
}
```

The `rms_allocate()` function is used in the implementation of the `allocate(1)` command to allocate resources to a command shell.

See Also

`rms_forkexecvp()`, `allocate`. See the `rrequest_t` structure on page 75.

rms_boardTypeString() — Printable board type string

Synopsis

```
#include <rmanager/uif.h>
char *rms_boardTypeString(BoardTypes board)
```

Availability

MEIKOcs2 — MKrms

Description

rms_boardTypeString() converts an enumerated BoardType value into a printable string. This function is used to display the type field in the board_t structure.

Return strings are:

Quattro	1x16 switch
Vector	Small switch
Dino	Switch buffer
4x4 switch	Module controller
2x8 switch	unknown (*value*)

Example

Display the type of all boards in the system:

```
#include <rmanager/uif.h>

main()
{
   board_t *board;
   int i = 0;

   while((board=(board_t*)rms_describe(RMS_BOARD, i++))!=NULL)
      printf("Board type is: %s\n",
             rms_boardTypeString(board->type));
}
```

See Also

rms_describe(). See also board_t on page 46.

rms_checkVersion() — Confirm library version

Synopsis

```
#include <rmanager/uif.h>
int rms_checkVersion(char *version);
```

Availability

MEIKOcs2 — MKrms

Description

This function checks the version string against the library version that your application has been linked with; it returns 1 if they are identical and 0 if they are not.

Example

`rms_checkVersion()` is usually passed the library version that is returned by `rms_version()` allowing you to confirm that your application is both compiled with and linked with the same library version.

```
#include <rmanager/uif.h>

main(int argc, char** argv)
{
   if (!rms_checkVersion(RMS_VERSION))
   {
      printf("'%s' incompatible with '%s' ('%s' expected)\n",
         argv[0], rms_version(), RMS_VERSION);
      exit(1);
   }
   else
      printf("Library version's correct\n");
}
```

See Also

`rms_version()`.

rms_confirm() — Confirm service availability

Synopsis

```
#include <rmanager/uif.h>
int rms_confirm(char *server);
```

Availability

MEIKOcs2 — MKrms

Description

`rms_confirm()` tests the availability of resource management services. It can be used to test the availability of the following:

Service	Description
`acctd`	Accounting daemon.
`mmanager`	The machine manager
`active/`*partition*	The partition manager for *partition*.

`rms_confirm()` returns 0 if the service is available and -1 if not.

Example

Confirm availability of all services:

```
#include <rmanager/uif.h>

main()
{
   partition_t *partn;
   int i;
   char name[NAME_SIZE];


   /* See if machine manager is there */
   if (rms_confirm("mmanager") == 0)
      printf("Machine manager is available.\n");
   else {
      printf("Machine manager not available.\n");
      exit (0);
   }


   /* See if accoutning daemon is there */
   if (rms_confirm("acctd") == 0)
      printf("Resource accounting daemon is available.\n");
   else
      printf("Resource accounting daemon not available.\n");
```

```
    /* Check all partitions in current (active) configuration */
    i = 0;
    while(partn=(partition_t*)rms_describe(RMS_PARTITION, i++))
    {
       sprintf(name, "active/%s", partn->name);

       printf("Partition %s ", partn->name);

       if (rms_confirm(name) == 0)
          printf("is available.\n");
       else
          printf("is not available.\n");
    }
}
```

rms_defaultResourceRequest() Get default resource specification from the environment

Synopsis

```
#include <rmanager/uif.h>
rrequest_t *rms_defaultResourceRequest();
```

Availability

MEIKOcs2 — MKrms

Description

rms_defaultResourceRequest() fetches default resource requirements from your environment and creates a rrequest_t structure that is initialised with these defaults.

If resource has already been allocated, possibly to the user's command shell, then the rrequest_t structure is initialised with information about that resource (identified by the RMS_RESOURCEID environment variable). The function then reads the following resource management environment variables; if an environment variable conflicts with the definition of an allocated resource it is ignored (e.g. you cannot ask for more processors than have already been allocated), otherwise it overrides.

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. Default is set in the defaults(4) file (-1 means no limit).
RMS_VERBOSE	Execute in verbose mode (display diagnostic messages).
RMS_DEBUG	Execute under the control of a debugger.
RMS_PARTITION	The name of the partition to use. Default is set in the defaults(4) file.

Variable	Meaning
RMS_IMMEDIATE	Exit if resources not immediately available. By default the calling process is blocked until resources are available.
RMS_BARRIER	Execute this program as a parallel application (processes will barrier synchronise with host). By default the resource management system makes its own evaluation.
RMS_ROUTETABLE	Identifies the name of the route table to use (for example "scatter", "random", or "user_default"). See also the rmsroutes(1m) manual page.

Example

Fetch the default resource requirements from the environment and then change as appropriate to this application — use 2 processors starting with processor 2 in the `parallel` partition. Execute with the verbose and timing flags set.

```
#include <sys/wait.h>
#include <stdio.h>
#include <rmanager/uif.h>

#define NPROCS 2
#define PARTITION "parallel"
#define EXAMPLE "/opt/MEIKOcs2/example/csn/csn"

main(int argc, char** argv)
{
   rrequest_t *rreq;
   int status;

   rreq = rms_defaultResourceRequest();

   rreq->nProcs = NPROCS;
   sprintf(rreq->partition, PARTITION);
   rreq->flags = REQUEST_VERBOSE | REQUEST_TIMING;

   /* Start the application */
   if(rms_forkexecvp(rreq, EXAMPLE, argv)) {
      fprintf(stderr, rms_forkexecvp() failed\n");
      exit(1);
   }
```

```
    /* Wait for the application to finish */
    if(rms_waitpid(rms_getgpid(), &status, 0)) exit(1);

    /* Exit with applications return status */
    return(WEXITSTATUS(status));
}
```

See Also

rms_forkexecvp(), allocate. See the rrequest_t structure on page 75.

rms_describe() — Query resource availability

Synopsis

```
#include <rmanager/uif.h>
void *rms_describe(RMS_OBJECT_TYPES type, int Id);
```

Availability

MEIKOcs2 — MKrms

Description

The resource management system supports a query mechanism that allows applications to explore the resources available to them. This interface covers both the hardware and the active configuration. The user interface to this facility is via the function `rms_describe()`.

The `type` argument specifies the type of object as described by the enumerated data type `RMS_OBJECT_TYPES`:

```
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_RESOURCE = 8,        /* application target */
   RMS_JOB = 9,             /* parallel program */
   RMS_PROCBYELANID = 10,   /* processing element from elanId */
   RMS_LINK = 11,           /* link between switches */
   RMS_RESOURCEBYID = 12,   /* processing resource */
   RMS_FILESYS = 13,        /* all the filesystems */
   RMS_SERVER = 14,         /* a filesystem server */
   RMS_FSYS = 15,           /* a filesystem */
} RMS_OBJECT_TYPES;
```

The `Id` argument is a logical id that is used to select an instance of the object. In general the logical id for each object type begins at 0 and is assigned sequentially to each object; the ordering of objects is undefined (so you should avoid making any assumptions based on an object's id). Exceptions are job's and resource's whose id's are relative to the partition that they are allocated to and do not begin at 0 (use the macro PARTITION_BASE() to get the id for the initial job/resource in a given partition), and objects of type RMS_RESOURCEBYID and RMS_PROCBYELANID which enable access to resource and processor objects by resource id or Elan id respectively.

`rms_describe()` returns a NULL pointer on error.

Hardware Resources

Programs that wish to query the hardware resources in the machine will usually begin with a call to `rms_describe()` with the object type `RMS_MACHINE` (note that the `Id` argument is always 0 because there is only ever one instance of a machine). This returns a `machine_t` structure that describes at the highest level the components in the machine:

```
machine = (machine_t *)rms_describe(RMS_MACHINE, 0);
```

(The `machine_t` structure is described in Chapter 2.)

Additional information about the machine's components may be obtained by subsequent calls to `rms_describe()`. Each call queries a lower level object until the desired level is reached. At each stage the range of appropriate logical id's is extracted from the previous stage. For example, the logical id's of all the modules in the machine are extracted from the machine description.

The following variants of `rms_describe()` are typically used to query the hardware resources in the machine (the data structures returned by these functions are described in Chapter 2):

```
module = (module_t *)rms_describe(RMS_MODULE, moduleId);
 board = (board_t *)rms_describe(RMS_BOARD, boardId);
  proc = (proc_t *)rms_describe(RMS_PROC, procId);
switch = (switch_t *)rms_describe(RMS_SWITCH, switchId);
device = (device_t *)rms_describe(RMS_DEVICE, deviceId);
```

The following example gets a description of the machine, a description of the modules in the machine, and a description of the boards in each module. This example shows a typical hierarchical query of resource objects.

```
#include <stdio.h>
#include <rmanager/uif.h>


main()
{
   int i, j, count, base;
   machine_t *machine;
   module_t *module;
```

```
    board_t *board;

    if((machine=(machine_t*)rms_describe(RMS_MACHINE,0))==NULL){
       fprintf(stderr, "Cannot get machine description\n");
       exit(1);
    }

    /* Get a description of all the modules in the machine */
    for(i=0; i<machine->nModules;i++)  {

       if((module=(module_t*)rms_describe(RMS_MODULE,i))==NULL){
          fprintf(stderr, "Cannot get module description\n");
          exit(1);
       }

       printf("Module type %s\n",
                rms_moduleTypeString(module->type));

       /* Get a description of the boards in the module */

       count = module->nProcs;   /* number of boards */
       base = module->baseProc; /* Logical id of 1st */

       for (j = base; j<count+base-1; j++)  {

          if((board=(board_t*)rms_describe(RMS_BOARD,j))==NULL){
             fprintf(stderr, "Cannot get board description\n");
             exit(1);
          }

          printf("Board type is %s\n",
                   rms_boardTypeString(board->type));
       }
    }
}
```

Configuration Resources

Programs that wish to query the current configuration will usually begin with a call to `rms_describe()` with the object type `RMS_CONFIGURATION` (note that the `Id` argument is always 0 because there is only ever one active configuration). This returns a `config_t` structure that describes at the highest level the make-up of the current configuration:

```
config = (config_t*)rms_describe(RMS_CONFIGURATION, 0);
```

Additional information about the configuration may be obtained by subsequent calls to rms_describe(). Each call queries a lower level object until the desired level is reached. At each stage the range of appropriate logical id's is extracted from the previous stage. For example, the logical id's of all the partitions in the machine are extracted from the configuration description.

The following variants of rms_describe() are typically used to query the configuration (the data structures returned by these functions are described in Chapter 2):

```
   config = (config_t *)rms_describe(RMS_CONFIGURATION, 0);
partition =(partition_t *)rms_describe(RMS_PARTITION, partId);
 resource = (resource_t *)rms_describe(RMS_RESOURCE, targetId);
      job = (job_t *) rms_describe(RMS_JOB, jobId);
```

The following example shows all the jobs in the parallel partition. Note that the initial logical id for the jobs in this partition is relative to the partition's logical id (and not 0 as with most other object types); this allows you to distinguish jobs in different partitions. Note also that we fetch all the job descriptions for this partition by calling rms_describe() until it returns NULL; you can apply the same technique to any object type when you wish to query all instances.

```
#include <stdio.h>
#include <rmanager/uif.h>

#define PARTITION "parallel"

main()
{
   partition_t *p;
   job_t *job;
   int i = 0;

   /* Get logical id of the partition */
   while((p=(partition_t*)rms_describe(RMS_PARTITION, i++))!=NULL)
      if (!strcmp(p->name, PARTITION)) break;
```

```
    /* Exit if we couldn't locate the partition */
    if (p == NULL) {
       printf("Failed to located partition %s\n", PARTITION);
       exit(1);
    }

    /* Job ids start at partition base */
    i = PARTITION_BASE(i-1);

    /* Get all the job descriptions for this partition */
    while( (job = (job_t*)rms_describe(RMS_JOB, i++)) != NULL) {
       printf("Process: %s  Owner: %d  Status: %s\n",
                rms_gpidString(job->gpid), job->uid,
                rms_jobStatusString(job->status));
    }
}
```

See Also

The descriptions of the data structures and their usage are listed in Chapter 2.

rms_elantohost() — Translate Elan Id to hostname

Synopsis

```
#include <rmanager/uif.h>
int rms_elantohost(char *hostname, int elanId);
```

Availability

MEIKOcs2 — MKrms

Description

rms_elantohost() translates the specified Elan Id to the processor's hostname. The result is stored in hostname.

Return values are 0 on success, -1 on failure.

See Also

rms_hosttoelan(), rms_ntoelan(), rms_elanton().

rms_elanton() — Elan Id to Ethernet address translation

Synopsis

```
#include <rmanager/uif.h>
#include <netinet/if_ether.h>
struct ether_addr *rms_elanton(int elanId);
```

Availability

MEIKOcs2 — MKrms

Description

rms_elanton() translates the specified Elan Id to the processor's Ethernet address — the address is standard 48 bit format, but only the last two fields are used. Return values are the address on success or -1 on failure.

See Also

rms_ntoelan(), rms_hosttoelan(), rms_elantohost().

rms_forkexecvp() **Process creation**

Synopsis

```
#include <rmanager/uif.h>
int rms_forkexecvp(rrequest_t *req, char *file,
                              char **args);
```

Availability

MEIKOcs2 — MKrms

Description

rms_forkexecvp() executes a parallel program on a resource.

The resources required by the parallel application are specified by an rrequest_t structure. If the partition field of the rrequest_t structure is unassigned then rms_forkexecvp() uses the default partition named in the defaults(4) file. If the rid field is unassigned rms_forkexecvp() uses rms_allocate() to allocate the requested resources.

```
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 rid;                   /* resource identifier */
   int flags;                 /* options on request */
   int routeTable;            /* route table to use */
   char partition[NAME_SIZE]; /* partition to use */
} rrequest_t;
```

In most cases the rrequest_t structure should be created and initialised by a call to rms_defaultResourceRequest(). This determines if the program is being run on pre-allocated resource (the command shell may have allocated resource) and uses that resource (if any) and your RMS environment variables (if any) to initialise the rrequest_t structure.

Note that the rrequest_t.rid field must be initialised with a resource id if resource has already been allocated; rms_forkexecvp() will fail if the field remains un-initialised under these circumstances. You must therefore use rms_defaultResourceRequest() to check your environment, or if you use rms_allocate() you must explicitly assign its return value.

The file argument is the name of the program to execute; the program must be executable, locatable in the user's search path, and the current working directory must exist on all processors.

argv is the argument array that is passed to the user's program.

The return value from rms_forkexecvp() is 0 on success or -1 on failure. rms_forkexecvp() will return when the processes have been created; use rms_waitpid() to block the calling program until they have completed (and to return the segment's exit status).

Example

Execute the program on 2 processors with the verbose flag set.

```
#include <sys/wait.h>
#include <stdio.h>
#include <rmanager/uif.h>

#define EXAMPLE "/opt/MEIKOcs2/example/csn/csn"

main(int argc, char** argv)
{
   rrequest_t *req;
   int status;

   /* Initialise default request structure */
   req = rms_defaultResourceRequest();

   /* Change the defaults that are inappropriate */
   req->nProcs = 2;
   req->flags |= REQUEST_VERBOSE;

   /* Execute the program using the specified resource */
   if(rms_forkexecvp(req, EXAMPLE, argv)) {
      fprintf(stderr, "rms_forkexecvp() failed\n");
      exit(1);
   }

   /* Wait for the applciation to complete */
   if(rms_waitpid(rms_getgpid(), &status, 0)) exit(1);

   /* Return the applications exit status */
   return(WEXITSTATUS(status));
}
```

See Also

rms_defaultResourceRequest(), rms_allocate().

See also the description of rrequest_t on page 75.

rms_getgpid() — Return global process id

Synopsis

```
#include <rmanager/uif.h>
gpid_t rms_getgpid();
```

Availability

MEIKOcs2 — MKrms

Description

rms_getgpid() returns the global process id of the calling process. A global process id consists of two components: the Elan Id of the processor and the local process id on that processor.

Two macros are provided in <rmanager/uif.h> for extracting the components from a gpid_t type; these are PROCESSOR(gpid) and PROCESS(gpid). A third macro, GPIDS_MATCH(), compares two gpid_t variables for equality.

The function rms_gpidString() will convert a global process id into a printable string.

See Also

rms_gpidString().

rms_getgsid() — Return global session id

Synopsis

```
#include <rmanager/uif.h>
gpid_t rms_getgsid(gdit_t gpid);
```

Availability

MEIKOcs2 — MKrms

Description

`rms_getgsid()` returns the global session id for the process identified by the global process id `gpid`.

See Also

`rms_setgsid()`

rms_gpidString() — Convert global process/segment id to printable string

Synopsis

```
#include <rmanager/uif.h>
char *rms_gpidString(gpid_t gpid);
```

Availability

MEIKOcs2 — MKrms

Description

`rms_gpidString()` converts a `gpid_t` data type into a printable string in the form *processor.process*.

See Also

`rms_getgpid()`.

rms_hosttoelan() — Translate hostname to Elan Id

Synopsis

```
#include <rmanager/uif.h>
int rms_hosttoelan(char *hostname);
```

Availability

MEIKOcs2 — MKrms

Description

rms_hosttoelan() translates the specified hostname to the processor's Elan Id. Return values are the Elan Id on success or -1 on failure.

See Also

rms_elantohost(), rms_ntoelan(), rms_elanton().

rms_jobStatusString() — Printable job status string

Synopsis

```
#include <rmanager/uif.h>
char *rms_jobStatusString(JobStatus status);
```

Availability

MEIKOcs2 — MKrms

Description

`rms_jobStatusString()` converts an enumerated `JobStatus` value into a printable status string. This function is used to display the `status` field in the `job_t` structure (returned by `rms_describe()`). Return strings are:

Return string	Meaning
`zombie`	Job has exited, failed or been killed on one but not all processors (job status is `JOB_RUNNING` & (`JOB_NOTRUN` | `JOB_KILLED` | `JOB_EXITED`)
`running`	Job is running (job status is `JOB_RUNNING`).
`starting`	Job is starting (job status is `JOB_STARTING`).
`killed`	Job was killed (job status is `JOB_KILLED`).
`exited`	Job finished normally (job status is `JOB_EXITED`).
`unknown (`*value*`)`	None of the above.

See Also

See also `job_t` on page 55.

rms_kill() — Deliver a signal to a parallel program

Synopsis

```
#include <rmanager/uif.h>
int rms_kill(gpid_t gpid, int signum);
```

Availability

MEIKOcs2 — MKrms

Description

This function delivers a signal to the specified process. `gpid` is a global process id, and `signum` is the signal that is to be delivered.

`rms_kill()` returns -1 on error and 0 on success.

A list of signal numbers is included in `signal(5)`.

See Also

`rms_getgpid()`, `signal(5)`, `rms_sigsend()`

rms_logbal() — Identify least loaded processor in a partition

Synopsis

```
#include <rmanager/uif.h>
int rms_logbal(uid_t id, char *partition,
               logbal_t *info);
```

Availability

MEIKOcs2 — MKrms

Description

rms_logbal() identifies the least loaded processor in partition. rms_logbal() uses the statistic specified in the system defaults(4) file to determine processor loading (this is specified by the System Administrator).

The id argument is the user's id as returned by getuid(2).

On return from this function the logbal_t structure is initialised with the hostname and IP address of the least heavily loaded processor in the partition.

rms_logbal() returns a value of -1 on error, and 0 on success.

Example

Find the least loaded processor in the parallel partition:

```
#include <stdio.h>
#include <rmanager/uif.h>

#define PARTITION "parallel"

main()
{
   logbal_t lbalinfo;
   uid_t myid;

   myid = getuid();
   if(rms_logbal(myid, PARTITION, &lbalinfo) == -1) {
      fprintf(stderr, "Cannot identify processor\n");
      exit(1);
   }
   printf("Use processor %s\n", lbalinfo.hostname);
}
```

See Also

See the description of logbal_t described on page 57.

rms_mapToString() — Display range string

Synopsis

```
#include <rmanager/if.h>
char *rms_mapToString(map_t *map);
```

Availability

MEIKOcs2 — MKrms

Description

`rms_mapToString()` reads a `map_t` structure and returns a printable string identifying all the bits that were set to 1. The string is a space separated list of integers or integer ranges (e.g. 1 2 4-7 9 10). The `map_t` structure is indexed from 0.

`map_t` structures are used in the `machine_t` structure (and others) to identify the availability of processors, switches and other components.

Example

The following example will identify the Elan Id's of the processors in your machine:

```
#include <rmanager/uif.h>


main()
{
    machine_t *machine;

    machine = (machine_t*) rms_describe(RMS_MACHINE, 0);
    printf("Machine has processors: %s\n",
        rms_mapToString(&machine->map));
}
```

See Also

See the description of the `map_t` structure on page 61.

rms_moduleTypeString() — Printable module type string

Synopsis

```
#include <rmanager/uif.h>
char *rms_moduleTypeString(ModuleTypes type)
```

Availability

MEIKOcs2 — MKrms

Description

`rms_moduleTypeString()` converts an enumerated `ModuleTypes` value to a printable string. This function is typically used with the `module_t` structure to interpret its `type` field.

Return strings are: `processor`, `switch`, `disk`, `peripheral`, or `unknown`.

Example

Fetch a description for all the modules in the machine and display the module types:

```
#include <rmanager/uif.h>

main()
{
    module_t *m;
    int i = 0;

    /* Repeat for all processors */
    while ((m=(module_t*) rms_describe(RMS_MODULE,i++)) != NULL)
    {
        /* Display module type */
        printf("Type is: %s\n", rms_moduleTypeString(m->type));
    }
}
```

See Also

`rms_describe()`. See also `module_t` on page 62.

rms_ntoelan() — Ethernet address to Elan Id translation

Synopsis

```
#include <rmanager/uif.h>
#include <netinet/if_ether.h>
int rms_ntoelan(struct ether_addr *e);
```

Availability

MEIKOcs2 — MKrms

Description

`rms_ntoelan()` translates the specified ethernet address (e) to the processor's Elan Id. Return values are the Elan Id on success or -1 on failure.

See Also

`rms_elantohost()`, `rms_hosttoelan()`, `rms_elanton()`.

rms_objectString() — Return object type string

Synopsis

```
#include <rmanager/uif.h>
char *rms_objectString(RMS_OBJECT_TYPES type);
```

Availability

MEIKOcs2 — MKrms

Description

rms_objectString() converts an enumerated RMS_OBJECT_TYPES value to a printable string. This function is typically used with the rmsobj_t structure to interpret its type field.

Return strings are: machine, module, board, switch, processor, link, device, configuration, partition, resource, job, or unknown.

Example

Print the type of object at CAN address 0x20000:

```
#include <stdio.h>
#include <sys/canif.h>
#include <rmanager/uif.h>

#define CAN_ADDRESS 0x20000

main()
{
   CAN_ADDR can;
   rmsobj_t *object;

   can.addr_int = CAN_ADDRESS;

   if((object = rms_translate(can)) == NULL) {
      fprintf(stderr, "Cannot get object description\n");
      exit(1);
   }

   printf("Object type: %s\n", rms_objectString(object->type);
}
```

See Also

rms_translate(). See the description of rmsobj_t on page 73.

rms_parseDefaultsFile() — Read system defaults

Synopsis

```
#include <rmanager/uif.h>
sysDefaults *rms_parseDefaultsFile(char *match);
```

Availability

MEIKOcs2 — MKrms

Description

Read system defaults from the `defaults(4)` file.

The `match` argument allows you to select the defaults that apply to a specific partition. Setting `match` to the name of a partition means that you require the defaults that apply to the partition. Specifying a `match` of NULL means that you don't care about partition specific defaults; the default value will be returned even if it only applies to a subset of the partitions in your configuration.

Example

Consider the following extract from a `defaults(4)` file:

```
access-control on parallel batch
timelimit 3000 parallel
```

With a NULL argument `rms_parseDefaultsFile()` returns the default values regardless of partition restrictions:

```
sysDefaults *defaults;

defaults = rms_parseDefaultsFile(NULL):
printf("access-cntrl %d\n",defaults->accessControl);
printf("timelimit %d\n", defaults->timelimit);
```

```
access-control 1
timelimit 3000
```

By requesting the defaults that apply to the `batch` partition the `timelimit` returned by `rms_parseDefaultsFile()` is the default that applies in the absence of a suitable entry in the `defaults(4)` file.

```
sysDefaults *defaults;

defaults = rms_parseDefaultsFile("batch"):
printf("access-cntrl %d\n",defaults->accessControl);
printf("timelimit %d\n", defaults->timelimit);
```

```
access-control 1
timelimit -1
```

See Also

defaults(4). See the sysDefaults structure on page 79.

rms_procStatusString() — Printable processor status

Synopsis

```
#include <rmanager/uif.h>
char *rms_procStatusString(ProcStatus status)
```

Availability

MEIKOs2 — MKrms

Description

`rms_procStatusString()` converts an enumerated `ProcStatus` value into a printable status string. This function is used to display the `status` field in the `proc_t` structure.

Return strings are:

`Configured out`	
`Needs fsck`	`Error`
`Unix booting`	`TFTP boot`
`VROM running`	`Self test`
`Powered down`	`ROM running`
`Unix level 6` *(or 5,4,3,2,1,0)*	`Reset`
`Single user`	`Elan running`
`Can running`	`Unknown` *(value)*

Example

Display the status of all processors:

```
#include <rmanager/uif.h>

main()
{
   proc_t *proc;
   int i = 0;

   while((proc = (proc_t*)rms_describe(RMS_PROC, i++))!=NULL)
      printf("Processor status is: %s\n",
            rms_procStatusString(proc->status));
}
```

See Also

See also `proc_t` on page 67.

rms_procTypeString() — Returns a processor type string

Synopsis

```
#include <rmanager/uif.h>
char *rms_procTypeString(ProcTypes procType);
```

Availability

MEIKOcs2 — MKrms

Description

`rms_procTypeString()` converts an enumerated `ProcTypes` value into a printable string. It is used to display the `type` field in the `proc_t` structure. Return strings are:

`Viking`	`Viking+Ecache`
`Pinnacle`	`unknown`
`CY605`	

The SPARC processor strings may also be appended by either `+VP` or `+cVP` (representing the vector processing units).

Example

Fetch a processor description for all the processors in the machine and display the processor types:

```
#include <rmanager/uif.h>

main()
{
   int i = 0;
   proc_t *proc;

   /* Repeat for all processors */
   while ((proc = (proc_t*) rms_describe(RMS_PROC, i++)) != NULL)
      /* Display the processor's type */
      printf("Type is: %s\n", rms_procTypeString(proc->type));
}
```

See Also

See also `proc_t` on page 67.

rms_resourceStatusString() Printable resource status

Synopsis

```
#include <rmanager/uif.h>
char *rms_resourceStatusString(ResourceStatus status);
```

Availability

MEIKOcs2 — MKrms

Description

`rms_resourceStatusString()` converts an enumerated `ResourceStatus` value into a printable status string. This function is used to display the `status` field in the `resource_t` structure. Return strings are:

`system`	`queued`
`in-use`	`free`
`xtime`	`unknown (`*value*`)`

Example

Display the status of all resources:

```
#include <rmanager/uif.h>

main()
{
   resource_t *resource;
   int i = 0;

   while((resource = (resource_t*)rms_describe(RMS_RESOURCE, i++))!=NULL)
      printf("Resource status is: %s\n",
            rms_resourceStatusString(resource->status));
}
```

See Also

See also `resource_t` on page 71.

rms_setgsid() — Set global session id

Synopsis

```
#include <rmanager/uif.h>
gpid_t rms_setgsid();
```

Availability

MEIKOcs2 — MKrms

Description

rms_setgsid() sets the process group ID and session ID of the calling process to the process ID of the calling process, and releases the process's controlling terminal.

See Also

rms_getgsid().

rms_sigsend() Signal a process

Synopsis

```
#include <rmanager/uif.h>
int rms_sigsend(idtype_t type, gpid_t gpid, int sig);
```

Availability

MEIKOcs2 — MKrms

Description

rms_sigsend() sends a signal to the process or group of processes identified by gpid and type.

The processor component of gpid (i.e. PROCESSOR(gpid)) identifies the target processor. The interpretation of the process component (i.e. PROCESS(gpid)) is dependent on the type argument as described by sigsend(2).

See Also

rms_kill(), signal(5), sigsend(2).

rms_translate() — Translate CAN address to object description

Synopsis

```
#include <rmanager/uif.h>
#include <sys/canif.h>
rmsobj_t *rms_translate(CAN_ADDR can);
```

Availability

MEIKOcs2 — MKrms.

Description

Translates a CAN address to a resource object description.

The `rmsobj_t` structure returned by `rms_translate()` is a generic data type that can be used to represent any of the resource object structures (it is implemented as a C union of all the resource object structures).

Example

The following example determines the type of object at CAN address 0x20000 (this represents processor 0 in module 2):

```
#include <stdio.h>
#include <sys/canif.h>
#include <rmanager/uif.h>

#define CAN_ADDRESS 0x20000

main()
{
   CAN_ADDR can;
   rmsobj_t *object;

   can.addr_int = CAN_ADDRESS;

   if((object = rms_translate(can)) == NULL) {
      fprintf(stderr, "Cannot get object description\n");
      exit(1);
   }

   printf("Object type: %s\n", rms_objectString(object->type);
}
```

See Also

The `rmsobj_t` data structure described on page 73.

rms_ttymsg() — Write message to a session's controlling terminal

Synopsis

```
#include <rmanager/uif.h>
int rms_ttymsg(gpid_t gsid, char *msg)
```

Arguments

MEIKOcs2 — MKrms

Description

Sends a message to the controlling terminal of the session `gsid`.

If PROCESS(gsid) < 0 and PROCESSOR(gsid) < 0 the message is sent to the controlling terminals of all sessions.

If PROCESS(gsid) < 0 and PROCESSOR(gsid) > 0 the message is sent to the controlling terminals of all processes on PROCESSOR(gsid).

The PROCESS() and PROCESSOR() macros are defined in <rmanager/uif.h>.

See Also

`rms_getgsid()`.

rms_version()

Library version string

Synopsis

```
#include <rmanager/uif.h>
char *rms_version();
```

Availability

MEIKOcs2 — MKrms

Description

This function identifies the library version that your application is compiled with.

The associated function `rms_checkVersion()` is used to compare the library version that the application is compiled with against the version of the library that it is linked with.

See Also

`rms_checkVersion()`.

rms_waitpid()

Wait for a parallel program segment to complete

Synopsis

```
#include <rmanager/uif.h>
int rms_waitpid(gpid_t pid, int *status, int options);
```

Availability

MEIKOcs2 — MKrms

Description

`rms_waitpid()` waits for the processes running in the segment to finish and returns the exit status in the manner of `waitpid(2)`. Execution of the calling process is blocked until the segment completes or the calling process itself is interrupted by a signal.

The return value from `rms_waitpid()` is -1 if the function exited as a result of a signal sent to the calling process (or some other reason for failure). Otherwise the return value is 0 and the exit status for the segment is stored in `status` — this may be interpreted using the macros defined in `<sys/wait.h>` and described in `wstat(5)`.

The `pid` argument is the controlling process's global process id, as returned by `rms_getgpid()`.

The `options` argument is currently ignored.

Example

The following example uses `rms_waitpid()` to get the exit status from our example parallel application. Note that the loader program is blocked by the call to `rms_waitpid()` until the parallel application has completed.

```
#include <rmanager/uif.h>
#include <sys/wait.h>
#include <stdio.h>

#define EXAMPLE "/opt/MEIKOcs2/example/csn/csn"

main(int argc, char** argv)
{
   rrequest_t *req;
   int status;
   int i;

   req = rms_defaultResourceRequest();

   if(rms_forkexecvp(req, EXAMPLE, argv) == -1) {
      fprintf(stderr, "Failed to fork application\n");
      exit(1);
```

```
    }

    /* Wait for the parallel program to finish */
    if(rms_waitpid(rms_getgpid(), &status, 0)) exit(1);

    if( WIFEXITED(status) )
       printf("Exited with status: %d\n", WEXITSTATUS(status));
}
```

See Also

rms_forkexecvp(), rms_getgpid(), wstat(5), waitpid(2).

Data Structures 2

The following data structures are used by the resource management user interface library. They are defined in the header file `<rmanager/uif.h>`, and have supporting macro definitions in the header file `<rmanager/machine.h>`.

The resource management system maintains arrays of these structures to describe the resources in the machine. An instance of any of these structures can be fetched by specifying the object type and a logical id to `rms_describe()`. The logical id, present as a field in many of the data structures, is the ordering of the structures by the resource management daemons. Logical id's for modules, boards, processors, and switches begin at 0. Logical id's for jobs and resources are relative to the partition that owns them.

board_t — Board Description

Synopsis

```
board = (board_t*)rms_describe(RMS_BOARD, n);
```

```
typedef struct {
   int id;                       /* logical id of this board */
   BoardTypes type;              /* board type */
   int idb;                      /* id of board in module */
   int moduleId;                 /* module housing this board */
   int baseProc;                 /* first processor */
   int nProcs;                   /* number of processors */
   int baseSwitch;               /* id of first switch */
   int nSwitches;                /* number of switches */
   CAN_ADDR can;                 /* CAN address of H8 on board */
   u_long romRevision;           /* H8 ROM revision */
   GeneralStatus status;         /* board status */
   int serialNumber              /* board serial number */
} board_t;
```

Description

The `board_t` structure describes any of the board types that can be fitted into a module, and may therefore describe processor boards, switch boards, small backplane switch cards, and module controllers. The fields have the following meanings:

Field	Meaning
`id`	The logical id of this board.
`type`	The board's type; this is one of the enumerated `BoardTypes` described below.
`idb`	Id of the board in its module.
`moduleId`	The logical Id of the module that contains this board. You can use this Id as an argument to `rms_describe()` to get the describing structure for the module.
`baseProc`	This is the logical id of the first processor on the board. You can use this id with `rms_describe()` to get the processor's description.
`nProcs`	The number of processors on the board.

Field	Meaning
baseSwitch	The logical Id of the first switch on the board. You can use this id with rms_describe() to get the switch's description.
nSwitches	The number of switches on the board.
can	This is the CAN address of the board. The definition of the CAN_ADDR type is included in <sys/canif.h>.
romRevision	The revision number of the board's H8 ROM.
status	The module's operating status; this is one of the enumerated types GeneralStatus (see below).
serialNumber	The Meiko serial number for this board.

Associated Definitions

The enumerated type BoardStatus defined in the header file <rmanager/machine.h>:

Value	Meaning
BOARD_TYPE_DINO	MK401 single SPARC + I/O board.
BOARD_TYPE_QUATTRO	MK405 quad SPARC board.
BOARD_TYPE_VECTOR	MK403 vector processing element.
BOARD_TYPE_SWITCH_4x4	MK529 four Elite board.
BOARD_TYPE_SWITCH_2x8	MK523 top switches.
BOARD_TYPE_SWITCH_1x16	MK522 two stage switch board.
BOARD_TYPE_SMALL_SWITCH	MK511 module switch card (1 Elite).
BOARD_TYPE_SWITCH_BUFFER	MK512 module switch buffer card.
BOARD_TYPE_CONTROLLER	MK515 module controller.

The enumerated type GeneralStatus defined in the header file <rmanager/machine.h>:

Value	Meaning
STATUS_ERROR	Misbehaving
STATUS_RUNNING	Responding to requests.
STATUS_POWERDOWN	Powered-down.
STATUS_CONFIGOUT	Configured-out.
STATUS_UNKNOWN	Unknown.

config_t — Configuration description

Synopsis

`config = (config_t*)rms_describe(RMS_CONFIGURATION,0);`

```
typedef struct {
  char name[NAME_SIZE]; /* configuration name */
  int nPartitions;      /* number of partitions */
} config_t;
```

Description

Describes the active configuration. Note that there is only one active configuration so the index argument to `rms_describe()` will always be 0.

The fields have the following meanings:

Field	Meaning
`name`	The configuration's name.
`nPartitions`	The number of partitions in the configuration.

device_t — Device description

Synopsis

```
device = (device_t*) rms_describe(RMS_DEVICE, n);
```

```
typedef struct {
    int id;                /* logical id of device */
    DeviceTypes type;      /* device type */
    char *name;            /* manufacturers name */
    int hostId;            /* Host processor */
    int controller;        /* SCSI controller (0-4) */
    int target;            /* target on SCSI bus (0-6) */
    int lun;               /* logical unit number */
    DeviceStatus status[5]; /* device status (upto 5 for RAID)*/
    int moduleId;          /* logical id of module housing device */
    int positionMask;      /* device positions in module */
    int raidLevel;         /* 1,3,5 (UNASSIGNED for single disks) */
    int nPhysDevs;         /* number of physcial devices */
    int slicesUsed;        /* which slices are in use */
} device_t;
```

Description

Describe a SCSI device.

Field	Meaning
`id`	The logical id of this device.
`type`	The device type; this is one of the enumerated `DeviceTypes` described below.
`name`	The device manufacturer's name.
`hostId`	The logical id of the processor that hosts this device. You can pass this id to `rms_describe()` to get the processor's description.
`controller`	Identifies the SCSI controller that the device is connected to. This will be in the range 0–4.
`target`	Identifies the device id on the SCSI bus. This will be in the range 0–6.
`lun`	Logical unit number (for use with RAID arrays).

Field	Meaning
`status`	An array of status values; up to 5 values may be recorded for RAID arrays. Each value may be one or more of the enumerated `DeviceStatus` values listed below.
`moduleId`	The logical id of the module that contains this device. You can pass this id to `rms_describe()` to get the module's description.
`positionMask`	Bit mask indicating the device's position in the module.
`raidLevel`	Identifies the RAID level as 1, 3, or 5. This field will be set to RMS_UNASSIGNED if the device is not part of a RAID array.
`nPhysDevs`	The number of physical devices that constitute this device.
`slicesUsed`	A bit mask identifying the slices that are in use; bits 0–7 are used.

Associated Definitions

The enumerated `DeviceTypes` type defined in `<rmanager/machine.h>`:

Value	Meaning
`DEVICE_TYPE_QITC`	Quarter inch tape device.
`DEVICE_TYPE_CDROM`	CD-ROM drive.
`DEVICE_TYPE_EXABYTE`	8mm tape device.
`DEVICE_TYPE_DISK`	3.5" disk device.
`DEVICE_TYPE_DISKARRAY`	Array of 3.5" disk devices.
`DEVICE_TYPE_UNKNOWN`	Unknown device type.

The enumerated `DeviceStatus` type defined in `<rmanager/machine.h>`:

Value	Meaning
DEVICE_PRESENT	Device has been detected.
DEVICE_POWERON	Power has been applied to the device.
DEVICE_POWEROFF	Power to the device is off.
DEVICE_RUNNING	Device is in operation.
DEVICE_ERROR	An error has been detected.
DEVICE_UNKNOWN	Unknown status.

fsys_t — Filesystem description

Synopsis

`fsys = (fsys_t*) rms_describe(RMS_FSYS, n);`

```
typedef struct {
   int id;              /* logical id of fsys */
   int type;            /* fsystem type (as returned by sysfs) */
   char slice[8];       /* c?t?t?s? */
   char *mountp;        /* mount point */
   int nDevices;        /* number of devices */
   int *deviceIds;      /* device ids */
   int nServers;        /* processors that serve this fsystem */
   int *serverIds;      /* their ids */
   map_t *nfsClients;   /* clients that mount filesystem */
} fsys_t;
```

Description

Describes a filesystem (including PFS and RAID filesystems).

Field	Meaning
`id`	The logical id of this filesystem description.
`type`	The filesystem's type; this is a filesystem type index as returned by `sysfs`(2).
`slice`	A string in the form c*x*t*x*d*x*s*x* identifying controller, target, logical unit number (LUN), and slice.
`mountp`	The filesystem's mount point.
`nDevices`	Number of devices used by this filesystems (applicable to PFS and RAID systems).
`deviceIds`	An array of logical device identifiers, one identifier for each of the `nDevices`; pass these to `rms_describe()` to get a description of the devices (instances of the `device_t` structure).

Field	Meaning
`nServers`	The number of servers of this filesystem.
`serverIds`	An array of logical processor identifiers, one for each of the `nServers`. You can use these id's with `rms_describe()` to get a description of the processors (instances of `proc_t` structures).
`nfsClients`	A processor map, indexed by logical id, identifying the processors that mount this filesystem.

job_t — Job (program) description

Synopsis

```
job = (job_t*) rms_describe(RMS_JOB, n);
```

```
typedef struct {
   gpid_t gpid;          /* gpid of controlling process */
   uid_t uid;            /* uid of owner */
   gpid_t rpid;          /* process allocating resource */
   int rid;              /* resource identifier */
   time_t start;         /* scheduled/actual start time */
   int baseProc;         /* first processor used for job */
   int nProcs;           /* number of processors */
   int memory;           /* memory (in MBytes) */
   JobStatus status;     /* status of job */
   char name[NAME_SIZE]; /* name of program */
} job_t
```

Description

Describes a parallel program. Identifies the program name, resource requirements, and owner.

Note that logical job id's are relative to the partition that is running the job. The logical id for the first job within a partition can be determined by specifying the partition id to the macro PARTITION_BASE(), which is defined in `<rmanager/uif.h>`. Alternatively it can be determined from the `partition_t` structure.

The fields have the following meaning:

Field	Meaning
`gpid`	The global process id of the job's controlling process.
`uid`	The user id of the owner of this job.
`rpid`	The global process id of the process that allocated the resource that is used by this job.
`rid`	The logical id of the resource used by this job. You can call `rms_describe(RMS_RESOURCEBYID,rid)` to get a `resource_t` structure describing the resource.
`start`	The time the job was started.

Field	Meaning
baseProc	The logical id of the first processor used by this job. You can use this id to select the appropriate proc_t structure with rms_describe().
nProcs	The number of processors used by this job. Jobs use a contiguous range of processors with logical id's from baseProc to (baseProc+nProcs-1).
memory	The maximum memory required by this job (in Mbytes).
status	The status of this job. This will be one or more of the enumerated JobStatus types — see below.
name	The name of the program.

Associated definitions

The enumerated type JobStatus defined in the header file <rmanager/uif.h>:

Value	Meaning
JOB_STARTING	Job has started.
JOB_RUNNING	Job is running.
JOB_EXITED	Job has finished.
JOB_KILLED	Job was killed by a signal.
JOB_NOTRUN	Job failed to run.
JOB_FINISHED	Job has run and now finished.
JOB_ZOMBIE	Job was stopped (killed/exited/not-run) abnormally.
JOB_LAUNCHED	Job is either running or in a zombie state

logbal_t — Describes the least loaded processor

Associated Functions

Used by rms_logbal().

```
typedef struct {
   char hostname[NAME_SIZE]; /* name of host to use */
   long addr;                /* IP address of host to use */
} logbal_t;
```

Description

Used by rms_logbal() to identify the least loaded processor in a partition. The resource management system uses the statistic specified in the defaults(4) file to measure processor loading.

The fields have the following meanings:

Field	Meaning
hostname	The hostname of the least loaded processor.
addr	The IP address of the least loaded processor

machine_t — Machine description

Synopsis

```
machine = (machine_t*) rms_describe(RMS_MACHINE, 0);
```

```
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 hostId;             /* machine host id */
    int serialNumber;       /* machine serial number */
    int gCANs;              /* number of global CAN networks */
    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 */
    time_t timestamp;       /* last modification time */
    time_t started;         /* time mmanager started */
    int nFsys;              /* number of file systems */
} machine_t;
```

Description

Used to describe the hardware components of your machine. The fields have the following meanings:

Field	Meaning
`nLevels`	The number of levels in the switch network.
`nModules`	The total number of modules in the system (includes switch, processor, and peripheral modules).
`nBoards`	The number of boards in the machine. This count includes all the boards in all the modules, and will include module switch boards, module control boards, processor boards, and switch boards.
`baseProc`	The Elan Id of the first processor in the machine.

Field	Meaning
`topProc`	The Elan Id of the last processor in the machine.
`nProcs`	The number of processors in the machine.
`nSwitches`	The number of switches in the machine.
`nDevices`	The number of devices in the machine.
`nBays`	The number of bays in the system.
`layers`	A bit mask of network layers — bit *n* represents layer *n*. Bits are set to indicate that a layer is available.
`hostId`	The machine's host id.
`serialNumber`	The machine's serial number.
`gCANs`	The number of global CAN networks in this system.
`name`	The machine's name.
`map`	A bit array showing the number of processors in the system. Within the bit array processors are represented by a single bit and are ordered by their Elan Id. Bits are set for processors that exist, and cleared for those that do not.
`proc_map`	This a processor map that shows the configuration state of the processors in the machine. It is a copy of the `map` field with configured-in processors having their bits set, and configured-out processors having their bits cleared.
`sw_map`	Shows the availability of switches. Each switch device in the machine has a bit in the array. Switches that are configured-in have their bit set; configured-out switches have their bits cleared. Switches are ordered in the bit array by using their logical id.
`board_map`	Shows the availability of boards (this will include module switch boards, module control cards, processor cards, and module switch cards). Each board in the machine is assigned a bit in the array. Boards that are configured-in have their corresponding bit set. Boards are ordered in the bit array by using their logical id.

Field	Meaning
timestamp	Last modification time for this structure.
started	Start time for the machine manager.
nFsys	The number of filesystems.

Example

The following example tests the configuration state of the switch with logical id 3. If the switch is configured-in we use `rms_describe()` to fetch the describing `switch_t` structure. Note that the ordering of bits in the `sw_map` uses the same logical id that is used with `rms_describe()`.

```
machine_t *machine;
switch_t *switch;

/* Get machine description */
if((machine=(machine_t*)rms_describe(RMS_MACHINE,0)) == NULL) {
   fprintf(stderr, "Cannot get machine description\n);
   exit(1);
}

/* Test switch availability */
if(MAPISSET(3, &machine->sw_map)) {
   printf("Switch 3 is available\n");

   /* Get more info about this switch */
   if((switch=(switch_t*)rms_describe(RMS_SWITCH,3)) == NULL) {
      fprintf(stderr, "Cannot get switch description\n");
      exit(1);
   }
   else
      printf("Switch is at level %d\n", switch->level);
}
```

See Also

See also the description of the `map_t` structure on page 61.

map_t — General purpose bit array

Associated Functions `rms_configure()`.

Description

The `map_t` structure is used as an array of `MAX_SWITCHES` bits.

Instances of these bit arrays are held within the `machine_t` structures (describing the resources within the machine) to describe the availability of processors and switches. Resource management functions that effect the availability of these components also notify the change by setting/clearing the appropriate bit within these arrays.

Associated Definitions

A number of macro's are defined in `<rmanager/uif.h>` to manipulate the bits within `map_t` structures. Each take a pointer to a map `map_t` structure. These are:

Macro	Purpose
`MAP_SET(p,&map)`	Set bit `p` in the specified `map`.
`MAP_CLR(p, &map)`	Clear bit `p` in the specified `map`.
`MAP_ISSET(p,&map)`	Returns true (non-zero) if bit `p` in the `map` is set.
`ZERO_MAP(&map)`	Clear all bits in the `map`.

Example

In the following example `rms_describe()` is used to get a description of the machine (an instance of a `machine_t` structure). Bit 1 in the `proc_map` field is tested to check the availability of the processor with Elan Id 1:

```
machine_t *machine;

/* Get a description of the machine */
if((machine=(machine_t*)rms_describe(RMS_MACHINE,0))==NULL){
   fprintf(stderr, "Cannot get machine description\n");
   exit(1);
}

/* Is processor with Elan Id 1 configured-in */
if(MAP_ISSET(1,&machine->proc_map))
   printf("Processor 1 is available\n");
```

See Also

See also the description of the map fields within the `machine_t` structure.

module_t — Module description

Synopsis

```
module = (module_t*) rms_describe(RMS_MODULE, n);
```

```
typedef struct {
   int id;                  /* logical id of this module */
   ModuleTypes type;        /* module type */
   CAN_ADDR can;            /* CAN address of controller */
   int baseBoard;           /* id of the first board */
   int nBoards;             /* number of boards */
   int baseProc;            /* first processor in module */
   int nProcs;              /* number of processors */
   int baseDevice;          /* id of first device */
   int nDevices;            /* number of devices */
   int position;            /* physical position in machine */
   int level;               /* network level */
   int netId;               /* network id */
   int plane;               /* plane number */
   int layer;               /* network layer number */
   int gCAN;                /* connected gCAN (-ve if none) */
   int controllerId         /* board id of controller */
   int power                /* power is good */
   char *console;           /* Cmd to grab console */
} module_t;
```

Description

A description of a module. The fields have the following meanings:

Field	Meaning
`id`	The logical id of this module.
`type`	The module type. This will be one of the enumerated `ModuleTypes` described below.
`can`	This is the CAN address of the module's controller. The definition of the CAN_ADDR type is included in `<sys/canif.h>`.
`baseBoard`	This is the logical id of the first board in the module. You can use this id to select the appropriate `board_t` structure with `rms_describe()`.

Field	Meaning
`nBoards`	The number of boards in the module. This count includes processor boards, switch boards, the module control board, and the small switch cards that can be plugged into the rear of the processor modules.
`baseProc`	This is the logical id of the first Unix processor in the module; you can use this id with `rms_describe()`.
`nProcs`	The number of processors in the module.
`baseDevice`	The is the logical id of the first device in the module; you can use this id with `rms_describe()`.
`nDevices`	The number of devices in the module.
`position`	This is the physical position of the module in the machine. This is specified by the Installation Engineer in the `machine.des`(4) file.
`level`	This is the switch level that the module is connected to.
`netId`	This is the module's network address.
`plane`	This field identifies the switch plane that this module contains.
`layer`	This field identifies the switch layer that this module contains (bit mask in which bit *n* represents layer *n*).
`gCAN`	If the module is a G-CAN router then this field is the id of its global CAN network. Otherwise it is negative.
`controllerId`	Logical id of board description for the module controller.
`power`	The status of the power supply voltages; a non-zero value indicates that the module power supply is good.
`console`	The command used to grab a console.

Note that the allocation of logical id's for processors, or boards, or devices is contiguous. The range of logical id's for all the processors in the module will therefore range from `baseProc` to (`baseProc+nProcs-1`).

Associated Definitions

The enumerated type `ModuleTypes` is defined in `<rmanager/uif.h>`:

Value	Meaning
MODULE_TYPE_PROCESSOR	Processor module.
MODULE_TYPE_SWITCH	Switch module.
MODULE_TYPE_PERIPHERAL	Peripheral (disk) module.

partition_t — Partition description

Synopsis

```
partn = (partition_t*) rms_describe(RMS_PARTITION, n);
```

```
typedef struct {
   int id;                /* logical id of partition */
   char name[NAME_SIZE];  /* partition name */
   int baseProc;          /* first processor */
   int topProc;           /* last processor */
   int nProcs;            /* number of processors */
   int baseResource;      /* first resource in partition */
   int nResources;        /* number of resources */
   int baseJob;           /* first job */
   int nJobs;             /* number of active jobs */
   time_t start;          /* time pmanager started */
   int active;            /* running or not */
   map_t map;             /* processor map */
} partition_t;
```

Description

Describes a partition. The fields have the following meanings:

Field	Meaning
`id`	The logical id of this partition.
`name`	The partition's name.
`baseProc`	The Elan Id of the first processor in the partition.
`topProc`	The Elan Id of the last processor in the partition.
`nProcs`	The number of processors in the partition.
`baseResource`	The logical id of the first resource held within this partition. You can use this id with `rms_describe()` to obtain the description of the first resource in the partition.
`nResources`	The number of resources in the partition.
`baseJob`	The logical id of the first job in this partition. You can use this id with `rms_describe()` to obtain a description of the first job in this partition.
`nJobs`	The number of active jobs in the partition.

Field	Meaning
`start`	Start time for the partition manager.
`active`	Either 0 or 1, will be set to 0 if the partition is down.
`map`	A map of the processors that are in this partition. The map is a bit array in which processors are represented by a single bit and are ordered by their Elan Ids. Bits are set to indicate that a processor is a member of the partition, and cleared if it is not.

See Also

See also the description of `map_t` on page 61.

proc_t — Processor description

Synopsis

```
proc = (proc_t*) rms_describe(RMS_PROC, n);
```

```
typedef struct {
    int id;                  /* logical id of this processor */
    int idp;                 /* id of processor on board */
    int boardId;             /* board id */
    int moduleId;            /* module id */
    ProcTypes type;          /* processor type */
    int memory;              /* memory (in MBytes) */
    int level;               /* switch network level */
    int elanId;              /* elan id (route down) */
    CAN_ADDR can;            /* CAN address of processor */
    ProcStatus status;       /* processor status */
    ulong romRevision;       /* Open Boot ROM revision */
    char *name;              /* Unix hostname */
    Gender gender;           /* Processor's role */
    int bootId;              /* Processor to boot from */
    int nDevices;            /* Number of devices */
    int *deviceIds;          /* Device identifiers */
    int nFsys;               /* Number of filesystems */
    int *fsysIds;            /* Filesystem identifiers */
    unsigned long iaddr;     /* Internet address */
} proc_t;
```

Description

Description of a Unix SPARC processor. The fields have the following meanings:

Field	Meaning
`id`	The logical id of this processor.
`idp`	The logical id of this processor relative to the others on the same board.
`boardId`	The logical Id of the processor's board.
`moduleId`	The logical Id of the processor's module.
`type`	The processor's type. One of the enumerated `ProcType` values described below. May also be one of the enumerated `VpuTypes` values if VPU co-processors are fitted.
`memory`	The amount of memory (in Mbytes).
`level`	This processor's level in the switch network.

Field	Meaning
elanId	This processor's Elan Id.
can	The CAN address of the processor's controlling H8 processor. The definition of the CAN_ADDR type is included in <sys/canif.h>.
status	The processor's status. One of the enumerated ProcStatus values described below.
romRevision	The revision number of the processor's Open Boot ROM.
name	The processor's Unix hostname.
gender	Describes the processor's role; this will be one or more of the enumerated Gender types described below.
bootId	Logical id of this processor's server.
nDevices	The number of attached devices.
deviceIds	An integer array of logical device id's. Use these with rms_describe() to get a description of the devices.
nFsys	The number of filesystems.
fsysIds	An integer array of logical filesystem id's. Use these with rms_describe() to get a description of the filesystems.
iaddr	The processor's internet address.

Associated definitions

The enumerated type ProcTypes is used to initialise the least significant byte of the proc_t.type field:

Value	Meaning
PROC_TYPE_605	Ross 605.
PROC_TYPE_PINNACLE	Ross Pinnacle.
PROC_TYPE_VIKING	Texas Instruments Viking.
PROC_TYPE_VIKING_ECACHE	TI Viking with external cache.
PROC_TYPE_H8	H8 processor.

The enumerated type `VpuTypes` is (optionally) used to initialise the second byte of the `proc_t.type` field:

Value	Meaning
VPU_TYPE_514	Non-cache coherent VPU.
VPU_TYPE_534	Cache coherent VPU.

Enumerated type `ProcStatus` — Processor Status definitions. Defined in `<rmanager/machine.h>`.

Value	Meaning
PROC_STATUS_RESET	Processor held in reset.
PROC_STATUS_ROM_RUNNING	At 'OK' (boot ROM prompt).
PROC_STATUS_SELF_TEST	Running remote self test.
PROC_STATUS_TFTP_LOAD	ROM loading external code.
PROC_STATUS_BOOTING	ROM about to run external code.
PROC_STATUS_ERROR	Processor is misbehaving.
PROC_STATUS_NEEDSFSCK	Disk needs checking.
PROC_STATUS_CAN_RUNNING	The CAN module has been loaded.
PROC_STATUS_RUNLEVEL_S	Unix running single user mode.
PROC_STATUS_RUNLEVEL_*0–6*	Unix going to run level 0–6.
PROC_STATUS_POWERDOWN	Power is down on module.
PROC_STATUS_CONFIGOUT	Processor is configured out.
PROC_STATUS_VROM	Processor is running in VROM.

Enumerated type `Gender` — processor roles. Defined in `<rmanager/machine.h>`.

Value	Meaning
GENDER_MEDIA	Media server (QITC/CD-ROM etc.)
GENDER_SERVER	Server for clients/filesystems.
GENDER_CLIENT	Client (no exported filesystems).
GENDER_GATEWAY	Network gateway.
GENDER_CONSOLE	Console host.

resource_t — Resource description

Synopsis

Returned by `rms_describe(RMS_RESOURCE...)`

```
typedef struct {
    int id;                         /* id (sequence number) */
    gpid_t gpid;                    /* process holding resource */
    gpid_t gsid;                    /* controlling session */
    uid_t uid;                      /* uid of owner */
    int baseProc;                   /* first processor */
    int nProcs;                     /* number of processors */
    time_t start;                   /* time queued/allocated */
    time_t timelimit;               /* allocation time in secs */
    int priority;                   /* priority of request */
    ResourceStatus status;          /* status */
    char partition[NAME_SIZE];      /* partition name */
} resource_t;
```

Description

Describes a resource.

Note that logical resource id's are relative to the partition that allocated the resource. The logical id for the first resource within a partition can be determined by specifying the partition id to the macro PARTITION_BASE(), which is defined in `<rmanager/uif.h>`. Alternatively it can be determined from the `partition_t` structure.

The fields have the following meanings:

Field	Meaning
`id`	Logical id of this resource.
`gpid`	The process id of the process that is holding this resource.
`gsid`	The global session id of the controlling session.
`uid`	The user id of the owner of this resource.
`baseProc`	The logical id of the first processor in this resource.
`nProcs`	The number of processors in this resource. Resources contain a contiguous range of processors with logical id's from `baseProc` to (`baseProc+nProcs-1`).

Field	Meaning
`start`	The time that the resource was either queued or allocated (to determine which applies look at the `status` field).
`timelimit`	The maximum time the resource can be held for specified in seconds. The time limit is inherited from the resource request structure (`rrequest_t`). -1 means no limit.
`priority`	The priority of the request.
`status`	The status of the resource. This is a bit mask that can be set/tested by the enumerated `ResourceStatus` values (see below).
`partition`	The name of the partition that this resource is allocated from.

Associated definitions

Enumerated type `ResourceStatus` — Resource Status values. Defined in `<rmanager/uif.h>`.

Value	Meaning
`RESOURCE_FREE`	Resource is free.
`RESOURCE_INUSE`	Resource in use.
`RESOURCE_QUEUED`	Resource request is queued.
`RESOURCE_XTIME`	Resources are being freed. Out of time and now in grace period.
`RESOURCE_SUSPENDED`	Use of resource has been suspended.
`RESOURCE_ESUSPENDED`	Externally suspended.
`RESOURCE_SYSTEM`	Resource in use by the system.

rmsobj_t — Generic resource description

Associated Functions `rms_translate()`.

```
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;
      resource_t  resource;
      job_t job;
      fsys_t fsys;
   } objs;
} rmsobj_t;
```

Description

This is a C union of several resource management data structures. The `rmsobj_t` structure is used to simplify the interface to functions that can operate on more than one type of resource object.

The fields have the following meanings:

Field	Meaning
`type`	The type of object described by this structure; one of the `RMS_OBJECT_TYPES` enumerated values (see below).
`obj`	A C union of the following data types:

`machine_t`	Machine description.
`module_t`	Module description.
`board_t`	Board description.
`switch_t`	Switch description.
`proc_t`	Processor description.
`device_t`	Device description.
`config_t`	Configuration description.

Field	Meaning	
	partition_t	Partition description.
	resource_t	Resource description.
	job_t	Job description.
	fsys_t	Filesystem description.

Associated Definitions

The enumerated RMS_OBJECT_TYPES values defined in <rmanager/uif.h>.

RMS_MACHINE	RMS_MODULE
RMS_BOARD	RMS_SWITCH
RMS_PROC	RMS_DEVICE
RMS_CONFIGURATION	RMS_PARTITION
RMS_RESOURCE	RMS_JOB
RMS_FSYS	

Example

`rms_translate()` takes a CAN address and returns a pointer to a resource management structure describing the object at that address. The type of the object is unknown until after the function call so a generic object type simplifies the functional interface:

```
CAN_ADDR can = 0x8400;
rmsobj_t *object;

if((object = rms_translate(can)) == NULL) {
   fprintf(stderr, "Cannot get object description\n");
   exit(1);
}

printf("Object type is %s\n", rms_objectString(object->type));
```

rrequest_t — Resource request

Associated Functions

Used by rms_forkexecvp(), rms_allocate(), rms_defaultResourceRequest().

```
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 rid;                     /* resource identifier */
   int flags;                   /* options on request */
   int routeTable;              /* route table to use */
   char partition[NAME_SIZE]; /* partition to use */
} rrequest_t;
```

Description

The rrequest_t structure is used to describe the resources required by a parallel application — it is passed as an argument to rms_forkexecvp() or rms_allocate()

An instance of the rrequest_t structure is created and initialised with the function rms_defaultResourceRequest(); the default values are read from the user's environment.

Field	Meaning
baseProc	The first processor that is required to run the user's program (the numbering is relative to the start of the partition and begins at 0).
nProcs	The number of processors to use.
memory	The maximum memory required by the program in Mbytes.
timelimit	Maximum run-time of the program specified in seconds. The program is sent a SIGXCPU after this period has elapsed, and a SIGKILL after a short grace-period.
rid	The logical id of a resource_t structure describing an allocated resource. This allows an existing resource to be used.

Field	Meaning
flags	1 bit per flag, set to 1 to enable. The enumerated type RequestFlags includes useful definitions (see below).
routeTable	Elan route table to use.
partition	The name of the partition to use (max. length currently 32 characters).

Associated Definitions

The enumerated type RequestFlags (defined in <rmanager/uif.h>) can be used to set bits in the rrequest_t.flags field:

Value	Meaning
REQUEST_DEBUG	Run program under the debugger.
REQUEST_CORE	Allow core file creation.
REQUEST_SEQ	Force no barrier synchronisation of slaves with host (treat as a Unix sequential program). The resource management system normally makes its own evaluation.
REQUEST_VERBOSE	Enable verbosity.
REQUEST_TIMING	Time the loading process and write to stdout.
REQUEST_TAG	Tag output with processor Id's.
REQUEST_EXTRAVERBOSE	Enable more verbose output.
REQUEST_BARRIER	Force barrier synchronisation of slaves with host (treat as a parallel application). The resource management system normally makes its own evaluation.
REQUEST_IMMEDIATE	Fail if resource is not immediately available; by default the resource request blocks the calling process until the resource is allocated.

Example

The following code fragment sets the debug and core file creation flags:

```
rrequest_t rreq;

rreq.flags = REQUEST_DEBUG | REQUEST_CORE;
```

switch_t — Switch Description

Synopsis

Returned by `rms_describe(RMS_SWITCH...)`

```
typedef struct {
    int id;                  /* logical id of this switch */
    int sid;                 /* Physical id of this switch */
    int level;               /* switch network level */
    int netId;               /* network id */
    int plane;               /* plane number */
    int layer;               /* network layer number */
    int moduleId;            /* module id */
    GeneralStatus status;    /* switch status */
    CAN_ADDR can;            /* can address of controlling H8 */
    int chip;                /* id on local H8 controller */
    int boardId;             /* board id */
} switch_t;
```

Description

Describes an Elite network switch, including its position in the switch network and the state of its links. The fields have the following meanings:

Field	Meaning
`id`	The logical id of this switch. See below.
`sid`	The physical id of this switch. See below.
`level`	The level in the switch network that this switch is placed.
`netId`	The switch's network Id. See below.
`plane`	The switch plane that the switch is in.
`layer`	The network layer that the switch is in.
`moduleId`	The logical id of the module that contains this switch.
`status`	The switch's operating status; this is one of the enumerated types `GeneralStatus` (see below).
`can`	This is the CAN address of the board that contains the switch. The definition of the CAN_ADDR type is included in `<sys/canif.h>`.
`chip`	The chip number on the controlling H8.
`boardId`	The logical id of the board.

Switch Numbering

Each switch has three identifiers (the `id`, `sid`, and `netId` fields in the `switch_t` structure).

The `id` is the logical id of this switch and relates solely to the ordering of the `switch_t` structures in the resource management system's list (i.e. the index that is passed to `rms_describe()`).

The `netId` is the decimal representation of the switch's network address which describes the route to the switch from the top of the network. All switches at the top of the network have a netId of 0. Remember that network routes take the form `<0-7>.<0-3>.<0-3>...`, so the switch at level 1 with the route 5.1 has Elan Id 21 (convert 5.1 to binary 101.01 and then to decimal). See the document entitled *Communication Network Overview* for a description of network addressing.

Switch id's (the `sid` field) are unique to each switch and identify the physical position of each switch within the network. The range of ids assigned to each network layer is determined by the network size (which can be determined using the definitions in `<rmanager/network.h>`). Switch id's begin at 0 in network layer 0, and are assigned from the top network stage to the bottom, and from left to right within each stage. The numbering for subsequent network layers continues where the previous range ended. When the network is incomplete there will be corresponding gaps in the assignment of switch id's. Consider, for example, a 3 stage network in which layer 0 switches have id's in the range 0–79; the top 16 switches have id's 0–15, the 32 switches at level 1 have id's in the range 16–47, and the 32 switches at the lowest level have id's 48–79.

Associated Definitions

The enumerated type `GeneralStatus` defined in the header file `<rmanager/machine.h>`:

Value	Meaning
`STATUS_ERROR`	Misbehaving
`STATUS_RUNNING`	Responding to requests.
`STATUS_POWERDOWN`	Powered-down.
`STATUS_CONFIGOUT`	Configured-out.
`STATUS_UNKNOWN`	Unknown

sysDefaults System defaults

Synopsis

`sysDefaults = rms_parseDefaultsFile(match);`

```
typedef struct {
   ulong romRevision;          /* minimum openboot ROM revision */
   ulong h8RomRevision;        /* minimum H8 revision date */
   int informationHiding;      /* only tell users about themselves */
   int canDo;                  /* machine has CAN */
   char partition[NAME_SIZE]; /* default partition */
   int timelimit;              /* timelimit on resource allocation */
   int gracePeriod;            /* grace period for timelimits */
   int haltOnError;            /* rms should stop on serious errors */
   int accounting;             /* accounting system is enabled */
   int accessControl;          /* enable access control checking */
   int logPermErrors;          /* log access permission errors */
   int logStats;               /* log resource usage statistics */
   int acctInterval;           /* sampling interval for accounting */
   char tmpdir[NAME_SIZE];     /* path to local tmp filespace */
   int logbalStatistic;        /* load balancing statistic */
   char logbalHosts[NAME_SIZE]; /* default places to log users in */
   int logfileSize;            /* logfile size in KBytes */
   int bootTime;               /* time allowed to boot */
   int haltTime;               /* time allowed to halt */
   int resetTime;              /* time allowed to pulse reset */
   int pulseTime;              /* time allowed to reset and test */
   int maxDeltaTime;           /* time between acct 'busy' reports */
   int maxIdleTime;            /* time between acct 'idle' reports */
} sysDefaults;
```

Description

This structure records system default values read from the `defaults`(4) file. Each entry in the `defaults` file has a corresponding field in the `sysDefaults` structure.

Field	Meaning
`romRevision`	Minimum permitted OpenBoot ROM revision to be used by any processor in this system. Default is 95.
`h8RomRevision`	Minimum permitted H8 ROM revision date to be used with any processor in this system. Default is 0x93090611.
`informationHiding`	Enable information hiding if this variable is non-zero (only tells users about resources that are available to them). Default is 0.
`canDo`	Specifies that this system is fitted with a CAN bus if this variable is non-zero. Default is 1.
`partition[]`	Default partition to use when no partition is explicitly named by user applications. Default partition is `login`.
`timelimit`	Timelimit, in seconds, on resource allocation. Jobs will be signalled (SIGXCPU) after this timelimit has elapsed. Default is -1 (no limit).
`gracePeriod`	Grace period, in seconds, for timelimits; jobs are permitted this period to respond to the timelimit signal; after the grace period has elapsed the job is killed (sent SIGKILL). Default is 10.
`haltOnError`	The resource management system will stop on serious errors if this variable is non-zero. Default is 1.
`accounting`	The resource management system accounting is enabled if this variable is non-zero. Default is 0.
`accessControl`	Access control is enabled if this variable is non-zero. Access to partitions is permitted to users listed in the `names`(4)/`permissions`(4) files. Default is 1.

Field	Meaning
`logPermErrors`	Enables logging by the partition managers of security violations when this variable is non-zero. The logfile is `/opt/MEIKOcs2/etc/`***name***`/security.log`. Default is 1.
`logStats`	Log resource usage statistics if this variable is non-zero. Default is 0. **This option currently unused.**
`acctInterval`	Sampling interval, in seconds, for resource accounting. Default is 30.
`tmpdir[]`	Path to local temporary filespace. Default is `/tmp`.
`logbalStatistic`	Load balancing statistic: 0 = User CPU, 1= Kernel CPU, 2= Idle CPU, 3 = Disk transfer rate, 4=page in+out rate, 5=swap in+out rate, 6=interrupts, 7=packets, 8=contexts, 9=load. Default is 9 (load).
`logbalHosts[]`	Identifies hosts to `logbal(1)` for load loadbalanced command shells. This variable is a space separated list of hostnames. Default is all processors in the `login` partition.
`logfileSize`	File size in Kbytes for the machine manager's event logs (this size is a maximum size; the logfiles are cyclic buffers). Default is 256.
`bootTime`	Time allowed to boot a processor. Default is 500.
`haltTime`	Time allowed to halt a processor. Default is 300.
`resetTime`	TIme allowed to pulse reset on a processor. Default is 400.
`pulseTime`	Time allowed to reset and test. Default is 45.
`maxDeltaTime`	Time between accounting "busy" reports. Default is 120 seconds.
`maxIdleTime`	Time between accounting "idle" reports. Default is 60 seconds.

Example Programs 3

Introduction

This chapter includes a number of example `librms` programs showing the most commonly used `librms` functions and data structures[1].

The following command line is used to compile all of the `librms` programs described in this chapter:

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

Program Loader

This example demonstrates a simple program loader offering a subset of the functionality of `prun`. The usage synopsis for this example is:

```
loader [-v] [-n nprocs] [-p partition] program
```

1. These programs are intended to be short examples of librms functionality and do not therefore include all the error checking, functionality, and style of commercial applications.

Program Description

The program begins with a call to `rms_defaultResourceRequest()` which reads a default resource specification from the environment and gives the user the option of specifying the target resource either by explicit use of the RMS environment variables, or by running the loader from a command shell with resources allocated to it (see `allocate(1)`).

```
#include <stdio.h>
#include <rmanager/uif.h>

extern int optind;
extern char *optarg;

main(int argc, char** argv)
{
   int status;
   gpid_t pid;
   rrequest_t *resources;
   int opt;

   /* Get default resource spec from the environment */
   resources = rms_defaultResourceRequest();
```

Having fetched the resource specification from the environment the user can override some attributes with command line arguments. Note that the loader program will terminate if command line arguments are incompatible with resources that have already been allocated to the command shell. To overcome this you could include for the `p` and `n` options a test of the `rrequest.rid` field, which will be a positive integer if resources have already been allocated; if they have been allocated you should ignore the user's partition specification and check that the specified processor count is less than or equal to that which has already been allocated.

```
   /* Override default with command line args */
   while((opt = getopt(argc, argv, "p:n:v")) != -1) {
      switch(opt) {
      case 'p':
         strncpy(resources->partition, optarg, NAME_SIZE);
         break;
      case 'n':
         resources->nProcs = atoi(optarg);
```

```
            break;
        case 'v':
            resources->flags |= REQUEST_VERBOSE;
            break;
        default:
            break;
        }
    }
```

The user's parallel application is executed on the target resource by `rms_forkexecvp()`. Note that if no target partition has yet been specified (either in the user's environment or on the command line) `rms_forkexecvp()` will determine a default partition from the system defaults file. `rms_forkexecvp()` allocates the target resource, if it hasn't already been allocated, starts the application, and then returns control to the calling process as soon as the processes in the parallel segment have executed their start-up barrier.

```
    /* Run the program on the resource */
    if(rms_forkexecvp(resources, argv[optind], &argv[optind])) {
        fprintf(stderr,"%s: Failed to execute on partition %s\n",
                  argv[0], resources->partition);
        exit(1);
    }
```

To prevent the loader program from terminating before the parallel segment has completed (which would cause the whole application to finish) a call to `rms_waitpid()` is used to block the loader program. `rms_waitpid()` is passed the global process id of the application's controlling process (i.e. the loader program) as returned by the call to `rms_getgpid()`. The exit status for the parallel segment is returned in the `status` variable and echoed to the screen when verbose reporting is enabled.

```
    /* Get pid of controlling process */
    pid = rms_getgpid();

    /* Wait for all processes to terminate */
    rms_waitpid(pid, &status, 0);

    /* Display exit status if verbosity is enabled */
```

```
    if(resources->flags & REQUEST_VERBOSE)
        printf("%s: exit status %x\n", argv[optind], status);
}
```

Examining the Configuration

This example examines the resources in a partition; it lists the processor types, their status, Elan Id's, and hostnames. A program of this type might be useful to those users who cannot use Pandora to visualise the availability and configuration of resources, and require more information than is provided by either `rinfo`(1) or `rcontrol`(1m).

The usage synopsis for this example is:

```
config [partition]
```

The program's output for a 1 processor partition might look like:

```
cs2-0: config p1
Partition p1 has 1 processor:

  Proc type: Viking+Ecache
     ElanId: 84
     Status: Unix level 3
   Hostname: cs2-84
```

Program Description

The program begins by fetching a default resource specification from the environment (with `rms_defaultResourceRequest`()) which will allow the program to target the resources that have been allocated to the command shell (if any), or to use the partition specified by the user's RMS_PARTITION environment variable (if set).

```
#include <stdio.h>
#include <rmanager/uif.h>

void printProcInfo(proc_t* p)
```

```
{
   /* Display info from a proc_t structure */
   printf(" Proc type: %s\n", rms_procTypeString(p->type));
   printf("    ElanId: %d\n", p->elanId);
   printf("    Status: %s\n", rms_procStatusString(p->status));
   printf("  Hostname: %s\n\n", p->name);
}


main(int argc, char** argv)
{

   int i = 0;
   int baseProc, topProc, nProcs;
   rrequest_t *resource;
   partition_t *partition;
   proc_t *proc;
   map_t *map;
   sysDefaults* def;


   /* Get default resource spec from the environment */
   resource = rms_defaultResourceRequest();
```

Having determined the default partition the program can override this with the partition named on the command line (if any). Note however that if the program is running in a shell with resources already allocated then it makes sense to target that partition, as the user's parallel applications will always be executed on that resource; in this case (indicated by a positive integer in the `rrequest.rid` field) the command line option will be ignored.

```
   /* Ignore args if resource is allocated to shell  *
    * otherwise override default partition with program args */
   if (argc > 1 && resource->rid < 0)
      strncpy(resource->partition, argv[1], NAME_SIZE);
```

For the case where no partition is specified a default partition name is read from the system defaults file.

```
   /* If no has been specified then read from defaults(4) */
   if(resource->partition[0] == 0) {
      defaults = rms_parseDefaultsFile("");
```

```
        strncpy(resource->partition, def->partition, NAME_SIZE);
    }
```

Having identified a target partition we can extract information about it with `rms_describe()`. In this case we scan the list of partition descriptions until the named partition is located, or until the end of the list is reached; we exit if the partition description cannot be found.

```
/* Get partition description for named partition */
while((partition = (partition_t*)rms_describe(RMS_PARTITION, i++)) != NULL)
   if(!strcmp(resource->partition, partition->name)) break;

/* `partition' is either NULL or pointer to a partition */
if (partition == NULL) {
   printf("Could not locate partition %s\n", resource->partition);
   exit(1);
}
```

The program extracts from the partition description a processor map that identifies the Elan Id's of the partition members. The map is a bit array, indexed by Elan Id, in which asserted bits indicate the group members. The program scans this map, between the upper and lower bounds identified from the partition description, and then uses `rms_describe()` to fetch a description of each member processor. Note that `rms_describe()` is passed the object type RMS_PROCBYELANID; this represents a list of processor descriptions that is ordered by Elan id, and differs from RMS_PROC in which the descriptions have an indeterminate ordering. Having fetched a processor description we can print the required information.

```
map = &partition->map;             /* map of partition members */
baseProc = partition->baseProc;    /* ElanID of first processor */
topProc = partition->topProc;      /* ElanId of last processor */
nProcs = partition->nProcs;        /* Number of processors */

printf("Partition %s has %d procs:\n\n", resource->partition, nProcs);

for(i=baseProc; i<=topProc; i++) {

   /* Bits set in map indicate ElanIds of partition members */
   if(MAP_ISSET(i, map)) {
```

```
         /* So get description of those processors */
         if((proc = (proc_t*) rms_describe(RMS_PROCBYELANID, i)) == NULL) {
            fprintf(stderr, "Cannot get processor description \n");
            exit(1);
         }

         printProcInfo(proc);
         }
      }
   }
```

In this case, we use the following simple display function.

```
void printProcInfo(proc_t* p)
{
   /* Display info from a proc_t structure */
   printf(" Proc type: %s\n", rms_procTypeString(p->type));
   printf("    ElanId: %d\n", p->elanId);
   printf("    Status: %s\n", rms_procStatusString(p->status));
   printf("  Hostname: %s\n\n", p->name);
}
```

Computing
Surface

CSN Communications Library for C

S1002–10M106.06

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

Using the C Communications Library 1

CSN Communication Routines

The CSN routines provide access to the Computing Surface Network, which provides a general point to point communications scheme. These routines must be explicitly referenced from the `libcsn` library, as shown later in this chapter.

CSN communications occur through *transports*; a transport is a bidirectional end point for communication. Each transport in the network has a unique address, which must be used by the sender of a message to identify the target of the communication. Individual programs can have many transports open simultaneously for the transmission and reception of messages. Facilities are provided (through the calls `csn_registername()`, `csn_lookupname()` and `csn_deregistername()`) to give meaningful names to transports, so that user code need not concern itself about the internal structure of network addresses.

Here is a full list of the CSN functions.

`csn_close()`	Close a CSN transport.
`csn_deregistername()`	Remove a name from a transport.
`csn_exit()`	Shut down network connection and exit process.
`csn_getId()`	Get transport address.
`csn_GET_NET()`	CSN address manipulation (macro).

`csn_GET_NODE()`	CSN address manipulation (macro).
`CSN_GET_TRANSPORT()`	CSN address manipulation (macro).
`csn_init()`	Set up the connection to the network.
`csn_lookupname()`	Find a transport from a textual name.
`CSN_MAKE_ID()`	CSN address manipulation (macro).
`csn_open()`	Open a new transport.
`csn_registername()`	Give a textual name to a transport.
`csn_rx()`	Receive a message.
`csn_rxnb()`	Queue a buffer for receiving a message.
`csn_statusString()`	Return textual status.
`csn_test()`	Test for completion of queued send/receive.
`csn_tx()`	Send a message.
`csn_txnb()`	Queue a message for transmission.

The CS-2 libcsn.a library includes two new routines providing information on the number of processors and the id of each processor.

`csn_nnodes()`	Number of processors.
`csn_node()`	Processor id.

The CS-2 `libcsn.a` library includes a number of support routines that were previously part of `libcs.a`; `libcs.a` itself is no longer needed.

`cs_abort()`	Terminate task.
`cs_getinfo()`	Get processor information.

Functions for Starting-up and Shutting-down

There are various functions which are useful when starting up a program and when closing it down. These include functions for giving names to transport addresses, so that other processes can communicate with them, and functions for finding out which process you are.

`cs_abort()`	Terminate task.
`csn_close()`	Close a CSN transport.
`csn_deregistername()`	Remove name from a transport.
`csn_exit()`	Shut down network connection and exit process.
`cs_getinfo()`	Get processor information.
`csn_init()`	Set up the connection to the network.
`csn_lookupname()`	Find a transport from a textual name.
`csn_nnodes()`	Number of processors.
`csn_node()`	Processor id.
`csn_open()`	Open a new transport.
`csn_registername()`	Give a textual name to a transport.

Warning – csn_init() must be called before using other CSN routines when running applications on the CS–2.

Functions for Inter-Processor Communication

The functions for performing inter-processor communication can be split into two classes: those that do not complete until the communication has completed, and those which return immediately, allowing the program to continue to execute while the communication takes place. The operation of the functions that suspend or block the user process are easier to understand; these functions are `cs-n_tx()`, and `csn_rx()`.

Note that all of the message sizes on receive and transmit are given in bytes.

`csn_rx()`	Receive a message.
`csn_tx()`	Send a message.

Functions for Non-blocking I/O

As well as the blocking CSN functions there are corresponding functions `csn_txnb()` and `csn_rxnb()` that can be used to start communications while allowing the user program to continue to execute. Using these functions it is possible to queue up many buffers into which receives will occur when messages are sent, thus insulating the sender from delays in the receiver, or queue many buffers to be sent as soon as a receiver is willing to accept them.

As soon as the sender has many buffers queued up for transmission or reception, one needs a way of testing whether a buffer has been sent so that we may reuse or destroy the buffer. This functionality is provided by `csn_test()`.

`csn_rxnb()`	Queue a buffer for receiving a message.
`csn_test()`	Test for completion of queued send/receive.
`csn_txnb()`	Queue a message for transmission.

Header Files

Two header files contain function prototypes and macro definitions for use with this library. The files are in the directory `/opt/MEIKOcs2/include/csn` and are called `csn.h` and `names.h`.

You should specify to your compiler the search path for these header files by using the command line option `-I` with the argument `/opt/MEIKOcs2/include`.

Library Files

All CSN libraries are stored in the directory `/opt/MEIKOcs2/lib`. Programs that use the CSN routines must be linked with the following command line options:

```
-L/opt/MEIKOcs2/lib -lcsn -lew -lelan
```

Tracing

To use the version of the CSN library that produces ParaGraph compatible trace files you precede the `-lcsn` in the above line by `-lcsn_pt`. Your attention is drawn to the following two sections which describe environment variables that are applicable to tracing, and also the tracing functions.

Debugging

There is also a debugging version of the library which attempts to provide more security and better error behaviour than the standard library — although it will also be slower. This library is available by specifying `-lcsn_dbg` in place of the standard version.

Environment Variables

The following environment variables are used by this library. Many are inherited from `libew` — the low level Elan Widget library.

`LIBCSN_TRACEFILE`	For use with `libcsn_pt` only, this variable specifies the name of the trace file to use; each node outputs to `$LIBCSN_TRACEFILE`.*nodeno*. Default name is `LIBCSN_TRACE`.*nodeno*.
`LIBCSN_TRACEBUF`	For use with `libcsn_pt` only, this variable specifies the number of events to allow in the trace buffer.
`LIBEW_WAITYPE`	Specifies how the low level Elan widget library (`libew`) routines wait for Elan events; either POLL or WAIT, default is to POLL.
`LIBEW_DMATYPE`	Specifies the type of DMA transfer used by the low level Elan widget library (`libew`). Either NORMAL or SECURE.
`LIBEW_DMACOUNT`	Specifies the permitted retry count for DMA transfers. Default is 1.
`LIBEW_RSYS_ENABLE`	Enables the remote system call server; when enabled `stdin`, `stdout`, and `stderr` are routed through the host process. May be either 0 (disabled) or 1 (enabled), default is 1.
`LIBEW_RSYS_BUFSIZE`	The buffer size used by the remote system call server. Default is 8192 bytes.

`LIBEW_RSYS_SERVER`	Virtual process ID of the processor that will run the system call server.
`LIBEW_CORE`	Enables core dump on exception. Values may be 1 (enabled) or 0 (disabled). By default core dumping is disabled.
`LIBEW_TRACE`	Enables a trace dump on exception. Values may be 1 (enabled) or 0 (disabled). By default trace dumping is disabled.

Program Tracing

Both ParaGraph and Alog/Upshot are supported for program tracing.

ParaGraph

Three functions in the low level Elan Widget library (`libew`) are applicable to program tracing — these are `ew_ptraceStart()`, `ew_ptraceStop()`, and `ew_ptraceFlush()`. None of these take arguments and none return values to the caller.

Programs that are traced must be linked with `libcsn_pt` as described in an earlier section. The resulting trace file may be analysed with ParaGraph.

`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

Full documentation for the tracing functions is included in the *Elan Widget Library* reference manual.

Alog/Upshot

As an alternative to ParaGraph the event/state display tool `upshot` is also supported. To use this you need to instrument your code with trace points. Details may be found in `/opt/MEIKOcs2/upshot/README-MEIKO`.

Reference Manual 2

This chapter includes detailed descriptions of each function in the CSN library.

cs_abort() — **Parallel C communications routine**

Synopsis

```
#include <cs.h>
void cs_abort(char *message, int exitCode);
```

Description

`cs_abort()` prints the given message to standard error and then causes an exception on the calling process. It will never return. No flushing of output buffers is performed, so this function should be used with caution.

See Also

`csn_exit()`.

cs_getinfo() | **Parallel C communications routine**

Synopsis

```
include <cstools/cstools.h>
void cs_getinfo(int *nProcs,int *procId,int *localId);
```

Description

`cs_getinfo()` returns the number of processors involved in the program (`nprocs`), the identity of the local processor (`processorId` = 0...(`nProcs`-1)), and the identity of this process on this processor (currently always 0). The result will be 0 for success, and less than zero in the case of an error.

csn_close() — Close a CSN transport

Synopsis

```
include <csn/csn.h>
int csn_close (Transport t);
```

Description

`csn_close()` closes the transport `t`. It fails (and the transport remains open) if there are outstanding sends or receives queued on the transport, or if the results of completed non-blocking communications have not been collected by `csn_test()`.

Return codes are as follows:

`CSN_OK`	Transport successfully closed.
`CSN_ENOTREADY`	Transport could not be closed due to outstanding communications in progress.

See Also

`csn_test()`.

csn_deregistername() CSN (Named Transport)

Synopsis

```
include <csn/names.h>
int csn_deregistername (Transport tpt);
```

Description

`csn_deregistername()` removes a naming association created by the function, `csn_registername()`, and must be called before the transport can be renamed. It returns `CSN_EBADREQ` if the transport has not been registered or looked-up, and `CSN_OK` on success.

See Also

`csn_lookupname()`, `csn_registername()`.

csn_exit() — **Shut down CSN connection and exit process**

Synopsis

```
include <csn/csn.h>
void csn_exit(int return_code);
```

Description

This function shuts down the connection to the CSN network, which causes any open transports to be closed. The process then terminates, returning the exit status `return_code`.

This function should be used in preference to `exit()` when running parallel programs using the CSN.

To kill a parallel application, all processes should globally synchronise. Each process then calls `csn_exit()`, but note that the process does not exit until all other processes have also called this function.

In current releases of this library, all outputs to the standard output device (`stdout`) are routed through a single process (to ensure they are correctly line buffered). You must ensure that all output is complete before the IO process terminates.

CSN_GET_NET() — Extract network number from CSN address

Synopsis

```
include <csn/csn.h>
CSN_GET NET(id)
```

Description

This macro is defined in the header file, <csn/csn.h>. It returns the network number from the CSN address, id, that is passed as an argument.

CSN addresses (as returned by csn_lookupname() and other CSN functions) are structures that consist of three fields: the network number, the node number, and the transport number.

See Also

CSN_GET_NODE(), CSN_GET_TRANSPORT(), CSN_MAKE_ID().

CSN_GET_NODE() — Extract node number from CSN address

Synopsis

```
include <csn/csn.h>
CSN_GET_NODE(id)
```

Description

This macro is defined in the header file, `<csn/csn.h>`. It returns the node number from the CSN address, `id`, that is passed as an argument.

CSN addresses (as returned by `csn_lookupname()` and other CSN functions) are structures that consist of three fields: the network number, the node number, and the transport number.

See Also

CSN_GET_NET(), CSN_GET_TRANSPORT(), CSN_MAKE_ID().

CSN_GET_TRANSPORT() Get transport number from CSN address

Synopsis

```
include <csn/csn.h>
CSN_GET_TRANSPORT(id)
```

Description

This macro is defined in the header file, `<csn/csn.h>`. It returns the transport number from the CSN address, `id`, that is passed as an argument. This only makes sense if the relevant transport is local to the processor calling the function.

CSN addresses (as returned by `csn_lookupname()` and other CSN functions) are structures that consist of three fields: the network number, the node number, and the transport number.

See Also

`CSN_GET_NET()`, `CSN_GET_NODE()`, `CSN_MAKE_ID()`.

csn_getId() — Get the CSN address of a transport

Synopsis

```
include <csn/csn.h>
netid_t csn_getId(transport t);
```

Description

This function gets the CSN address of the local transport, `t`.

See Also

`CSN_GET_NET()`, `CSN_GET_NODE()`, `CSN_GET_TRANSPORT()`, `CSN_MAKE_ID()`.

csn_init() — Initialise the CSN

Synopsis

```
void csn_init();
```

Description

This function sets up the network connection between the current process and the CSN network — it must be the first function that is called by the process.

Before the CSN can be used, `csn_init()` must be called to perform any system initialisation which may be required. After calling `csn_init()`, a program will normally create a set of Transports (using `csn_open()`), give each of the transports a meaningful name (using `csn_registername()`), and then (using `csn_lookupname()`) discover the addresses of the transports to which it intends to transmit. It is normal for all programs to create their transports before looking up any others to avoid potential deadlocks where two programs are each waiting for the other to create and register a transport.

csn_lookupname() **CSN (Named Transport)**

Synopsis

```
include <csn/names.h>
int csn_lookupname (netid_t *peer, char *name,
                    int block);
```

Description

`csn_lookupname()` looks for the specified `name` in the global name space. If `block` is set the function will wait until the name has been declared, otherwise it will fail and return `CSN_ENOTREADY`.

This function returns `CSN_OK` on success, and sets `*p` to be the network `id` associated with that name.

See Also

`csn_registername()`, `csn_deregistername()`.

CSN_MAKE_ID() — Assemble CSN address

Synopsis

```
include <csn/csn.h>
CSN_MAKE_ID(net, node, transport);
```

Description

This macro is defined in the header file, `<csn/csn.h>`. It assembles a CSN address from a network number, `net`, a node number, `node`, and a transport number, `transport`.

CSN addresses (as returned by `csn_lookupname()` and other CSN functions), are structures that consist of three fields: the network number, the node number, and the transport number.

Warning – In the current implementation `net` must be 0.

Warning – Manipulation of the internal structure of network addresses is not recommended.

See Also

`CSN_GET_NET()`, `CSN_GET_NODE()`, `CSN_GET_TRANSPORT()`.

csn_nnodes() — Number of Processors

Synopsis

```
include <csn/csn.h>
int csn_nnodes();
```

Description

Parallel programs are run one process per processor on the CS–2. This function returns the number of processors executing this application.

See Also

`csn_node()`

csn_node() — Processor Id

Synopsis

```
include <csn/csn.h>
int csn_node();
```

Description

Parallel programs are run one process per processor on CS–2. This function returns the ID of the processor executing this process. IDs will lie in the range 0 to `nodes-1`, where `nodes` is returned by `csn_nnodes()`.

See Also

`csn_nnodes()`.

csn_open() — Open a CSN transport

Synopsis

```
include <csn/csn.h>
int csn_open (int index, Transport *t);
```

Description

`csn_open()` creates a new CSN transport; if successful it returns it in `*t`. `index` may either be set to the desired transport number, or to -1 indicating that any free transport number may be used.

Return values from `csn_open()` are as follows:

`CSN_OK`	New transport successfully created and returned in `*t`.
`CSN_ERANGE`	Requested transport index out of range.
`CSN_EALLOC`	Requested transport index already allocated. If a specific transport index was not requested, this result means that all transports are allocated.
`CSN_ENOHEAP`	No heap space left to build transport.

csn_registername() **CSN (Named Transport)**

Synopsis

```
include <csn/names.h>
int csn_registername (Transport tpt, char *name);
```

Description

`csn_registername()` declares the specified `name` to be associated with the CSN address of the transport `t`. It may return `CSN_EBADREQ` if the transport already has a naming scheme associated with it (that is, it hasn't been deregistered before changing it's name), `CSN_ENOHEAP` if a descriptor cannot be created in memory, or `CSN_EALLOC` if the name is already declared in the global name space. `CSN_OK` is returned on success.

See Also

`csn_lookupname()`, `csn_deregistername()`.

csn_rx() — Receive a message from a CSN transport

Synopsis

```
include <csn/csn.h>
int csn_rx (Transport t, netid_t *fromId_p,
char *data, int nob);
```

Description

`csn_rx()` queues the message buffer data for receiving up to `nob` bytes on transport `t`. The contents of the message buffer may be updated by the CSN at any time until the communication completes.

`csn_rx()` blocks until a message has been received. If `fromId_p` is non-NULL, the address of the source transport is passed back in it. The non-blocking version, `csn_rxnb()`, returns immediately, and completion of the receive it initiates must be determined by calling `csn_test()`.

Return values for `csn_rx()` are as follows:

`n >= 0`	This result indicates that a message of size `n` bytes was received successfully.
`CSN_EABORT`	A call to `csn_cancel()` on this transport caused this communication to abort.

See Also

`csn_tx()`, `csn_test()`.

csn_rxnb()

Receive a message from a CSN transport

Synopsis

```
include <csn/csn.h>
int csn_rxnb (Transport t, char *data, int nob);
```

Description

`csn_rxnb()` queues the message buffer data for receiving up to `nob` bytes on transport `t`. The contents of the message buffer may be updated by the CSN at any time until the communication completes.

`csn_rx()` blocks until a message has been received. If `fromId_p` is non-NULL, the address of the source transport is passed back in it. The non-blocking version, `csn_rxnb()`, returns immediately, and completion of the receive it initiates must be determined by calling `csn_test()`.

Return values for `csn_rxnb()` are as follows:

CSN_OK	The message buffer has been queued successfully on transport `t`. The contents of the message buffer should not be inspected or altered until a call to `csn_test()` determines that this communication has completed, when one of the above `csn_rx()` results will be returned.
CSN_ENOHEAP	The message buffer was not queued for receiving, due to lack of heap space.

See Also

`csn_rx()`, `csn_tx()`, `csn_test()`.

csn_statusString() — Return CSN error string

Synopsis

```
include <csn/csn.h>
char* csn_statusString(int status);
```

Description

This function returns a pointer to a static string containing a textual version of the CSN error code status.

csn_test() — **Test for completion of non-blocking CSN communications**

Synopsis

```
include <csn/csn.h>
int csn_test (Transport t, int flags, long timeOut,
        netid_t *id p, char **data_p, int *status_p);
```

Description

`csn_test()` allows a process to detect the completion of communications initiated by `csn_txnb()` and/or `csn_rxnb()` on transport `t`. The `flags`, `id_p`, and `data_p` parameters determine the class of completed communications to wait for (for example, any send or receive, any send to a particular transport address, any receive of data into a particular message buffer).

Setting `flags` to 0 causes `csn_test()` to wait for any completed non-blocking communication, subject to the restrictions imposed by the other parameters. The test may be restricted to communications initiated by `csn_txnb()` by setting `flags` to `CSN_TXREADY`, and to communications initiated by `csn_-rxnb()` by setting `flags` to `CSN_RXREADY`. OR'ing these flags has the same effect as passing 0. Passing any other value into `flags` is an error.

Negative values of `timeOut` cause `csn_test()` to block indefinitely until a specified communication completes, otherwise it specifies a number of microseconds to wait before returning failure.

Setting `id_p` to NULL or setting `*id_p` to `CSN_NULL_ID` will cause `csn_test()` to ignore the source/destination address when it looks for a completed communication. Otherwise the test is restricted to messages sent to or received from `*id_p`. Note that passing an impossible address in `*id_p` causes the test to block until the time-out expires.

Setting `data_p` to NULL or setting `*data_p` to NULL causes `csn_test()` to ignore the message buffer when it looks for a completed communication. Otherwise the test is restricted to messages sent from or received into `*data_p`. Note that passing an impossible message buffer in `*data_p` causes the test to block until the time-out expires.

`csn_test()` must be used to free-up the memory used by non-blocking communications.

Possible results of `csn_test()` are as follows:

CSN_TXREADY	A communication initiated by csn_txnb() completed or cancelled.
CSN_RXREADY	A communication initiated by csn_rxnb() completed or cancelled.
0	No specified communications completed and at least timeOut micro-seconds had elapsed since calling csn_test().
CSN_EBADREQ	Illegal value for flags.
CSN_EABORT	Transport t was closed while csn_test() was blocked. Note that a transport may only be closed after all outstanding communications on it have completed.

On successfully finding a completed communication, if id_p is non-NULL, *id_p contains the source/destination transport address of the completed communication. If data_p is non-NULL, *data_p contains the message buffer of the completed communication. Also if status_p is non-NULL, *status_p contains the return status of the completed communication. In the case of a cancelled communication status_p is set to CSN_EABORT (and csn_test() returns either CSN_TXREADY or CSN_RXREADY).

See Also

csn_txnb(), csn_rxnb(), csn_close().

csn_tx() — Send a message via CSN

Synopsis

```
include <csn/csn.h>
int csn_tx (Transport t, int flags, netid_t toId,
            char *data, int nob);
```

`csn_tx()` queues the message buffer `data` for transmission of `nob` bytes to the transport at address `toId`. The `flags` parameter is currently unused, and should always be set to 0. The contents of the message buffer should not be altered until the communication completes.

`csn_tx()` blocks until the communication is complete. The non-blocking version, `csn_txnb()`, returns immediately, and completion of the communication it initiated must be determined by calling `csn_test()`.

`toId` may be set to CSN_NULL_ID, targeting the message at a notional transport which is always ready to receive messages of arbitrary size.

Return values for `csn_tx()` are as follows:

`n == nob`	This result indicates that the communication completed successfully.
`CSN_ENOSPACE`	No space to buffer this message at the destination transport. When many processes all send messages to a single destination transport, the destination may not have enough space to buffer all the pending messages and may cause one or more of the source transports to attempt re-transmission. This result is returned if re-transmission has not been successful after the source transport's re-transmission timeout has expired.
`CSN_ENODEST`	No transport exists with address `toId`. This result is returned when the net ID or node ID components of `toId` refer to non-existent network or node numbers, when the destination transport is refusing messages from this source or when the destination transport does not exist and the source transport's re-transmission timeout has expired.
`CSN_EOVERRUN`	Message too large for the receiving process's buffer.

See Also

`csn_txnb()`, `csn_open()`, `csn_rx()`, `csn_test()`.

csn_txnb() — **Send a message via CSN**

Synopsis

```
include <csn/csn.h>
int csn_txnb (Transport t, int flags, netid_t toId,
              char *data, int nob);
```

Description

`csn_txnb()` queues the message buffer `data` for transmission of `nob` bytes to the transport at address `toId`. The `flags` parameter is reserved for future use and should always be set to 0. The contents of the message buffer should not be altered until the communication completes.

`csn_tx()` blocks until the communication is complete. The non-blocking version, `csn_txnb()`, returns immediately, and completion of the communication it initiated must be determined by calling `csn_test()`.

`toId` may be set to CSN_NULL_ID, targeting the message at a notional transport which is always ready to receive messages of arbitrary size.

Return values for `csn_txnb()` are as follows:

`CSN_OK`	The message buffer has been queued successfully on transport `t`. The contents of the message buffer should not be altered until a call to `csn_test()` determines that this communication has completed, when one of the above `csn_tx()` results will be returned.
`CSN_ENOHEAP`	The message buffer was not queued for transmission due to lack of heap space.

See Also

`csn_tx()`, `csn_open()`, `csn_rx()`, `csn_test()`, `csn_cancel()`.

Tutorial Examples 3

Overview

This chapter includes a number of examples showing how to use the CSN communication library. It discusses the use of *transports* and the choice of *blocking* versus *non-blocking* communications.

Compilation and Execution

All the examples in this chapter can be compiled with the following command line:

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

The programs are executed with `prun(1)` and will use command lines like that shown below. Note that *number* is the number of processors required, *partition* is the name of the partition that you will use, and *myprogram* is the name of the program.

```
user@cs2: prun -nnumber -ppartition myprogram
```

Full information about `prun(1)` command may be obtained from the reference manual page.

Two Communicating Processes

The following example defines two processes that use a single blocking CSN communication for synchronisation.

This example introduces *transports* and shows how they are used for a simple blocking communication between two processes.

Transports

A *transport* is a connection from a process to the Computing Surface Network. There is no limit on the number of transports that a process can use, so it is normal to create a transport that is dedicated to specific classes of communication, or to specific senders. In this example each process uses just one transport.

Each transport has an associated address, or *net id*. To send data to a remote transport the sender must first determine the address of the destination transport. To do this the receiver registers a name for its transport with `csn_registername()`; the sending process determines the net id of this transport by looking-up the name with `csn_lookupname()`.

A useful analogy that helps explain the use of transports is to compare the CSN with a telephone network. Using this analogy people represent processes, the telephone lines represent transports, and the telephone exchange represent the CSN network. Each person's telephone line allows them to communicate with any other (and there may be many lines each dedicated to a specific type of communication) but to make a call the person must first determine the receiver's number by looking up a name in the directory.

Blocking Communications

The CSN supports two types of communication: blocking and non-blocking. In this example we consider blocking communications — the communication between sender and receiver is delayed until both processes have called their communication function. It is this implicit sychronisation that is exploited in this example.

Program Description

This example is a parallel implementation of the standard Hello World program found in C programming tutorials. In this example there are two processes; one writes `Hello` to the screen, the other writes `World`. A simple blocked communication is used to synchronise the processes.

The program begins with initialisation code that is common to both processes. `csn_init()` is used to initialise the network, `cs_getinfo()` identifies each process's virtual process number and the total number of processes in the application, and `csn_open()` creates a transport.

The process with virtual process number 0 will be the sender of the blocked communication. The sender determines the network address of the recipient's transport by looking-up the transport's name with `csn_lookupname()` (the third argument is non-zero indicating that `csn_lookupname()` should wait for the other process to register its transport's name if it has not already done so). Our sending process then writes its string to the screen[1], and uses `csn_tx()` to send a simple integer data item. At this point the sender will block until the recipient is ready to take the data.

Process 1 is the recipient of the communication. The recipient must register a name for its transport with `csn_registername()` so that it is visible to our sender. The recipient waits until it receives a communication from the sender (using `csn_rx()`), and then writes its part of the string to the screen.

Both process finish by calling `csn_exit()`.

Program Listing

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


main( argc, argv )
int argc;
```

1. Because the Hello string is not terminated by a line feed it is necessary to use fflush() to force the string onto the screen; otherwise it would not be written until the process finishes.

```
char* argv[];
{
   Transport transport;
   netid_t   networkid;
   int flag = 1;
   int status;
   int nprocs, me, dummy;
   int nob;

   csn_init();

   cs_getinfo(&nprocs, &me, &dummy);

   if(nprocs != 2) {
      /* Only process 0 prints the error message */
      if(me == 0) fprintf(stderr, "Need two processors for this example\n");
      exit(1);
   }

   status = csn_open( CSN_NULL_ID, &transport );
   if( status != CSN_OK ) {
      fprintf(stderr, "Process %d: Cannot open transport\n", me);
      exit(1);
   }


   if( me == 0 ) {
      /* Process 0 will be the sender */

      status = csn_lookupname( &networkid, "Receiver", 1 );
      if( status != CSN_OK ) {
         fprintf(stderr,"Process %d: Cannot lookup transport\n", me);
         exit(1);
      }

      printf("Hello "); fflush(stdout);

      /* Awake process 1 by sending a token integer */
      nob = csn_tx( transport, 0, networkid, (char*)&flag, sizeof(flag) );
      if( nob != sizeof(flag) ) {
         fprintf(stderr, "Process %d: Failed to transmit\n", me);
         exit(1);
      }
   }
   else {
```

```
        /* Process 1 will be the receiver */

        status = csn_registername( transport, "Receiver" );
        if( status != CSN_OK ) {
           fprintf(stderr, "Process %d: Cannot register transport\n", me);
           exit(1);
        }

        /* Wait for synchronisation from process 0 */
        nob = csn_rx( transport, NULL, (char*)&flag, sizeof(flag) );
        if( nob < 0 ) {
         fprintf(stderr, "Process %d: Failed to receive\n", me);
           exit(1);
        }

        printf("world\n");
    }
    csn_exit(0);
}
```

Bidirectional Communications

The following example is suitable for use with 2 or more processors. It defines a master process and a number of slaves; the slaves send data to a master which broadcasts a result back.

The example shows how to use transports for bidirectional communications, and also introduces a style of programming that is suitable for a variable number of target processors.

Transports

In this example each process creates just one transport that is used for both incoming and outgoing communications. The processes could use a separate transport for each direction, or indeed dedicate a transport to each pair or processes.

To select the best use of transports for your application you should consider the message receiving functions `csn_rx()` and `csn_rxnb()`. These can both identify the network address of the sending transport (although this facility is not used in this example). By using a transport for a specific type of message the recipient of a message can infer a context for the data that it has received.

Program Description

All the processes begin by calling `csn_init()` to initialise the network, and follow this with a call to `cs_getinfo()` to get their virtual process number and the number of processes in the application. Each process then opens a single transport which will be used for both outgoing and incoming communications.

Each process registers it's own transport's name, and then looks-up the network address for all the other transports. Note that each transport's name is derived from the owning process's virtual process number, and that the network addresses are stored in an array that is indexed by virtual process number. This strategy keeps the program code compact, and allows the number of target processors to be specified at execution time.

At this point the program splits into the code for our master, and code for the slaves. The master receives from each slave data that is simply added and then broadcast back to all the slaves.

Program Listing

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

#define MAXPROCS 10
#define NAMELEN 20

main( argc, argv )
int argc;
char* argv[];
{
   Transport transport;
   netid_t networkid[MAXPROCS];
   int nprocs, me, dummy;
   int status, nob;
   int i;
   int result=0;
   char name[NAMELEN];

   struct {
      int data;
   } packet;


   /* Initialise */
   csn_init();

   /* Get my process id & number of procs */
   cs_getinfo(&nprocs, &me, &dummy);

   if(nprocs > MAXPROCS) {
      /* Only process 0 prints this error */
      if(me==0) fprintf(stderr, "Less that %d processes expected\n", MAXPROCS);
      exit(1);
   }

   /* Open my transport */
   status = csn_open( CSN_NULL_ID, &transport );
   if( status != CSN_OK ) {
      fprintf(stderr, "Process %d: Cannot open transport\n", me);
      exit(1);
   }
```

```
/* Register my transport */
sprintf(name, "Proc%d", me);
status = csn_registername(transport, name);
if( status != CSN_OK ) {
   fprintf(stderr, "Process %d: Cannot register transport\n", me);
   exit(1);
}


/* Lookup all the other transports */
for(i=0; i<nprocs; i++) {

   if(i==me) continue; /* Don't lookup my own tranport */

   sprintf(name, "Proc%d", i);
   status = csn_lookupname( &networkid[i], name, 1 );
   if( status != CSN_OK ) {
      fprintf(stderr,"Process %d: Cannot lookup transport\n", me);
      exit(1);
   }
}


/* Process 0 is the master */
if(me==0) {

   /* Get data from all the workers */
   for( i=1 ; i<nprocs; i++) {
      nob=csn_rx( transport, NULL, (char*)&packet, sizeof(packet));
      if( nob != sizeof(packet)) {
         fprintf(stderr, "Process %d: Failed to receive\n", me);
         exit(1);
      }

      printf("Master receives data\n");
      result += packet.data;
   }

   /* Now broadcast a result back to all the processes */
   packet.data = result;

   for(i=1; i<nprocs; i++) {
      nob = csn_tx( transport, 0, networkid[i], (char*)&packet, sizeof(packet));
      if( nob != sizeof(packet)) {
         fprintf(stderr, "Process %d: Failed to transmit\n", me);
```

```
            exit(1);
         }
      }
   }
   else {

      /* I am a worker */

      /* Initialise the data packet with some data */
      packet.data = me;

      /* Send my data to the master (process 0) */
      nob = csn_tx( transport, 0, networkid[0], (char*)&packet, sizeof(packet) );
      if( nob != sizeof(packet)) {
         fprintf(stderr, "Process %d: Failed to transmit\n", me);
         exit(1);
      }

      /* Get the result back from the master */
      nob=csn_rx( transport, NULL, (char*)&packet, sizeof(packet));
      if( nob != sizeof(packet)) {
         fprintf(stderr, "Process %d: Failed to receive\n", me);
         exit(1);
      }

      /* Display the result */
      printf("Slave process %d: received %d from master\n", me, packet.data);
   }
   csn_exit(0);
}
```

Non-Blocking Communications

The following example runs on 2 processors. It defines a Producer process that wishes to send a large number of messages to a Consumer process.

The example simulates the case where a process wishes to send a large number of non-blocking messages to a receiver process. The receiver does not know in advance how many messages will be sent, nor can the producer assume that the consumer has sufficient heap space to receive them all. The producer and consumer therefore periodically synchronise with a blocking communication so that the number of non-blocking communications is agreed before they are sent.

Non-Blocking Communications

This form of communication between processes does not require the sender and recipient to synchronise, and is therefore more appropriate to time critical applications where processes cannot be allowed to idle.

Non-blocking communications allow a sender to initiate a transmission and to continue immediately without waiting for the communication to complete. Similarly a receiver can initiate a receive without waiting for the message to arrive.

Non-blocking sends are initiated by `csn_txnb()`. The data identified by this function will be transferred from the process's address space at some indeterminate time in the future. To test the status of the transfer the program *must* use `csn_test()` — only when the transfer has completed may the data buffer be modified or destroyed.

Non-blocking receives are initiated by `csn_rxnb()`. This function identifies a data buffer that can receive the incoming data. To test the status of the transfer the program *must* use `csn_test()` — only when the transfer has completed may the data buffer be modified or destroyed.

Program Description

Following the initialisation of the CSN and of each process's transports the program defines two processes: process 0 is a producer, and process 1 a consumer.

The producer sends a blocking communication to the consumer to agree a number of non-blocking communications that may follow. If the consumer accepts, the agreed number of non-blocking sends are initiated with `csn_txnb()`.

The producer can, without waiting for the communications to complete, continue with other meaningful work, until it is ready to use `csn_test()` to confirm that the transfers completed successfully.

The consumer awaits the blocking communications from the producer by making the required number of calls to `csn_rxnb()`. Each call identifies a unique data buffer for each of the incoming communications — these buffers must not be modified or destroyed until the communications are complete. The receiver can test the status of the communications at any time by calling `csn_test()`.

Program Listing

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

#define MAXMESSAGES 50
#define NAMELEN 20
#define STOP -1
#define REQSIZE 10
#define TIMEOUT 1000000

main( argc, argv )
int argc;
char* argv[];
{
   Transport transport;
   netid_t networkid;
   int nprocs, me, dummy;
   int status, nob;
   int data = 99;
   int i;
   int *rxbuffer;
   int messages, requestsize;
   char name[NAMELEN];

   /* Initialise */
   csn_init();

   /* Get my process id & number of procs */
   cs_getinfo(&nprocs, &me, &dummy);

   if(nprocs != 2) {
      /* Only process 0 prints this error */
      if(me==0) fprintf(stderr, "This example requires 2 processes\n");
      exit(1);
   }

   /* Open my transport */
   status = csn_open( CSN_NULL_ID, &transport );
   if( status != CSN_OK ) {
      fprintf(stderr, "Process %d: Cannot open transport\n", me);
      exit(1);
   }
```

```
   /* Register my transport */
   sprintf(name, "Proc%d", me);
   status = csn_registername(transport, name);
   if( status != CSN_OK ) {
      fprintf(stderr, "Process %d: Cannot register transport\n", me);
      exit(1);
   }


   /* Lookup my partner's transport */
   sprintf(name, "Proc%d", (me==0) ? 1 : 0);
   status = csn_lookupname( &networkid, name, 1 );
   if( status != CSN_OK ) {
      fprintf(stderr,"Process %d: Cannot lookup transport\n", me);
      exit(1);
   }


   if(me==0) {

      /* Process 0 is the producer */

      messages = MAXMESSAGES;
      while(messages > 0) {

         /* request a batch of buffers ... */
         requestsize = ((messages > REQSIZE) ? REQSIZE : messages);
         messages -= requestsize;
         printf("Producer requests %d buffers\n", requestsize);

         /* ... with a blocking communication */
         nob = csn_tx(transport, 0, networkid, (char*)&requestsize, sizeof(requestsize));
         if( nob != sizeof(requestsize)) {
            fprintf(stderr, "Process %d: Failed blocking transmit\n", me);
            exit(1);
         }

         /* Send a batch of messages ... */
         for(i=0; i<requestsize; i++) {

            printf("Producer sets-up non-blocking send\n");

            /* ... with a non-blocking communication */
            status = csn_txnb(transport, 0, networkid, (char*)&data, sizeof(data));
            if( status != CSN_OK ) {
```

```
                fprintf(stderr, "Producer: Failed non-blocking transmit\n");
                exit(1);
            }
        }

        /* Do some work here, if we want to */
        printf("Producer doing some other work\n");


        /* test for completion of non-blocking transmits */
        /* and also free-up internal CSN buffers */
        for(i=0; i<requestsize; i++) {
            status = csn_test(transport, CSN_TXREADY, TIMEOUT, NULL, NULL, NULL);
            if(status != CSN_TXREADY) {
                fprintf(stderr, "Producer: Non-blocking timeout or failure\n");
                exit(1);
            }
        }
        printf("Producer reports non-blocking sends are complete\n");
    }

    /* No more messages so request consumer to stop */
    requestsize = STOP; /* Send stop flag */
    printf("Producer requests consumer to STOP\n");

    nob = csn_tx(transport, 0, networkid, (char*)&requestsize, sizeof(requestsize));
    if( nob != sizeof(requestsize)) {
        fprintf(stderr, "Process %d: Failed blocking transmit\n", me);
        exit(1);
    }
}
else {
    /* Process 1 is the consumer */

    while(1) { /* Repeat forever */

        /* Get message count from producer */
        nob=csn_rx( transport, NULL, (char*)&requestsize, sizeof(requestsize));
        if( nob != sizeof(requestsize)) {
            fprintf(stderr, "Process %d: Failed to receive\n", me);
            exit(1);
        }

        /* Is this a request to stop? */
        if(requestsize == STOP) {
```

```
            printf("Consumer stopped by producer\n");
            break;
         }

         /* Allocate requested number of buffers */
         printf("Consumer receives request for %d buffers\n", requestsize);

         /* Allocate buffer space */
         if((rxbuffer = (int*) malloc(requestsize*sizeof(data)))==NULL) {
            fprintf(stderr, "Consumer connot allocate buffer space\n");
            csn_exit(1);
         }

         /* Receive a batch of messages - non-blocking receive */
         for(i=0; i<requestsize; i++) {
            status=csn_rxnb( transport, (char*)&rxbuffer[i], sizeof(data));
            if( status != CSN_OK ) {
               fprintf(stderr, "Consumer: Failed non-blocking receive\n");
               exit(1);
            }
            printf("Consumer sets-up non-blocking receive\n");
         }

         /* We could do some work here, if we want to */
         printf("Consumer doing some other work\n");

         /* test for completion of non-blocking transmits */
         /* and also free-up internal CSN buffers */
         for(i=0; i<requestsize; i++) {
            status = csn_test(transport, CSN_RXREADY, TIMEOUT, NULL, NULL, NULL);
            if(status != CSN_RXREADY) {
               fprintf(stderr, "Consumer: Non-blocking timeout or failure\n");
               exit(1);
            }
         }
         printf("Consumer reports non-blocking receives are complete\n");
         printf("Consumer frees buffer space\n");

         free(rxbuffer);

      } /* while loop */
   }
   csn_exit(0);
}
```

Error Messages 4

Message Format

The functions in the CSN library (`libcsn`) are built upon the functions in the Elan Widget library. Errors within `libcsn` are reported via the Widget library exception handler; this writes diagnostic messages to the standard error device and kills the application.

The format of `libcsn` messages is:

```
CSN EXCEPTION @ process : error_code (error_text)
Additional information: error message string
```

The *error message strings* are described later in this chapter. The *process* is the virtual process number of the process that detected the error; if the exception occurs before the process has attached to the network (i.e. before `csn_init()` is called) then this is shown as `----`. The *error code* (and its textual equivalent the *error text*) are one of:

Error Code	Error Text
2000	`Ok`
2001	`No Destination`
2002	`Buffer Overflow`
2003	`No space at destination`

Error Code	Error Text
2004	`No heap`
2005	`Bad request`
2006	`Already allocated`
2007	`Out of range`
2008	`Aborted`
2009	`Not ready`
2010	`Interrupted`
2011	`Bad Address`

Widget Library Exceptions

Functions in `libcsn` are implemented on functions in the Elan Widget library. When an exception occurs within a Widget library function this is handled by the Widget library's own exception handler. The Widget library handler is similar to that used by `libcsn` but produces errors in the form:

`EW_EXCEPTION @` *process* `:` *error_code* `(`*error_text*`)`
error message string

These exceptions are fully described in *The Elan Widget Library*, Meiko document number S1002–10M104.

Note for Fortran Programmers

All errors apply to both C and Fortran implementations unless the description specifies a specific language. Often the error message repeats the parameters that were passed to the failed call; these will be the parameters that were passed to the underlying C implementation of the function, and may not be identical to those passed to the Fortran binding.

Error Messages

In the following list italicised text represents context specific text or values.

'*csn_version*' incompatible with '*elan_version*' ('*elan_version*' expected)
Error type is 2008 (Aborted). Occurs in `csn_init`(); Elan library version incompatibility. This library was linked with an out of date version of `libelan`.

'*csn_version*' incompatible with '*ew_version*' ('*ew_version*' expected)
Error type is 2008 (Aborted). Occurs in `csn_init`(); Elan Widget library incompatibility. This library was linked with an out of date version of `libew`.

Can't allocate *count* message descriptors
Error type is 2004 (No heap). Occurs in `csn_rxnb`() and `csn_txnb`(). A call to `calloc`() failed (insufficient memory). A descriptor is required for each pending non-blocking communication; tried to allocate a batch of additional descriptors for non-blocking communications but was unable. Maybe there are too many outstanding communications, are you clearing them with `csn_test`()?

Can't allocate message port
Error type is 2004 (No heap). Occurs in `csn_init`(); a call to `ew_allocate`()[1] failed maybe because heap or swap space exhausted.

Can't allocate yp ports
Error type is 2004 (No heap). Occurs in `csn_init`(). A call to `ew_allocate`() failed maybe because heap or swap space exhausted.

CS_ABORT (*message*: *status*)
Error type is 2008 (Aborted). Occurs if `cs_abort`() is called.

csn_checkVersion(self)
Error type is 2008 (Aborted). Occurs in `csn_init`(); internal incompatibility of library source files.

Unexpected flag *flag* in csn_test
Error type is 2005 (Bad request). Occurs in `csn_test`(); expecting either CSN_TXREADY or CSN_RXREADY but found something else. This is an internal library error, not an error that is directly attributable to the user (specifying the wrong type of flag to a function is flagged as an error by return codes from the function).

1. ew_allocate() is a Widget library function.

C o m p u t i n g
S u r f a c e

CSN Communications Library for Fortran

S1002–10M107.05

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

Using the Fortran CSN Library 1

The CSN routines provide access to the Computing Surface Network, which provides a general point to point communications scheme. These routines are not included in any of the Fortran libraries, but must be explicitly referenced from `libcsn` as shown later in this chapter.

CSN communications occur through *transports*; a transport is a bidirectional end point for communication. Each transport in the network has a unique address, which must be used by the sender of a message to identify the target of the communication. Individual programs can have many transports open simultaneously for the transmission and reception of messages. Facilities are provided (through the calls `csnregname()`, `csnlookupname()` and `csnderegname()`) to give meaningful names to transports, so that user code need not concern itself about the internal structure of network addresses.

Here is a full list of the CSN functions.

`csnclose()`	Close a CSN transport.
`csnderegname()`	Remove a name from a transport.
`csnexit()`	Shut down network connection and exit process.
`csngetid()`	Get transport address.
`csngetnet()`	CSN address manipulation. Defined as statement functions in the header, `csn/csnmcs.inc`.

`csngetnode()`	CSN address manipulation. Defined as statement functions in the header, `csn/csnmcs.inc`.
`csngettransport()`	CSN address manipulation. Defined as statement functions in the header, `csn/csnmcs.inc`.
`csninit()`	Set up the connection to the network.
`csnlookupname()`	Find a transport from a textual name.
`csnmakeid()`	CSN address manipulation. Defined as statement functions in the header, `csn/csnmcs.inc`.
`csnopen()`	Open a new transport.
`csnregname()`	Give a textual name to a transport.
`csnrx()`	Receive a message.
`csnrxnb()`	Queue a buffer for receiving a message.
`csnstatusstring()`	Return textual status.
`csntest()`	Test for completion of queued send/receive.
`csntx()`	Send a message.
`csntxnb()`	Queue a message for transmission.

The CS-2 `libcsn.a` library includes two new routines providing information on the number of processors and the ID of each processor.

`csnnnodes()`	Number of processors.
`csnnode()`	Processor ID.

The CS-2 `libcsn.a` library includes a number of support routines that were previously part of `libcs.a`; libcs.a itself is no longer needed.

`csabort()`	Terminate task.
`csgetinfo()`	Get processor information.

Functions for Starting-up and Shutting-down

There are various functions which are useful when starting up a program, and when closing it down. These include functions for giving names to transport addresses, so that other processes can communicate with them, and functions for finding out which process you are.

`csabort()`	Terminate task.
`csnclose()`	Close a CSN transport.
`csnderegname()`	Remove name from a transport.
`csnexit()`	Shut down network connection and exit process.
`csgetinfo()`	Get processor information.
`csninit()`	Set up the connection to the network.
`csnlookupname()`	Find a transport from a textual name.
`csnnnodes()`	Number of processors.
`csnnode()`	Processor ID.
`csnopen()`	Open a new transport.
`csnregname()`	Give a textual name to a transport.

Warning – `csninit()` must be called before using other CSN routines when running applications on the CS–2.

Functions for Performing Communication

The functions for performing communication can be split into two classes: those that do not complete until the communication has completed, and those that return immediately allowing the program to continue to execute while the communication takes place. The operation of the functions that suspend or block the user process are easier to understand; these functions are:

`csnrx()`	Receive a message.
`csntx()`	Send a message.

All of the message sizes on receive and transmit are given in bytes.

Functions for Non-blocking I/O

As well as the blocking CSN functions there are corresponding functions `csntxnb()`, and `csnrxnb()` that can be used to start communications while allowing the user program to continue to execute. Using these functions it is possible to queue up many buffers into which receives will occur when messages are sent, thus insulating the sender from delays in the receiver, or queue many buffers to be sent as soon as a receiver is willing to accept them.

As soon as the sender has many buffers queued up for transmission or reception, one needs a way of testing whether a buffer has been sent so that we may reuse or destroy the buffer. This functionality is provided by `csntest()`.

`csnrxnb()`	Queue a buffer for receiving a message.
`csntest()`	Test for completion of queued send/receive.
`csntxnb()`	Queue a message for transmission.

Header Files

Various constant values and type specifications are required when interfacing to the CSN. In particular, all the CSN functions are named with the initial letters `cs`, but their types are not implicit real. The header files include the correct type definitions for the CSN functions, and define macros names for various parameters and return values.

Two header files have been included in this release. These are called `csn.inc` and `csnmcs.inc`, and reside in `/opt/MEIKOcs2/include/csn`.

You must ensure that the contents of these files are included at the beginning of each Fortran CSN program — you can automate this process by including the following lines at the head of your program, and by passing it through a C preprocessor. Many compilers automatically invoke the C preprocessor if the Fortran file name includes a `.F` suffix in place of the usual `.f`.

```
    #include <csn/csn.inc>
C Variable declarations here
    #include <csn/csnmcs.inc>
C Executable code and statement functions ONLY here.
```

You should specify the search path for these header files to your compiler by using the command line option `-I/opt/MEIKOcs2/include`.

Library Files

All CSN libraries are stored in the directory `/opt/MEIKOcs2/lib`. Programs that use the CSN routines must be linked with the following command line options:

```
-L/opt/MEIKOcs2/lib -lcsn -lew -lelan
```

Tracing

To use the version of the CSN library that produces ParaGraph compatible trace files you precede the `-lcsn` in the above line by `-lcsn_pt`. Your attention is drawn to the following two sections which describe environment variables that are applicable to tracing, and also the tracing functions.

Debugging

There is also a debugging version of the library which attempts to provide more security and better error behaviour than the standard library — although it will also be slower. This library is available by specifying `-lcsn_dbg` in place of the standard version.

Environment Variables

The following environment variables are used by this library. Many are inherited from `libew` — the low level Elan Widget library.

`LIBCSN_TRACEFILE`	For use with `libcsn_pt` only, this variable specifies the name of the trace file to use; each node outputs to `$LIBCSN_TRACE-FILE`.*nodeno*. Default name is `LIBCSN_ TRACE`.*nodeno*.
`LIBCSN_TRACEBUF`	For use with `libcsn_pt` only, this variable specifies the number of events to allow in the trace buffer.
`LIBEW_WAITYPE`	Specifies how the low level Elan widget library (`libew`) routines wait for Elan events; either POLL or WAIT, default is to POLL.
`LIBEW_DMATYPE`	Specifies the type of DMA transfer used by the low level Elan widget library (`libew`). Either NORMAL or SECURE.
`LIBEW_DMACOUNT`	Specifies the permitted retry count for DMA transfers. Default is 1.
`LIBEW_RSYS_ENABLE`	Enables the remote system call server; when enabled `stdin`, `stdout`, and `stderr` are routed through the host process. May be either 0 (disabled) or 1 (enabled), default is 1.
`LIBEW_RSYS_BUFSIZE`	The buffer size used by the remote system call server. Default is 8192 bytes.
`LIBEW_RSYS_SERVER`	Virtual process ID of the processor that will run the system call server.
`LIBEW_CORE`	Enables core dump on exception. Values may be 1 (enabled) or 0 (disabled). By default core dumping is disabled.
`LIBEW_TRACE`	Enables a trace dump on exception. Values may be 1 (enabled) or 0 (disabled). By default trace dumping is disabled.

Program Tracing

Both ParaGraph and Alog/Upshot are supported for program tracing.

ParaGraph

Three C-language functions in the low level Elan Widget library (`libew`) are applicable to program tracing — these are `ew_ptraceStart()`, `ew_ptraceStop()`, and `ew_ptraceFlush()`. None of these take arguments and none return values to the caller.

Programs that are traced must be linked with `libcsn_pt` as described in an earlier section. The resulting trace file may be analysed with ParaGraph.

`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

Full documentation for the tracing functions is included in the *Elan Widget Library* reference manual.

Alog/Upshot

As an alternative to ParaGraph the event/state display tool `upshot` is also supported. To use this you need to instrument your code with trace points. Details may be found in `/opt/MEIKOcs2/upshot/README-MEIKO`.

Reference Manual 2

This chapter includes detailed descriptions of each function in the CSN library.

csabort() **Parallel communications routine**

Synopsis

```
#include <cs.inc>
subroutine csabort(string, exitcode)
character *(*) string
integer exitcode
```

Description

`csabort()` prints the given string to the standard output device, and then causes an exception. It will never return. No flushing of output buffers is performed, so this function should be used with caution.

See Also

`csnexit()`.

csgetinfo() **Parallel communications routine**

Synopsis

```
#include <cs.inc>
subroutine csgetinfo(nprocs, procid, localid)
integer nprocs, procid, localid
```

Description

`csgetinfo()` returns the number of processors involved in the program (`nprocs`), the identity of the local processor (`processorId` = 0...(`nprocs`-1)), and the identity of this process on this processor (currently always 0). The result will be 0 for success, and less than zero in the case of an error.

csnclose() **Close a CSN Transport**

Synopsis

```
#include <csn/csn.inc>
integer function csnclose(itransport)
integer itransport
```

Description

This function closes the transport `itransport`. The close will fail if there are any outstanding receives or transmits pending on the transport.

See Also

`csnopen()`, `csntest()`.

csnderegname() **Remove a transport's name**

Synopsis

```
#include <csn/names.inc>
integer function csnderegname(itransport)
integer itransport
```

Description

This function removes any name which was previously associated with the transport `itransport`. This is automatically performed when the transport itself is closed, so the only occasion on which this function needs to be explicitly called is if you wish to remove one name from a transport and then give it a new name. This is a rare occurrence.

See Also

`csnregname()`, `csnlookupname()`.

csnexit() — Shut down network connection and exit process

Synopsis

```
#include <csn/csn.inc>
subroutine csnexit(istatus)
integer istatus
```

Description

This subroutine never returns. It closes all of the transports and then causes the calling program to exit with status `istatus`. It can be used to provide a (relatively) clean termination in the case of an error.

To kill a parallel application, all processes should globally synchronise. Each process then calls `csnexit()`, but note that the process does not exit until all other processes have also called this function.

Warning – In current releases of this library, all outputs to the standard output device are routed through a single process (to ensure they are correctly line buffered). You must ensure that all output is complete before the IO process terminates.

csngetid() — Get the CSN address of a transport

Synopsis

```
#include <csn/csn.inc>
integer function csngetid(itransport)
integer itransport
```

Description

This function gets the CSN address of the local transport `itransport`.

See Also

`csngetnet()`, `csngetnode()`, `csngettransport()`, `csnmakeid()`.

csngetnet() — Extract network number from CSN address

Synopsis

```
#include <csn/csnmcs.inc>
csngetnet(peerid)
```

Description

This statement function is defined in the header file, `<csn/csnmcs.inc>`. It returns the network number from the CSN address, `peerid`, that is passed as an argument.

CSN addresses (as returned by `csnlookupname()` and other CSN functions), consist of three parts: the network number, the node number, and the transport number.

See Also

`csngetnode()`, `csngettransport()`, `csnmakeid()`.

csngetnode() — **Extract node number from CSN address**

Synopsis

```
#include <csn/csnmcs.inc>
csngetnode(peerid)
```

Description

This statement function is defined in the header file, `<csn/csnmcs.inc>`. It returns the node number from the CSN address, `peerid`, that is passed as an argument.

CSN addresses (as returned by `csnlookupname()` and other CSN functions), consist of three parts: the network number, the node number, and the transport number.

See Also

`csngetnet()`, `csngettransport()`, `csnmakeid()`

csngettransport() — Get transport number from CSN address

Synopsis

```
#include <csn/csnmcs.inc>
csngettransport(peerid)
```

Description

This statement function is defined in the header file, `<csn/csnmcs.inc>`. It returns the transport number from the CSN address, `peerid`, that is passed as an argument. This only makes sense if the relevant transport is local to the processor calling the function.

CSN addresses (as returned by `csnlookupname()` and other CSN functions), consist of three parts: the network number, the node number, and the transport number.

See Also

`csngetnet()`, `csngetnode()`, `csnmakeid()`.

csninit() **Initialise the CSN**

Synopsis

```
#include <csn/csn.inc>
subroutine csninit()
```

Description

This subroutine sets up the network connection between the current process and the CSN network — it must be the first function that is called by the process.

Before the CSN can be used, the subroutine `csninit()` must be called to perform any system initialisation which may be required. After calling `csninit()`, a program will normally create a set of Transports (using `csnopen()`), give each of the transports a meaningful name (using `csnregname()`), and then (using `csnlookupname()`) discover the addresses of the transports to which it intends to transmit. It is normal for all programs to create their transports before looking up any others to avoid potential deadlocks where two programs are each waiting for the other to create and register a transport.

csnlookupname() — Look-up a named Transport

Synopsis

```
#include <csn/names.inc>
integer function csnlookupname(inetaddr, cname, lblock)
integer inetaddr
character *(*) cname
logical lblock
```

Description

This function looks up the name, `cname`, and returns the associated CSN address in the variable, `inetaddr`. The argument, `lblock`, determines the behaviour of the function when the given name has not yet been registered. If `lblock` is `.true.` then `csnlookupname()` does not return until the name is registered, otherwise `csnlookupname()` returns immediately with an error status as its result. Note that it is advisable that a process always registers transport names before looking-up, to prevent deadlock. If this advice is not followed, you should not set `lblock` to `.true.`.

Example

Here is a sample code fragment which looks up a transport called MASTER.

```
C
C Find the master
C
   if (csnlookupname(masterTpt,'MASTER',.TRUE.) .ne. CSN OK) then
       stop 'Slave can't find master'
    end if
```

See Also

`csnregname()`, `csnderegname()`.

csnmakeid() — **Assemble CSN address**

Synopsis

```
#include <csn/csnmcs.inc>
csnmakeid(netid, nodeid, transportid)
```

Description

This statement function is defined in the header file, `<csn/csnmcs.inc>`. It assembles a CSN address from a network number, `netid`, a node number, `nodeid`, and a transport number, `transportid`.

CSN addresses (as returned by `csnlookupname()` and other CSN functions), consist of three parts: the network number, the node number, and the transport number.

Warning – In the current implementation `netid` must be 0.

Warning – Manipulation of the internal structure of network addresses is not recommended.

See Also

`csngetnet()`, `csngetnode()`, `csngettransport()`.

csnnodes() **Number of Processors**

Synopsis

```
#include <csn/csn.inc>
integer function csnnodes();
```

Description

Parallel programs are run one process per processor on the CS–2. This function returns the number of processors executing this application.

See Also

`csnnode()`.

csnnode() **Processor Id**

Synopsis

```
include <csn/csn.inc>
integer function csnnode();
```

Description

Parallel programs are run one process per processor on the CS–2. This function returns the ID of the processor executing this process. IDs will lie in the range 0 to `nodes-1`, where `nodes` is returned by `csnnnodes()`.

See Also

`csnnnodes()`.

csnopen() — Open a CSN Transport

Synopsis

```
#include <csn/csn.inc>
integer function csnopen(index, itransport)
integer index, itransport
```

Description

This function allows a program to create a transport, and thus to access the CSN. The first argument is the network address to give to the created port, or CSN-NULLID to allow the system to choose a suitable address. (Advice: always let the system choose). The second argument is assigned the transport that is created. The result is zero on success, or a negative value on failure.

Example

To create a transport:

```
   integer mastertpt
c
c Create the master transport
c
   if (csnOpen(CSN NULL ID, mastertpt) .ne. CSN OK) then
      stop 'Master failed to open a transport'
   end if
```

See Also

csnclose(), csnregname(), csnlookupname().

csnregname() — Name a CSN Transport

Synopsis

```
#include <csn/names.inc>
integer function csnregname(itransport, cname)
integer itransport
character *(*) cname
```

Description

This function associates the textual name, `cname`, with the transport, `itransport`. When the name has been associated, then other processes within the configuration can obtain the network address of the transport by performing a `csnlookupname()`.

Example

To create and name a transport we use `csnopen()` and `csnregname()` as shown below:

```
   integer mastertpt
c
c Create the master transport
c
   if (csnOpen(CSN NULL ID, mastertpt) .ne. CSN OK) then
      stop 'Master failed to open a transport'
   end if
c
c Register a name for the Transport
c
   if (csnRegName(mastertpt,'MASTER') .ne. CSN OK) then
      stop 'Master failed to register name ''MASTER'''
   end if
```

See Also

`csnlookupname()`, `csnderegname()`.

csnrx() — Blocking receive via CSN

Synopsis

```
#include <csn/csn.inc>
integer function csnrx(itransport, ipeerid,
                       ibuffer, imaxsize)
integer itransport, ipeerid, ibuffer, imaxsize
```

Description

This function receives a message on transport `itransport` into the buffer `ibuffer`. The maximum message size which will be accepted is `imaxsize`. The argument `ipeerid` must be a VARIABLE, since it is assigned the transport address of the transport from which the received message was sent.

The function returns the number of bytes actually received, or an error code.

Example

To receive a four byte message:

```
null = CSN NULL ID
if (csnrx(slavetpt, null, processno, 4) .ne. 4) then
   stop 'Slave failed to receive process number'
end if
processno = processno + 1
call csntx(slavetpt, 0, nexttpt, processno, 4)
```

See Also

`csntx()`, `csnrxnb()`.

csnrxnb() — Non-blocking receive via the CSN

Synopsis

```
#include <csn/csn.inc>
integer function csnrxnb(itransport, ibuffer,
                                 imaxsize, itag)
integer itransport, ibuffer(*), imaxsize, itag
```

Description

This routine is the non-blocking analogue of `csnrx()`. It is used to queue a buffer into which reception of messages will occur. As with `csntxnb()` the tag is used to identify this particular transaction to `csntest()`.

Example

Here is a call from the master in a load balancer in which it queues up a number of buffers to receive results from the slaves. An array of buffers is used, the index of the buffer being used as its tag.

```
C
C First queue up the result buffers, their tags are negated, so
C that they can easily be distinguished from the job buffers when
C we do the csnTest.
C
   do i = 1, nresultbuffers
   call csnrxnb(mastertpt, resultBuffer(0,i),
  + (resultSize+1)*4, -i)
   end do
```

See Also

`csntxnb()`, `csnrx()`.

csnstatusstring() **Return CSN error string**

Synopsis

```
#include <csn/csn.inc>
character *(*) csnstatusstring(ierrno)
integer ierrno
```

Description

This function returns a string containing a textual version of the CSN error code `ierrno`.

csntest() — **Test for completion of non-blocking communication**

Synopsis

```
#include <csn/csn.inc>
integer function csntest (itransport, iflags,
                          timeout, ipeerid, itag, status)
integer itransport, iflags, timeout
integer ipeerid, itag, status
```

Description

This routine tests for the completion of communications initiated by the non-blocking calls `csntxnb()` and `csnrxnb()`. It waits for timeout microseconds (or forever if the timeout argument is CSNNULLTIMEOUT) for a buffer meeting the criteria set by the `iflags` and `itag` arguments to be found.

The `iflags` argument determines what sort of communication is being tested for completion, it can be 0 meaning either transmission or reception, or one of the values CSNTXREADY or CSNRXREADY to test for the readiness of a buffer queued by `csntxnb()` or `csnrxnb()` respectively.

The `ipeerid` argument must be a variable, since it is assigned within the function with the value of the network address with which the successful communication took place. In addition if the value on entry to the function is not CSNNULLID, then only buffers involved in communication with that specific network address are considered. (Note that it is an easy bug to forget to re-assign CSNNULLID to the variable passed to the formal argument `ipeerid`, this has the effect of unnecessarily filtering the `csntest()` call, and will manifest itself either as a deadlock, or a starvation of all but one other network address).

The `itag` argument must be a variable, since it is assigned the tag which was associated with the buffer whose communication has completed. As with the `ipeerid` the initial value of the `itag` argument is used as a selection criterion, so if all buffers are to be considered then the `itag` formal argument must be assigned the value CSNNULLTAG.

Warning – `csntest()` **must be used to free-up the memory used by non-blocking communications.**

The return values from `csntest()` are as follows:

`CSNTXREADY`	A communication initiated by `csntxnb()` completed or cancelled.
`CSNRXREADY`	A communication initiated by `csnrxnb()` completed or cancelled.
0	No specified communications completed and at least timeout microseconds had elapsed.
`CSNEBADREQ`	Illegal values for `flags`.
`CSNEABORT`	Transport was closed while `csntest()` was blocked. Note that a transport may only be closed after all outstanding communications on it have completed. When either CSNTXREADY or CSNRXREADY are returned, the value of `status` may be used to determine if the communication completed or was cancelled. `status` is set to CSNEABORT if it was cancelled.

See Also

`csntxnb()`, `csnrxnb()`.

csntx() Blocking transmission via CSN

Synopsis

```
#include <csn/csn.inc>
integer function csntx(itransport, iflags, ipend,
                              ibuffer, isize)
integer itransport, iflags, ipend, ibuffer, isize
```

Description

This function transmits a message through the transport `itransport` to the transport whose address is `ipend`. The message data is taken from `ibuffer`, and the number of bytes transmitted is `isize`. The argument `iflags` is not currently used and should be set to 0.

The function will not return until either an error can be detected, or the data has been placed in a user buffer at the recipient. The result returned is the number of bytes sent if the transmission was successful, or an error return if the transmission failed.

Example

To send the 4 byte integer, 0, through the transport mastertpt:

```
C
C Inject zero into the front of the pipe
C
if (csntx(mastertpt, 0, nexttpt, 0, 4) .ne. 4) then
   stop 'Master can''t inject zero into pipe'
end if
```

See Also

`csnrx()`, `csntxnb()`.

csntxnb() — Non-blocking transmission via CSN

Synopsis

```
#include <csn/csn.inc>
integer function csntxnb(itransport, iflags, peerid,
                         ibuffer, isize, itag)
integer itransport, iflags, peerid
integer ibuffer(*), isize, itag
```

Description

The arguments to this routine are identical to those for `csntx()`, but with an additional `itag` argument. This is used to identify this transaction when querying its status using `csntest()`. The return from the function occurs as soon as the buffer has been queued, thus a successful return from `csntxnb()` does not imply that the data has been sent yet, merely that there were sufficient local resources to request transmission. The return status for the whole transaction is returned by the call to `csntest()` which returns this buffer. The contents of the buffer will not be copied by the system, and should not therefore be modified until the system has returned ownership of the buffer by returning it as the result of a `csntest()` call.

Example

Here is a call from the master in a load balancer which is queuing a job to send to a slave. Here the master has allocated a two dimensional array to serve as buffers, each column representing a single buffer. The column index is then used as the tag, so that the correct buffer can be reused when the `csntest()` is complete.

```
C
C There is a job to be done, so queue it.
C
   call csntxnb(mastertpt, 0, slavetpt(i),
  +   jobbuffer(0,i), (jobsize+1)*4, i)
```

See Also

`csntx()`, `csnrxnb()`.

Tutorial Examples 3

Overview

This chapter includes a number of examples showing how to use the CSN communication library. It discusses the use of *transports* and the choice of *blocking* versus *non-blocking* communications.

Compilation and Execution

All the examples in this chapter can be compiled with the following command line:

```
user@cs2: f77 -o myprogram -I/opt/MEIKOcs2/include \
-L/opt/MEIKOcs2/lib myprogram -lcsn -lew -lelan
```

The programs are executed with prun(1) and will use command lines like that shown below. Note that *number* is the number of processors required, *partition* is the name of the partition that you will use, and *myprogram* is the name of the program.

```
user@cs2: prun -nnumber -ppartition myprogram
```

Full information about prun(1) command may be obtained from the reference manual page.

Two Communicating Processes

The following example defines two processes that use a single blocking CSN communication for synchronisation.

This example introduces *transports* and shows how they are used for a simple blocking communication between two processes.

Transports

A *transport* is a connection from a process to the Computing Surface Network. There is no limit on the number of transports that a process can use, so it is normal to create a transport that is dedicated to specific classes of communication, or to specific senders. In this example each process uses just one transport.

Each transport has an associated address, or *net id*. To send data to a remote transport the sender must first determine the address of the destination transport. To do this the receiver registers a name for its transport with `csnregname()`; the sending process determines the net id of this transport by looking-up the name with `csnlookupname()`.

A useful analogy that helps explain the use of transports is to compare the CSN with a telephone network. Using this analogy people represent processes, the telephone lines represent transports, and the telephone exchange represent the CSN network. Each person's telephone line allows them to communicate with any other (and there may be many lines each dedicated to a specific type of communication) but to make a call the person must first determine the receiver's number by looking up a name in the directory.

Blocking Communications

The CSN supports two types of communication: blocking and non-blocking. In this example we consider blocking communications — the communication between sender and receiver is delayed until both processes have called their communication function. It is this implicit sychronisation that is exploited in this example.

Program Description

This example is a simple program that writes `Hello` and `World` on your screen. There are two processes; one writes `Hello`, the other writes `World`. A simple blocked communication is used to synchronise the processes.

The program begins with initialisation code that is common to both processes. `csninit()` is used to initialise the network, `csgetinfo()` identifies each process's virtual process number and the total number of processes in the application, and `csnopen()` creates a transport.

The process with virtual process number 0 will be the sender of the blocked communication. The sender determines the network address of the recipient's transport by looking-up the transport's name with `csnlookupname()` (the third argument is non-zero indicating that `csnlookupname()` should wait for the other process to register its transport's name if it has not already done so). Our sending process then writes its string to the screen, and uses `csntx()` to send a simple integer data item. At this point the sender will block until the recipient is ready to take the data.

Process 1 is the recipient of the communication. The recipient must register a name for its transport with `csnregname()` so that it is visible to our sender. The recipient waits until it receives a communication from the sender (using `csnrx()`), and then writes its part of the string to the screen.

Both process finish by calling `csnexit()`.

Program Listing

```
      PROGRAM hello

      IMPLICIT NONE

#include <csn/csn.inc>
#include <csn/names.inc>

      INTEGER transport, networkid, flag, status, nob
      INTEGER sizeofflag, sender
      INTEGER nprocs, me, dummy

      PARAMETER (flag=1, sizeofflag=4)

      CALL csninit()

      status = csgetinfo(nprocs, me, dummy)

      IF (nprocs.NE.2) THEN
         CALL csabort('Need two processors for this example', 1)
      ENDIF

      status = csnopen(CSNNULLID, transport)
      IF (status.NE.CSNOK) THEN
         CALL csabort('Cannot open transport',1)
      ENDIF

      IF (me.EQ.0) THEN
C
C Process 0 is the sender
C
        status = csnlookupname(networkid, 'Receiver', .TRUE.)
        IF (status.NE.CSNOK) THEN
           CALL csabort('Cannot lookup transport', 1)
        ENDIF

        PRINT *,'Hello '

        nob = csntx(transport, 0, networkid, flag, sizeofflag)
        IF (nob.NE.sizeofflag) THEN
           CALL csabort('Failed to transmit', 1)
        ENDIF
      ELSE
```

```
C
C Process 1 is the receiver
C
        status = csnregname(transport, 'Receiver')
        IF (status.NE.CSNOK) THEN
           CALL csabort('Cannot register transport', 1)
        ENDIF

        nob = csnrx(transport, sender, flag, sizeofflag)
        if (nob.NE.sizeofflag) THEN
           CALL csabort('Failed to receive', 1)
        ENDIF

        PRINT *,'World'
      ENDIF

      CALL csnexit(0)

      END
```

Bidirectional Communications

The following example is suitable for use with 2 or more processors. It defines a master process and a number of slaves; the slaves send data to a master which broadcasts a result back.

The example shows how to use transports for bidirectional communications, and also introduces a style of programming that is suitable for a variable number of target processors.

Transports

In this example each process creates just one transport that is used for both incoming and outgoing communications. The processes could use a separate transport for each direction, or indeed dedicate a transport to each pair or processes.

To select the best use of transports for your application you should consider the message receiving functions `csnrx()` and `csnrxnb()`. These can both identify the network address of the sending transport (although this facility is not used in this example). By using a transport for a specific type of message the recipient of a message can infer a context for the data that it has received.

Program Description

All the processes begin by calling `csninit()` to initialise the network, and follow this with a call to `csgetinfo()` to get their virtual process number and the number of processes in the application. Each process then opens a single transport which will be used for both outgoing and incoming communications.

Each process registers it's own transport's name, and then looks-up the network address for all the other transports. Note that each transport's name is derived from the owning process's virtual process number, and that the network addresses are stored in an array that is indexed by virtual process number[1]. This strategy keeps the program code compact, and allows the number of target processors to be specified at execution time.

At this point the program splits into the code for our master, and code for the slaves. The master receives from each slave data that is simply added and then broadcast back to all the slaves.

Program Listing

```
      PROGRAM master

      IMPLICIT NONE

#include <csn/csn.inc>
#include <csn/names.inc>
#define MAXPROCS 20
#define NAMELEN 20

      INTEGER transport
      INTEGER networkid(MAXPROCS)
      INTEGER nprocs, me, dummy
      INTEGER status, nob
      INTEGER i, j
      INTEGER result
      CHARACTER*NAMELEN name

      INTEGER data, sizeofint
      PARAMETER (sizeofint = 4)
```

1. Note that process numbers start at 0 but the arrays are indexed from 1 (i.e. process id +1).

```
10     FORMAT(A,I1)

C  Initialise CSN
       CALL csninit()

C Get my process id & number of procs
       status = csgetinfo(nprocs, me, dummy)

       IF (nprocs.GT.MAXPROCS) THEN
          CALL csabort('Too many processors', 1)
       ENDIF

C Open transport
       status = csnopen(CSNNULLID, transport)
       IF (status.NE.CSNOK) THEN
          CALL csabort('Cannot open transport', 1)
       ENDIF

C Register my transport

       write(name,10) 'Proc',me
       status = csnregname(transport, name)
       IF (status.NE.CSNOK) THEN
          CALL csabort('Cannot register transport')
       ENDIF


C Look up all the other transports (but not my own)
C Remember proc ids are 0-(n-1) but the networkid array is indexed from 1 ...
C ... so 'i' is a processor id, 'j' is index into the array.

       i = 0
       j = 1
       DO WHILE (i.LT.nprocs)
          IF (me.NE.i) THEN
             write(name,10) 'Proc',i
             status = csnlookupname(networkid(j), name, .true.)
             IF (status.NE.CSNOK) THEN
                CALL csabort('Cannot lookup transport',1)
             ENDIF
             j = j+1
          ENDIF
          i = i+1
       ENDDO
```

```
C Process 0 is the master

      IF (me.EQ.0) THEN

C Get data from all the workers
         i = 1
         result =0

         DO WHILE (i.LT.nprocs)

            nob = csnrx(transport, 0, data, sizeofint)
            IF (nob.NE.sizeofint) THEN
               CALL csabort('Failed to receive',1)
            ENDIF

            i = i+1
            PRINT *,'Master receives data'
            result = result+data

         ENDDO


C Now broadcast a result back to all the processes

         i = 1
         DO WHILE (i.LT.nprocs)
            nob = csntx(transport, 0, networkid(i), result, sizeofint)
            IF (nob.NE.sizeofint) THEN
               CALL csabort('Failed to transmit',1)
            ENDIF
            i = i+1
         ENDDO

      ELSE

C I am a worker
C Send some data (my process id) to the master.

         data = me

         nob = csntx(transport, 0, networkid(1), data, sizeofint)
         IF (nob.NE.sizeofint) THEN
            CALL csabort('Failed to transmit',1)
         ENDIF
```

```
C Get a result back from the master

         nob = csnrx(transport, 0, result, sizeofint)
         IF (nob.NE.sizeofint) THEN
            CALL csabort('Failed to receive',1)
         ENDIF

         PRINT *, 'Received from master:', result

      ENDIF

      CALL csnexit(0)

      END
```

Non-Blocking Communications

The following example runs on 2 processors. It defines a Producer process that wishes to send a large number of messages to a Consumer process.

The example simulates the case where a process wishes to send a large number of non-blocking messages to a receiver process. The receiver does not know in advance how many messages will be sent, nor can the producer assume that the consumer has sufficient heap space to receive them all. The producer and consumer therefore periodically synchronise with a blocking communication so that the number of non-blocking communications is agreed before they are sent.

Non-Blocking Communications

This form of communication between processes does not require the sender and recipient to synchronise, and is therefore more appropriate to time critical applications where processes cannot be allowed to idle.

Non-blocking communications allow a sender to initiate a transmission and to continue immediately without waiting for the communication to complete. Similarly a receiver can initiate a receive without waiting for the message to arrive.

Non-blocking sends are initiated by `csntxnb()`. The data identified by this function will be transferred from the process's address space at some indeterminate time in the future. To test the status of the transfer the program *must* use `csntest()` — only when the transfer has completed may the data buffer be modified or destroyed.

Non-blocking receives are initiated by `csnrxnb()`. This function identifies a data buffer that can receive the incoming data. To test the status of the transfer the program *must* use `csntest()` — only when the transfer has completed may the data buffer be modified or destroyed.

Program Description

Following the initialisation of the CSN and of each process's transports the program defines two processes: process 0 is a producer, and process 1 a consumer.

The producer sends a blocking communication to the consumer to agree a number of non-blocking communications that may follow. If the consumer accepts, the agreed number of non-blocking sends are initiated with `csntxnb()`. The producer can, without waiting for the communications to complete, continue with other meaningful work, until it is ready to use `csntest()` to confirm that the transfers completed successfully.

The consumer awaits the blocking communications from the producer by making the required number of calls to `csnrxnb()`. Each call identifies a unique data buffer for each of the incoming communications — these buffers must not be modified or destroyed until the communications are complete. The receiver can test the status of the communications at any time by calling `csntest()`.

Program Listing

```
      PROGRAM nonblock

      IMPLICIT NONE

#include <csn/csn.inc>
#include <csn/names.inc>
#define MAXMESSAGES 50
#define NAMELEN 20
#define STOP -1
#define REQSIZE 10
#define TIMEOUT 1000000
#define MAXBUFFS 100

      INTEGER transport
      INTEGER networkid
      INTEGER nprocs, me, dummy
      INTEGER status, nob, peerid, tag
      INTEGER data
      PARAMETER (data=99)
      INTEGER i
      INTEGER rxbuffer(MAXBUFFS)
      INTEGER messages, requestsize
      CHARACTER*NAMELEN name
      INTEGER sizeofint
      PARAMETER (sizeofint=4)

C Initialise
      CALL csninit()

C Get my process id & number of procs
      status = csgetinfo(nprocs, me, dummy)

      IF (nprocs.NE.2) THEN
         CALL csabort('This example requires 2 processors',1)
      ENDIF

C Open my transport
      status = csnopen(CSNNULLID, transport)
      IF (status.NE.CSNOK) THEN
         CALL csabort('Cannot open transport',1)
      ENDIF

C Register my transport
```

```
 20    format (A,I1)

       write(name,20), 'Proc', me

       status = csnregname(transport, name)
       IF (status.NE.CSNOK) THEN
          CALL csabort('Cannot register transport',1)
       ENDIF

C Lookup my partner's transport

       IF (me.EQ.0) THEN
          write(name,20), 'Proc', 1
       ELSE
          write(name,20), 'Proc', 0
       ENDIF

       status = csnlookupname(networkid, name, 1)
       IF (status.NE.CSNOK) THEN
          CALL csabort('Cannot lookup transport',1)
       ENDIF


       IF (me.EQ.0) THEN

C         Process 0 is the producer

          messages = MAXMESSAGES

          DO WHILE (messages.GT.0)

C            request a batch of buffers ...

             IF (messages.GT.REQSIZE) THEN
                requestsize = REQSIZE
             ELSE
                requestsize = messages
             ENDIF

             messages = messages - requestsize

             PRINT *,'Producer requests', requestsize, ' buffers'

C            ... with a blocking communication
```

```
            nob = csntx(transport,0,networkid,requestsize,sizeofint)

            IF (nob.NE.sizeofint) THEN
               CALL csabort('Failed blocking transmit',1)
            ENDIF

C           Send a batch of messages

            i=1
            DO WHILE (i.LE.requestsize)
               PRINT *, 'Producer sets-up non-blocking send'

C              ... with a non-blocking communication

               status=csntxnb(transport,0,networkid,data,sizeofint,i)
               IF (status.NE.CSNOK) THEN
                  CALL csabort('Producer failed to transmit',1)
               ENDIF

               i = i+1
            ENDDO

C           Do some work here, if we want to

            PRINT *, 'Producer doing some other work'

C           test for completion of non-blocking transmits
C           and also free-up internal CSN buffers

            i=1
            DO WHILE (i.LE.requestsize)
               peerid = CSNNULLID
               tag = CSNNULLTAG
               status = csntest(transport, CSNTXREADY, TIMEOUT,
     +            peerid, tag, status)

               IF (status.NE.CSNTXREADY) THEN
                  CALL csabort('Non-blocking timeout or failure',1)
               ENDIF

               PRINT *, 'Producer reports completion'

               i = i+1
            ENDDO
```

```
         ENDDO

C        No more messages so request consumer to stop
         requestsize = STOP

         PRINT *,'Producer requests consumer to STOP'

         nob = csntx(transport,0,networkid,requestsize,sizeofint)

         IF (nob.NE.sizeofint) THEN
            CALL csabort('Failed blocking transmit',1)
         ENDIF

      ELSE

C        Process 1 is the consumer

         DO WHILE (.TRUE.)

            nob = csnrx(transport, 0, requestsize, sizeofint)
            IF (nob.NE.sizeofint) THEN
               CALL csabort('Failed to receive',1)
            ENDIF

C           Is this a request to stop?
            IF(requestsize.EQ.STOP) THEN
               GOTO 10
            ENDIF

C           Allocate requested number of buffers
            PRINT *, 'Consumer receives request for ',
     +         requestsize, ' buffers'

C           Allocate buffer space
C           Should create heap space, but I'll use stack here.
            IF(requestsize.GT.MAXBUFFS) THEN
               CALL csabort('Exceeded size of rxbuffer array',1)
            ENDIF

            i = 1
            DO WHILE (i.LE.requestsize)
               status = csnrxnb(transport,rxbuffer(i),sizeofint,i)
               IF (status.NE.CSNOK) THEN
                  CALL csabort('Failed non-blocking receive',1)
```

```
            ENDIF

            PRINT *,'Consumer sets-up non-blocking receive'

            i = i +1
         END DO

C        We could do some work here, if we want to
         PRINT *,'Consumer doing some other work'

C        test for completion of non-blocking transmits
C        and also free-up internal CSN buffers

         i = 1
         DO WHILE (i.LE.requestsize)
            peerid = CSNNULLID
            tag = CSNNULLTAG
            status = csntest(transport, CSNRXREADY, TIMEOUT,
     +         peerid, tag, status)

            IF (status.NE.CSNRXREADY) THEN
               CALL csabort('Non-blocking timeout or failure',1)
            ENDIF

            i = i + 1
         END DO

         PRINT *,'Consumer reports completion'

       END DO
      ENDIF

10    CALL csnexit(0)

      END
```

Error Messages 4

Message Format

The functions in the CSN library (`libcsn`) are built upon the functions in the Elan Widget library. Errors within `libcsn` are reported via the Widget library exception handler; this writes diagnostic messages to the standard error device and kills the application.

The format of `libcsn` messages is:

```
CSN EXCEPTION @ process : error_code (error_text)
Additional information: error message string
```

The *error message strings* are described later in this chapter. The *process* is the virtual process number of the process that detected the error; if the exception occurs before the process has attached to the network (i.e. before `csn_init()` is called) then this is shown as `----`. The *error code* (and its textual equivalent the *error text*) are one of:

Error Code	Error Text
2000	`Ok`
2001	`No Destination`
2002	`Buffer Overflow`
2003	`No space at destination`

Error Code	Error Text
2004	No heap
2005	Bad request
2006	Already allocated
2007	Out of range
2008	Aborted
2009	Not ready
2010	Interrupted
2011	Bad Address

Widget Library Exceptions

Functions in `libcsn` are implemented on functions in the Elan Widget library. When an exception occurs within a Widget library function this is handled by the Widget library's own exception handler. The Widget library handler is similar to that used by `libcsn` but produces errors in the form:

```
EW_EXCEPTION @ process : error_code (error_text)
error message string
```

These exceptions are fully described in *The Elan Widget Library*, Meiko document number S1002–10M104.

Note for Fortran Programmers

All errors apply to both C and Fortran implementations unless the description specifies a specific language. Often the error message repeats the parameters that were passed to the failed call; these will be the parameters that were passed to the underlying C implementation of the function, and may not be identical to those passed to the Fortran binding.

Error Messages

In the following list italicised text represents context specific text or values.

'*csn_version*' incompatible with '*elan_version*' ('*elan_version*' expected)
Error type is 2008 (Aborted). Occurs in `csn_init`(); Elan library version incompatibility. This library was linked with an out of date version of `libelan`.

'*csn_version*' incompatible with '*ew_version*' ('*ew_version*' expected)
Error type is 2008 (Aborted). Occurs in `csn_init`(); Elan Widget library incompatibility. This library was linked with an out of date version of `libew`.

Can't allocate *count* message descriptors
Error type is 2004 (No heap). Occurs in `csn_rxnb`() and `csn_txnb`(). A call to `calloc`() failed (insufficient memory). A descriptor is required for each pending non-blocking communication; tried to allocate a batch of additional descriptors for non-blocking communications but was unable. Maybe there are too many outstanding communications, are you clearing them with `csn_test`()?

Can't allocate message port
Error type is 2004 (No heap). Occurs in `csn_init`(); a call to `ew_allocate`()[1] failed maybe because heap or swap space exhausted.

Can't allocate yp ports
Error type is 2004 (No heap). Occurs in `csn_init`(). A call to `ew_allocate`() failed maybe because heap or swap space exhausted.

CS_ABORT (*message*: *status*)
Error type is 2008 (Aborted). Occurs if `cs_abort`() is called.

csn_checkVersion(self)
Error type is 2008 (Aborted). Occurs in `csn_init`(); internal incompatibility of library source files.

Unexpected flag *flag* in csn_test
Error type is 2005 (Bad request). Occurs in `csn_test`(); expecting either CSN_TXREADY or CSN_RXREADY but found something else. This is an internal library error, not an error that is directly attributable to the user (specifying the wrong type of flag to a function is flagged as an error by return codes from the function).

1. ew_allocate() is a Widget library function.

Computing Surface

Tagged Message Passing & Global Reduction

S1002–10M108.06

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

Introduction 1

The message passing functions described in this document use tagged message passing; each message has an associated user-specified tag, and receivers' may elect to filter incoming messages using these tags. The library also defines a number of global operations.

A tracing version of the library is available which produces ParaGraph compatible trace files, and a debugging version of the library is also provided offering greater security and better error behaviour.

Implementation Notes

This library is implemented on the low level communication functions in the Elan Widget Library (libew) and the resource management functions in the Resource Management User Interface Library (librms). (Both libraries are described in separate documents within your CS-2 documentation set.)

This section describes how the architecture of the CS-2 affects the implementation of this library.

Programming Models

This implementation of libmpsc supports both hosted and hostless applications.

Hosted applications consist of two programs; a host and a number of identical node processes. The libmpsc application is initiated by executing the host process which is then responsible for spawning the node processes. All processes, including the host itself, use libmpsc communication functions to cooperate and complete the task.

Hostless applications have a number of identical node processes that are started by using a loader program such as `prun`.

Resource Allocation

All libmpsc applications must liaise with the CS-2 Resource Manager for processing resource. This liaison takes place within either the host process (for hosted applications) or the loader process (for hostless applications).

In either case the host/loader runs in your login partition as a sub-process of your command shell. The host/loader process calls upon functions in the resource management user interface library to liaise with the resource manager for the nodes' processing resource. In the case of a loader, such as `prun`, the liaison is via a direct calls to `rms_forkexecvp()` in librms. In the case of a host process the liaison happens when the host process calls `mpsc_getnodes()` or `load()`, (which in turn call `rms_forkexecvp()`).

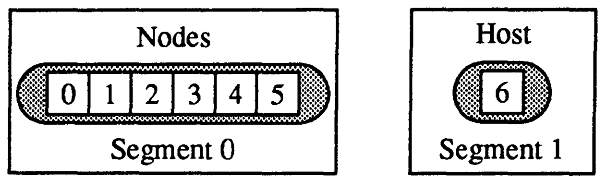
The resource management function uses the user's id and other criteria specified by your System Administrator to identify a suitable partition for the node processes. If you don't like the default resource you can specify your preferences by setting environment variables — the most useful variable is `RMS_PARTITION` which identifies your preferred partition, but there are others too (see page 7 or the documentation for `rms_forkexecvp()`). Alternatively you can explicitly pre-allocate resources using the `allocate` command or `mpsc_getnodes()`.

Process Communication

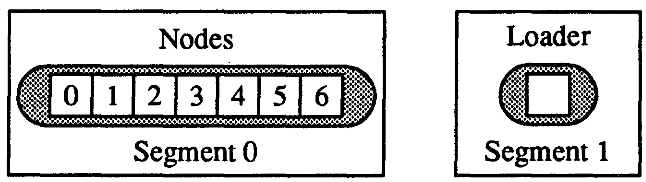
libmpsc communication functions are built upon the tagged message port (TPORT) functions in the Elan Widget library. libmpsc applications are 2 segment CS-2 applications in which the host or loader program and the nodes run in separate segments. The two segments will usually run in separate partitions.

libmpsc processes have two numbering schemes associated with each process: there are the node ids which are visible within the libmpsc application, and there are internal (virtual process) numbers that are used by the low level communication routines. In this implementation the node ids and virtual process ids are the same.

For an example 6 process libmpsc application the virtual process numbers/node id's are assigned as shown, with the node processes numbered from 0:



For a 6 process hostless application the virtual process numbers and the node ids are allocated as follows — note that the loader program does not form part of the application and has no id of its own:



In general the allocation of each segment's processes to processors in a partition mirrors the allocation of the virtual process numbers; processes with low virtual process numbers are usually allocated to processors with low Elan id's.

Features of this Release

This manual describes `libmpsc` version 3.0. The reader is advised to note the following points in relation to this implementation:

- `csend()` and `isend()` support only a single destination node or all nodes.
- Only process ID 0 is supported; there is only 1 process per node.
- Only process type 0 is supported with the extended receive and probe functions; there is only one process per node.

- Only exact match and match any tag selectivity is supported (that is, no bitmask encoding when tag is less than -1).
- There are no "force" types.
- There are no versions of message passing calls that deliver signals.
- The use of the special array `msginfo` with extended receive and probe is not supported.

Compiling and Linking libmpsc Programs

The header file `mpsc.h` in `/opt/MEIKOcs2/include/mpsc` contains prototype definitions for libmpsc. You should therefore compile with the option `-I/opt/MEIKOcs2/include` and refer to the header file in your programs with `#include <mpsc/mpsc.h>`.

Several variants of the library are provided; all are available in the directory `/opt/MEIKOcs2/lib.`

Node Programs

Node programs should be compiled with the following options:

```
-I/opt/MEIKOcs2/include -L/opt/MEIKOcs2/lib -lmpsc -lew -lelan
```

Host Programs

Host programs must be linked with `-lmpsc_host` (in addition to those libraries used by node programs) and you must also specify the Meiko `lib` directory after the `-R` option (to ensure that the dynamic libraries can be found at run time):

```
-I/opt/MEIKOcs2/include -L/opt/MEIKOcs2/lib -R/opt/MEIKOcs2/lib\
-lmpsc_host -lmpsc -lew -lelan
```

Programs that are linked without the `-R` option will fail to execute with the following error message.

```
ld.so.1: mkaudit: fatal: librms.so.2: can't open file: errno=2
Killed
```

To overcome this you must either recompile the application, or you can include in your `LD_LIBRARY_PATH` variable the pathname of the Meiko library directory as shown in the following (C-shell) example — this allows the runtime linker to locate the shared libraries:

```
% setenv LD_LIBRARY_PATH /opt/MEIKOcs2/lib:$LD_LIBRARY_PATH
```

Notes for Users of the SunPro F77 Compiler

When using the SunPro Fortran77 compiler the `-R` option as described above will not work. You may either set the environment variable `LD_RUN_PATH` to identify the Meiko library directory (this must be done before you execute your compiler driver) or you can use the compiler driver's `-R` option with both the Meiko and the SunPro library directories specified:

```
-I/opt/MEIKOcs2/inlucde -L/opt/MEIKOcs2/lib \
-R/opt/MEIKOcs2/lib:/opt/SUNWspro/lib \
-lmpsc_host -lmpsc -lew -lelan
```

Tracing

To use the version of the library which produces ParaGraph compatible trace files you should link with `-lmpsc_pt` in addition to `-lmpsc`. Your attention is drawn to the following sections which describe environment variables that are applicable to tracing, and also the tracing functions.

For node programs compile with the following libraries:

```
-lmpsc_pt -lmpsc -lew -lelan
```

Host programs are compiled with the following libraries:

```
-lmpsc_host -lmpsc_pt -lmpsc -lew -lelan
```

Debugging

There is also a debugging version of the library available, which attempts to provide more security and better error behaviour, it will however execute slower than the standard version. This is available as `-lmpsc_dbg` which should be linked instead of `-lmpsc`.

For node programs compile with the following libraries:

```
-lmpsc_debug -lew -lelan
```

Host programs are compiled with the following libraries:

```
-lmpsc_host -lmpsc_debug -lew -lelan
```

Environment Variables

A hosted application that uses `load()` to spawn the node processes identifies your preferred resource requirements from the following environment variables:

Variable	Description
`RMS_PARTITION`	The name of your preferred partition. If you fail to set this variable your node processes are executed on the default partition specified by your System Administrator.
`RMS_NPROCS`	The number of node processes. If you fail to set this variable your node processes are executed on all nodes in the partition.
`RMS_BASEPROC`	Id of the first processor within the partition that will host the node process; usually the first processor in the partition (logical id 0) is used, or the first available processor.
`RMS_VERBOSE`	Set level of status reporting.
`RMS_MEMORY`	The minimum memory requirements for each process, suffixed by K or M (for kilobytes and megabytes respectively).
`RMS_CORESIZE`	Enable core dumping if this variable is set.

The following environment variables are also used by this library; many are inherited from the Elan Widget library:

Variable	Description
`LIBMPSC_TRACEFILE`	For use with `libmpsc_pt` only, this variable specifies the name of the trace file; each node outputs to `$LIBMPSC_TRACEFILE`.*nodeno*. Default name is `LIBMPSC_TRACE`.*nodeno*.
`LIBMPSC_TRACEBUF`	For use with `libmpsc_pt` only, this variable specifies the number of events to allow in the trace buffer.

Variable	Description
LIBEW_WAITTYPE	Specifies how the low level Elan widget library (libew) routines wait for Elan events; either POLL or WAIT, default is to POLL.
LIBEW_DMATYPE	Specifies the type of DMA transfer used by the low level Elan widget library (libew). Either NORMAL or SECURE.
LIBEW_DMACOUNT	Specifies the permitted retry count for DMA transfers. Default is 1.
LIBEW_GROUP_BUFSIZE	Used by global operations such as gsum(). Specifies the buffer size used for communications between processes in a group. The default is 8192 bytes.
LIBEW_GROUP_BRANCH	Used by global operations such as gsum(). Specifies the branching ratio used for the processes in a group. Default is 2.
LIBEW_GROUP_HWBCAST	Used by global operations such as gsum(). Specifies that the Elan communications processor's broadcast hardware is to be used for message broadcasts within the group. May be set to 0 (false) or 1 (true). Default is 1.
LIBEW_TPORT_SMALLMSG	Default small message size used by send and receive functions. Default value is 4096 bytes.
LIBEW_RSYS_ENABLE	Enables the remote system call server; when enabled stdin, stdout, and stderr are routed through the host process. May be either 0 (disabled) or 1 (enabled), default is 1.
LIBEW_RSYS_BUFSIZE	The buffer size used by the remote system call server. Default is 8192 bytes.

Variable	Description
`LIBEW_RSYS_SERVER`	Virtual process ID of the processor that will run the system call server.
`LIBEW_CORE`	Enables core dump on exception. Values may be 1 (enabled) or 0 (disabled). By default core dumping is disabled.
`LIBEW_TRACE`	Enables a trace dump on exception. Values may be 1 (enabled) or 0 (disabled). By default trace dumping is disabled.

Program Tracing

Both ParaGraph and Alog/Upshot are supported for program tracing.

ParaGraph

Three functions in the low level Elan Widget library (`libew`) are applicable to program tracing — these are `ew_ptraceStart()`, `ew_ptraceStop()`, and `ew_ptraceFlush()`. None of these take arguments and none return values to the caller.

Programs that are traced must be linked with `libmpsc_pt` as described in an earlier section. The resulting trace file may be analysed with ParaGraph.

`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

Full documentation for the tracing functions is included in the Elan Widget Library reference manual.

Alog/Upshot

As an alternative to ParaGraph the event/state display tool `upshot` is also supported. To use this you need to instrument your code with trace points. Details may be found in `/opt/MEIKOcs2/upshot/README-MEIKO`.

Tagged Message Passing 2

The following message passing functions are defined within the `libmpsc` library (the global operation functions are listed in Chapter 3).

Initialisation

`mpsc_init()`	Initialisation function.
`mpsc_fini()`	Finalisation function.

Information

`myhost()`	Obtain node ID of the calling process.
`mynode()`	Obtain node ID of the process.
`mypid()`	Obtain node operating system process ID.
`nodedim()`	Obtain cube dimensions.
`numnodes()`	Obtain node count for cube.

Message Passing

`cprobe()`	Wait for a message.
`cprobex()`	Wait for a message (extended).
`crecv()`	Receive a message.
`crecvx()`	Receive a message (extended).
`csend()`	Send a message and wait for it to depart.
`csendrecv()`	Send a message and block until replied.
`gsendx()`	Send a message and wait for departure (extended).
`infocount()`	Determine length of received message.
`infonode()`	Determine node ID of sending process.
`infopid()`	Determine process ID of sending process.
`infotype()`	Determine type of received message.
`iprobe()`	Determine if message is pending.
`iprobex()`	Determine if message is pending (extended).
`irecv()`	Receive a message.
`irecvx()`	Receive a message (extended).
`isend()`	Send a message.
`isendrecv()`	Send message and setup for reply.
`msgdone()`	Determine if non-blocking transaction is complete.
`msgwait()`	Wait for completion of non-blocking transaction.

Miscellaneous

`led()`	Set front panel LEDs.
`flick()`	No-Op — included for portability.

`gray()`	Gray code.
`mclock()`	Elapsed time in ms since `mpsc_init()`.
`ginv()`	Inverse Gray code.

cprobe() — Wait for a message

Synopsis

```
SUBROUTINE CPROBE(type)
INTEGER type
```

Synopsis

```
void cprobe(int type);
```

Arguments

`type` Specifies the type of message you are waiting for. The following values for `type` are valid:

- If `type` is a non-negative integer then a specific message type will be recognised.
- If `type` is -1 then the next message will be recognised, regardless of type.
- If `type` is any negative number other than -1 then an exception is generated.

Description

`cprobe()` blocks the calling process until a message of the selected type is available to be received. When `cprobe()` returns you can use `crecv()` or `irecv()` to initiate the receipt of the message.

Notes:

- The message type is specified by the sender (either `csend()` or `isend()`).
- Use the *info* functions to get more information about a received message (such as its length or the ID of the sender).
- Use `iprobe()` and not `cprobe()` if you do not wish to block the process while waiting for a message.

cprobex() **Wait for a message (extended)**

Synopsis

```
SUBROUTINE CPROBEX(type, sender, ptype, info)
INTEGER type, sender, ptype, info(8)
```

Synopsis

```
void cprobex(int type,int sender,int ptype,int* info);
```

Arguments

`type` Specifies the type of message you are waiting for. The following values for `type` are valid:

- If `type` is a non-negative integer then a specific message type will be recognised.
- If `type` is -1 then the next message will be recognised, regardless of type.
- If `type` is any negative number other than -1 then an exception is generated.

`sender` Specifies the source (sending node) of the message you are waiting for. The following values are valid:

- If `sender` is a non-negative integer then the message must have been sent by this node.
- If `sender` is -1 then the message may have been sent by any node
- If `sender` is negative and not -1 then an exception is generated.

`ptype` Specifies the process type of the sender. Values other than 0 or -1 will cause an exception (there is only one per process per node in this implementation).

`info` Returns the values that are normally returned by the additional `infonode()`, `infocount()`, and `infotype()` functions. The first element of `info` contains the message type. The second element of `info` contains the message length. The third element of `info` contains the node number of the sender.

Description

`cprobex()` is the same as `cprobe()` but allows selection by source and returns additional information that `cprobe()` does not (and requires additional use of the *info* functions to obtain).

Warning – The *info* functions should not be used after `cprobex()` as the relevant data has already been returned to you.

---

crecv() **Receive a message**

Synopsis

```
SUBROUTINE CRECV(type, buf, len)
INTEGER type
INTEGER buf(*)
INTEGER len
```

Synopsis

```
void crecv(int type, void* buf, int len);
```

Arguments

`buf` Identifies the buffer where the received message will be stored.

`len` Specifies the length of the message buffer in bytes.

`type` Specifies the type of message you are waiting for. The following values for type have the meanings shown:

- If `type` is a non-negative integer then a specific message type will be recognised.
- If `type` is -1 then the next message will be recognised, regardless of type.
- If `type` is any negative number other than -1 then an exception is generated.

Description

This function is used to initiate the receipt of a message. The calling process is blocked until a message of the appropriate type is received. The received message is stored in the buffer `buf`.

Notes:

- Use the *info* functions to obtain more information about a received message (such as its length or the ID of the sender).
- Use `irecv()` when you do not want the calling process to block.

crecvx()

Receive a message (extended)

Synopsis

```
SUBROUTINE CRECVX(type, buf, len, sender, ptype, info)
INTEGER type, len, sender, ptype
INTEGER buf(*)
INTEGER info(8)
```

Synopsis

```
void crecvx(int type, void* buf, int len, int sender,
            int ptype, int* info);
```

Arguments

`buf` Identifies the buffer where the received message will be stored.

`len` Specifies the length of the message buffer in bytes.

`type` Specifies the type of message you are waiting for. The following values for type have the meanings shown:

- If `type` is a non-negative integer then a specific message type will be recognised.
- If `type` is -1 then the next message will be recognised, regardless of type.
- If `type` is any negative number other than -1 then an exception is generated.

`sender` Specifies the source (sending node) of the message you are waiting for. The following values are valid:

- If `sender` is a non-negative integer then the message must have been sent by this node.
- If `sender` is -1 then the message may have been sent by any node
- If `sender` is negative and not -1 then an exception is generated.

`ptype` Specifies the process type of the sender. Values other than 0 or -1 will cause an exception (only 1 process per node in this implementation).

`info` Returns the values that are normally returned by the additional `infonode()`, `infocount()`, and `infotype()` functions. The first element of `info` contains the message type. The second element of `info` contains the message length. The third element of `info` contains the node number of the sender.

Description

This function is the same as `crecv()` but allows selection by source and returns additional information that `crecv()` does not (and requires additional use of the *info* functions to obtain).

Warning – The *info* functions should not be used after `crecvx()` as the relevant data has already been returned to you.

---

csend() — Send a message and wait for it to depart

Synopsis

```
SUBROUTINE CSEND(type, buf, len, node, pid)
INTEGER type
INTEGER buf(*)
INTEGER len, node, pid
```

Synopsis

```
void csend(int type, void* buf, int len,
           int node, int pid);
```

Arguments

`type` Specifies the type of message that is being sent. It is recommended that you use values in the range 0 to 999,999,999. Unpredictable results occur if types outside the specified range are used.

`buf` Identifies the buffer that contains the message.

`len` Specifies the size of the message (in bytes).

`node` Specifies the recipient's node ID. If this variable contains a positive integer then the message is sent to that node. Nodes within a cube domain are numbered from 0; use of a node number that is greater than the highest node in the cube causes an error. If node ID is set to -1 the message is broadcast to all nodes.

`pid` Specifies the recipient's process ID. If a global send specifies its own ID then the sender does not receive the message. If an alternative ID is specified the sending node always receives the message.

Description

This function sends a message to a process and causes the sender to block until it is sent. Completion of this function does not indicate that the message arrived at its destination, although it does imply that the sender's message buffer is available for reuse.

csendrecv() — Send a message and block until replied

Synopsis

```
INTEGER FUNCTION CSENDRECV(type, sbuf, slen, tonode,
                           topid, rtype, rbuf, rlen)
INTEGER type, rtype
INTEGER sbuf(*), rbuf(*)
INTEGER slen, tonode, topid, rlen
```

Synopsis

```
int csendrecv(int type, void* sbuf, int slen,
              int tonode, int topid, int rtype,
              void* rbuf, int rlen);
```

Arguments

`type` This specifies the type of the message that is being sent. It is recommended that you use values in the range 0 to 999,999,999. Unpredictable results occur if types outside the specified range are used.

`sbuf` Specifies the source buffer.

`slen` Specifies the size of message to be sent from `sbuf`, in bytes.

`tonode` Specifies the ID of the recipient node.

`topid` Specifies the ID of the recipient process. Negative IDs are reserved for system programs and should not be used.

`rtype` Specifies the reply message type. The following values are permitted:

- If `type` is a non-negative integer then a specific type of message will be recognised.
- If `type` is -1 then the next message will be recognised, regardless of type.
- If `type` is any negative number other than -1 then an exception is generated.

`rbuf` Specifies the buffer that will receive the reply message.

`rlen` Specifies the size of the receive buffer in bytes.

Description

This function is used to send a message and to simultaneously post a receive; the calling process is blocked until the reply is received. When a reply matching the specified reply type (`rtype`) is received it is stored in `rbuf` and the calling process resumes execution.

Notes:

- This function is intended for use with remote procedure calls (a sender posts a request for information and a server returns a result).
- Use `isendrecv()` if you do not want the calling process to block while waiting for the reply.
- Use the *info* functions to obtain information about the received message (such as its length or the ID of the sender).

flick()	**No operation**
Synopsis	`SUNBROUTINE FLICK()`
Synopsis	`void flick(void);`
Description	This function is a no-op; it is included for portability.

gray() — Gray code

Synopsis

```
INTEGER FUNCTION GRAY(val)
INTEGER val
```

Synopsis

```
int gray(int val);
```

Description

Returns the Gray code of the integer argument `val`. It converts integers which differ by 1 to integer which differ by a power of 2.

The table below enumerates the function for small binary integers.

n	gray(n)
0	0
1	1
10	11
11	10
100	110
101	111
110	101

ginv() — Inverse Gray code

Synopsis

```
INTEGER FUNCTION GINV(val)
INTEGER val
```

Synopsis

```
int ginv(int val);
```

Description

Returns the inverse Gray code; this function is the inverse of the `gray()` function.

gsendx() — Send a message to many nodes and wait for it to depart

Synopsis

```
SUBROUTINE GSENDX(type, buf, len, nodes, nnodes)
INTEGER type
INTEGER buf(*)
INTEGER len
INTEGER nnodes, nodes(nnodes)
```

Synopsis

```
void gsendx(int type, void* buf, int len, int* nodes,
            int nnodes);
```

Arguments

`type`	Specifies the type of message you are sending.
`buf`	Identifies the buffer that contains the message.
`len`	Specifies the length of the message in bytes.
`nodes`	Contains a set of node numbers to which data is sent.
`nnodes`	The number of node numbers in `nodes`.

Description

`gsendx()` sends a message to each of the nodes specified by the `nodes` array. The messages are sent by `csend()`, so `gsendx()` is functionally equivalent to the C program:

```
for (i=0; i<nnodes; i++)
   csend(type, buf, len, nodes[i],0);
```

infocount()/node()/pid()/type()Get message information

Synopsis

```
INTEGER FUNCTION INFOCOUNT()
INTEGER FUNCTION INFONODE()
INTEGER FUNCTION INFOPID()
INTEGER FUNCTION INFOTYPE()
```

Synopsis

```
int infocount(void);
int infonode(void);
int infopid(void);
int infotype(void);
```

Description

These functions return information about a received message. The returned value is undefined unless it follows a `recv()`, `sendrecv()`, `probe()`, `msg-done()`, or `msgwait()`.

`infocount()`	Returns the length of the message (in bytes).
`infonode()`	Returns the node ID of the sending process.
`infopid()`	Returns the PID of the sending process.
`infotype()`	Returns the type of message.

Warning – These functions will not return the expected results if used after an extended operation (`cprobex()`, `iprobex()`, `crecvx()`, or `irecvx()`).

iprobe() — Determine if message is present

Synopsis

```
INTEGER FUNCTION IPROBE(type)
INTEGER type
```

Synopsis

```
int iprobe(int type);
```

Arguments

`type` Specifies the type of message you are waiting for. The following values for type are valid:

- If `type` is a non-negative integer then a specific message type will be recognised.
- If `type` is -1 then the next message will be recognised, regardless of type.
- If `type` is any negative number other than -1 then an exception is generated.

Description

This function determines if a message of the specified type is ready for receipt. If a suitable message is ready `iprobe()` returns a value of 1; if no suitable message is ready the function returns 0. When a value of 1 is returned, the *info* functions can be used to obtain information about the message.

This function does not block the calling process; use `cprobe()` if the calling process must be blocked until a suitable message arrives.

iprobex() — Determine if a message is present (extended)

Synopsis

```
INTEGER FUNCTION IPROBEX(type, sender, ptype, info)
INTEGER type, sender, ptype, info(8)
```

Synopsis

```
int iprobex(int type, int sender, int ptype, int* info);
```

Arguments

`type` Specifies the type of message you are waiting for. The following values for `type` are valid:

- If `type` is a non-negative integer then a specific message type will be recognised.
- If `type` is -1 then the next message will be recognised, regardless of type.
- If `type` is any negative number other than -1 then an exception is generated.

`sender` Specifies the source (sending node) of the message you are waiting for. The following values are valid:

- If `sender` is a non-negative integer then the message must have been sent by this node.
- If `sender` is -1 then the message may have been sent by any node
- If `sender` is negative and not -1 then an exception is generated.

`ptype` Specifies the process type of the sender. Values other than 0 or -1 will cause an exception (only 1 process per node in this implementation).

`info` Returns the values that are normally returned by the additional `infonode()`, `infocount()`, and `infotype()` functions. The first element of `info` contains the message type. The second element of `info` contains the message length. The third element of `info` contains the node number of the sender. **Note: the info array is only modified if the iprobex() was successful (and returned 1).**

Description

`iprobex()` is the same as `iprobe()` but allows selection by source and returns additional information that `iprobe()` does not (and requires additional use of the *info* functions to obtain).

Warning – The *info* functions should not be used after `iprobex()` as the relevant data has already been returned to you.

Warning – The `info` array is only modified if the `iprobex()` was successful (and returned 1).

---

irecv() **Receive a message**

Synopsis

```
INTEGER FUNCTION IRECV(type, buf, len)
INTEGER type
INTEGER buf(*)
INTEGER len
```

Synopsis

```
int irecv(int type, void* buf, int len);
```

Arguments

`buf` Specifies the buffer where the received message will be stored.

`len` Specifies the length of the message buffer in bytes.

`type` Specifies the type of message you are waiting for. The following values for type are valid:

- If `type` is a non-negative integer then a specific message type will be recognised.
- If `type` is -1 then the next message will be recognised, regardless of type.
- If `type` is any negative number other than -1 then an exception is generated.

Description

This function allows the caller to setup message buffers for an incoming message, but does not force the caller to wait for the message to arrive. `irecv()` returns a message ID immediately it is called. This message ID is used in subsequent calls to `msgwait()` or `msgdone()` to determine if the message has actually arrived. The message ID is a positive integer greater than 0.

Use the similar function `crecv()` if you want the calling process to block while it waits for the message to arrive.

irecvx() — Receive a message (extended)

Synopsis

```
INTEGER FUNCTION IRECVX(type, buf, len, sender,
                        ptype, info)
INTEGER type, len, sender, ptype
INTEGER buf(*)
INTEGER info(8)
```

Synopsis

```
int irecvx(int type, void* buf, int len, int sender, int
           ptype, int* info);
```

Arguments

`buf` Identifies the buffer where the received message will be stored.

`len` Specifies the length of the message buffer in bytes.

`type` Specifies the type of message you are waiting for. The following values for type have the meanings shown:

- If `type` is a non-negative integer then a specific message type will be recognised.
- If `type` is -1 then the next message will be recognised, regardless of type.
- If `type` is any negative number other than -1 then an exception is generated.

`sender` Specifies the source (sending node) of the message you are waiting for. The following values are valid:

- If `sender` is a non-negative integer then the message must have been sent by this node.
- If `sender` is -1 then the message may have been sent by any node
- If `sender` is negative and not -1 then an exception is generated.

`ptype` Specifies the process type of the sender. Value other than 0 or -1 will cause an exception (only 1 process per node in this implementation).

`info` Returns the values that are normally returned by the additional `infonode()`, `infocount()`, and `infotype()` functions. The first element of `info` contains the message type. The second element of `info` contains the message length. The third element of `info` contains the node number of the sender.

Description

This function is the same as `irecv()` but allows selection by source and returns additional information that `irecv()` does not (and requires additional use of the *info* functions to obtain).

Warning – The *info* functions should not be used after `irecvx()` as the relevant data has already been returned to you.

Warning – The `info` argument only contains valid results after a successful `msgdone()` or `msgwait()` on the message id returned by `irecvx()`.

---

isend() — **Send a message**

Synopsis

```
INTEGER FUNCTION ISEND(type, buf, len, node, pid)
INTEGER type
INTEGER buf(*)
INTEGER len, node, pid
```

Synopsis

```
int isend(int type, void* buf, int len,
          int node, int pid);
```

Arguments

`type` Specifies the type of message that is being sent. It is recommended that you use values in the range 0 to 999,999,999. Unpredictable results occur if types outside the specified range are used.

`buf` Specifies the buffer that contains the message. The data type of the send and receive buffer should be the same.

`len` Specifies the size of the message in bytes.

`node` Specifies the recipient's node ID. Nodes within a partition are numbered from 0. Use of a node number that is greater than the highest node in the partition (or is negative) causes an error.

`pid` Specifies the recipient's process ID. If a global send (broadcast) specifies its own ID then the sender does not receive the message. If an alternative ID is specified the sending node always receives the message.

Description

This function initiates a message transmission to a process but does not wait for the transmission to complete before returning to the caller. `isend()` returns a message ID that may be passed to `msgdone()` or `msgwait()` to determine the status of the transmission. The message ID is a positive integer greater than 0.

You should use the similar function, `csend()`, if you want the calling process to block until the message has been sent.

isendrecv() — **Send a message and setup for reply**

Synopsis

```
INTEGER FUNCTION ISENDRECV(type, sbuf, slen, tonode,
                                 topid, rtype, rbuf, rlen)
INTEGER type, rtype
INTEGER sbuf(*), rbuf(*)
INTEGER slen, tonode, topid, rlen
```

Synopsis

```
int isendrecv(int type, void* sbuf, int slen, int
              tonode, int topid, int rtype,
              void* rbuf, int rlen);
```

Arguments

`type` Specifies the type of message that is being sent. It is recommended that you use values in the range 0 to 999,999,999. Unpredictable results occur if types outside the specified range are used.

`sbuf` Specifies the source buffer that contains the message.

`slen` Specifies the size of message, in bytes, to be sent from `sbuf`.

`tonode` Specifies the ID of the recipient node.

`topid` Specifies the ID of the recipient process. Negative IDs are reserved for system programs and should not be used.

`rtype` Specifies the types of reply message:

If `type` is a non-negative integer then a specific message type will be recognised.

If `type` is -1 then the next message will be recognised, regardless of type.

If `type` is any negative number other than -1 then an exception message is generated.

`rbuf` Specifies the buffer that will receive the reply message.

`rlen` Specifies the size, in bytes, of the receive buffer.

Description

This function is used to send a message and to simultaneously post a receive for the reply. When a reply with the specified type (`rtype`) is received it is stored in the buffer that is identified by `rbuf`.

The calling process is not blocked during this transaction. `isendrecv()` returns a message ID that may be passed to `msgdone()` or `msgwait()` to determine the status of the transfer.

Notes:

- This function is intended for use with remote procedure calls.
- If you want the calling process to block while waiting for the reply, use `csendrecv()`.
- Use the *info* functions to get information about the received message (its size and the sender ID, for example).

led() — Set front panel LEDs

Synopsis

```
INTEGER FUNCTION LED(ipat)
INTEGER ipat
```

Synopsis

```
int led(int pattern);
```

Description

Sets the LEDs on the node to the specified pattern. The bits that are used are hardware dependent.

The return value is the previous setting of the LEDs, which can be used to restore the old pattern.

mclock() — **Elapsed time.**

Synopsis

```
INTEGER FUNCTION MCLOCK()
```

Synopsis

```
int mclock(void);
```

Description

This function returns the elapsed time, in milliseconds, since the execution of the initialisation function `mpsc_init()`.

mpsc_init()	Initialisation function
Synopsis	`SUBROUTINE MPSCINIT()`
Synopsis	`void mpsc_init(void);`
Description	Initialisation function. Each process must call this function before any other function in the `libmpsc` library.

mpsc_fini() — Finalisation function

Synopsis

```
SUBROUTINE MPSCFINI()
```

Synopsis

```
void mpsc_fini(void);
```

Description

Optional finalisation function.

msgdone() — **Test for completion of non-blocking transaction**

Synopsis

```
INTEGER FUNCTION MSGDONE(id)
INTEGER id
```

Synopsis

```
int msgdone(int id);
```

Arguments

`id` The ID that is returned by `isend()`, `irecv()`, or `irecvx()`.

Description

Use this function to determine if an `isend()`, `irecvx()`, or `irecv()` transaction has completed. `msgdone()` returns 1 when the `isend()` buffer is available for reuse (the message has gone) or when the `irecv()`/`irecvx()` buffer contains a message of the appropriate type.

Note that the message ID is cleared after `msgdone()` has returned a value of 1. Subsequent uses of that ID are no longer valid.

A value of 0 is returned if the transaction is not complete. You may repeatedly use `msgdone()` with the same ID until completion has been signalled.

msgwait() — Wait for completion of non-blocking transaction

Synopsis

```
INTEGER FUNCTION MSGWAIT(id)
INTEGER id
```

Synopsis

```
int msgwait(int id);
```

Arguments

`id` The ID that is returned by `isend()`, `irecv()`, `irecvx()`.

Description

Use this function to wait until an `isend()`, `irecvx()` or `irecv()` transaction has completed. The calling process is blocked until the transfer is complete. When `msgwait()` returns control to the process, thus signalling completion, the message ID is cleared and no longer valid.

When the message transfer is complete the `isend()` buffer is available for reuse (the message has gone), and the `irecv()`/`irecvx()` buffer contains a message of the appropriate type.

myhost() — **Obtain node ID of calling process**

Synopsis

```
INTEGER FUNCTION MYHOST()
```

Synopsis

```
int myhost(void);
```

Description

Returns the node ID for the host process. The return value will be -2 if there is no host process. (This will ensure that a program that executes code like:

```
csend(?,?,?, myhost(), ?);
```

will abort when there is no host, rather than send a message to a valid node.)

mynode() — Obtain node ID of the process

Synopsis

```
INTEGER FUNCTION MYNODE()
```

Synopsis

```
int mynode(void);
```

Description

This function returns the node ID for this process.

mypid() **Obtain OS process ID**

Synopsis

```
INTEGER FUNCTION MYPID()
```

Synopsis

```
int mypid(void);
```

Description

This function returns the process ID for this process (always 0).

nodedim() **Obtain cube dimensions**

Synopsis

```
INTEGER FUNCTION NODEDIM()
```

Synopsis

```
int nodedim(void);
```

Description

Returns the dimension of the allocated cube. The dimension of a 64 node cube is 6 because $2^6 = 64$. Use `numnodes()` to return the number of nodes.

Warning – This function will cause an exception if the number of nodes is not a power of 2.

---

numnodes() **Obtain node count for cube**

Synopsis

```
INTEGER FUNCTION NUMNODES()
```

Synopsis

```
int numnodes(void);
```

Description

Returns the number of nodes in the allocated cube. Use `nodedim()` to obtain the cube dimension.

In a host program prior to `load()`, `numnodes()` will return:

1. the number of nodes allocated by the `allocate` command if an allocation is in effect.
2. the number of nodes which were allocated by `mpsc_getnodes()`.
3. the value 0 (no pre-allocation, and no nodes yet loaded).

After `load()` (and therefore at all times in the node programs) `numnodes()` returns the number of nodes which were loaded.

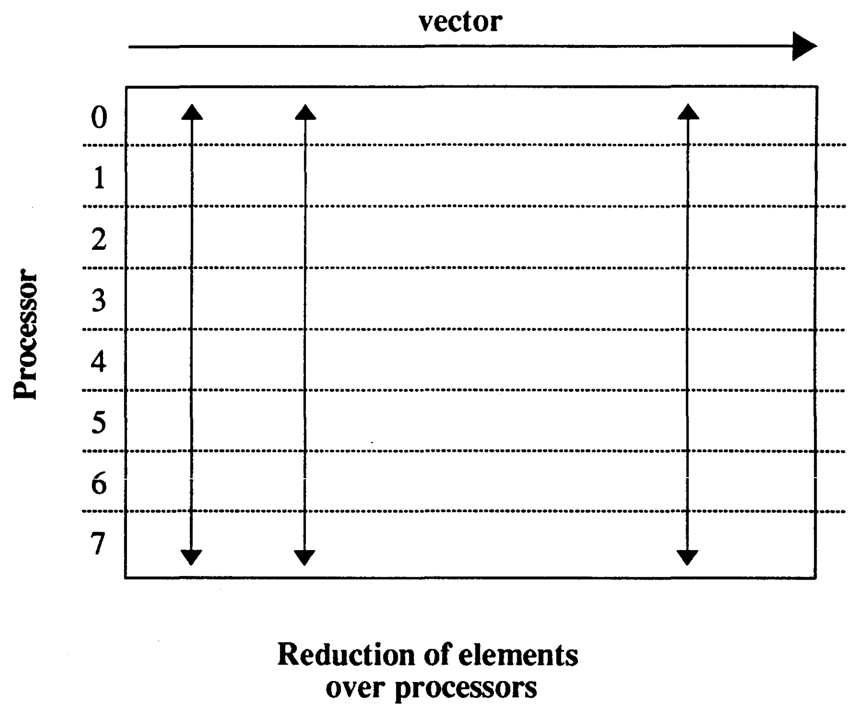
Global Reduction Operations 3

Overview

Global reduction operations take an item of data from each processor in the machine, combine them according to some function, and return the result to all processors. Execution continues when all processors have called the global operation, communicated their data, and returned.

Global operations implement a series of communication and calculation actions more efficiently than the equivalent use of explicit message passing and calculation functions. The global operations are also synchronised so that none may begin its calculations until the others are ready.

Figure 3-1 Vectors Distributed Over 7 Processors



Example — gdsum()

`gdsum()` takes a vector of double precision numbers from each processor, and returns to each processor a vector of sums. If `gdsum()` is called with a vector of 4 doubles then the result is also a vector of four doubles, each the sum over the processors of successive elements. In the example below, the vector `v` is both the source and destination operand; the parameter `work` is not used.

The results vector `v[]` is the same after:

```
gdsum(v[1], 4, work)
```

as it is after:

```
gdsum(v[1], 1, work)
gdsum(v[2], 1, work)
gdsum(v[3], 1, work)
gdsum(v[4], 1, work)
```

The latter is slower because it requires four times the number of system calls and message transfers. The message length for the first method will be longer, of course, but the increased transmission time will be insignificant for small vectors.

Function List

The following functions are defined within the `libmpsc` library:

`gdhigh()`	Global vector double precision Maximum operation.
`gdlow()`	Global vector double precision Minimum operation.
`gdprod()`	Global vector double precision Multiply.
`gdsum()`	Global vector double precision Sum.
`giand()`	Global vector integer bitwise AND.
`gihigh()`	Global vector integer Maximum operation.
`gilow()`	Global vector integer Minimum operation.
`gior()`	Global vector integer bitwise OR.
`giprod()`	Global vector integer Multiply.
`gisum()`	Global vector integer Sum.
`gixor()`	Global vector integer bitwise XOR.
`gland()`	Global vector logical AND.

`glor()`	Global vector logical OR.
`glxor()`	Global vector logical XOR.
`gshigh()`	Global vector real Maximum operation.
`gslow()`	Global vector real Minimum operation.
`gsprod()`	Global vector real Multiply.
`gssum()`	Global vector real Sum.
`gsync()`	Global synchronisation.

gdhigh(), gihigh(), gshigh() — Global Maximum operation

Synopsis

```
SUBROUTINE GDHIGH(x, n, work)
DOUBLE PRECISION x(n)
INTEGER n
DOUBLE PRECISION work(n)

SUBROUTINE GIHIGH(x, n, work)
INTEGER x(n)
INTEGER n
INTEGER work(n)

SUBROUTINE GSHIGH(x, n, work)
REAL x(n)
INTEGER n
REAL work(n)
```

Synopsis

```
void gdhigh(double* x, int n, double* work);

void gihigh(int* x, int n, int* work);

void gshigh(float* x, int n, float* work);
```

Arguments

`x`	The input vector (or scalar). This vector will contain the result when the function completes.
`n`	The number of elements in the input array.
`work`	Not used; included for compatibility.

Description

These functions calculate the maximum of `x` across all nodes. The result is returned in `x` to every node.

gdlow(), gilow(), gslow() — Global Minimum operation

Synopsis

```
SUBROUTINE GDLOW(x, n, work)
DOUBLE PRECISION x(n)
INTEGER n
DOUBLE PRECISION work(n)

SUBROUTINE GILOW(x, n, work)
INTEGER x(n)
INTEGER n
INTEGER work(n)

SUBROUTINE GSLOW(x, n, work)
REAL x(n)
INTEGER n
REAL work(n)
```

Synopsis

```
void gdlow(double* x, int n, double* work);

void gilow(int* x, int n, int* work);

void gslow(float* x, int n, float* work);
```

Arguments

`x`	The input vector (or scalar). This vector will contain the result when the function completes.
`n`	The number of elements in the input array.
`work`	Not used; included for compatibility.

Description

These functions calculate the minimum of `x` across all nodes. The result is returned in `x` to every node.

gdprod(), giprod(), gsprod() Global multiply operation

Synopsis

```
SUBROUTINE GDPROD(x, n, work)
DOUBLE PRECISION x(n)
INTEGER n
DOUBLE PRECISION work(n)

SUBROUTINE GIPROD(x, n, work)
INTEGER x(n)
INTEGER n
INTEGER work(n)

SUBROUTINE GSPROD(x, n, work)
REAL x(n)
INTEGER n
REAL work(n)
```

Synopsis

```
void gdprod(double* x, int n, double* work);

void giprod(int* x, int n, int* work);

void gsprod(float* x, int n, float* work);
```

Arguments

`x`	The input vector (or scalar). This vector will contain the result when the function completes.
`n`	The number of elements in the input array.
`work`	Not used; included for compatibility.

Description

These functions calculate the product of x across all nodes. The result is returned in x to every node.

gdsum(), gisum(), gssum() — Global sum operation

Synopsis

```
SUBROUTINE GDSUM(x, n, work)
DOUBLE PRECISION x(n)
INTEGER n
DOUBLE PRECISION work(n)

SUBROUTINE GISUM(x, n, work)
INTEGER x(n)
INTEGER n
INTEGER work(n)

SUBROUTINE GSSUM(x, n, work)
REAL x(n)
INTEGER n
REAL work(n)
```

Synopsis

```
void gdsum(double* x, int n, double* work);

void gisum(int* x, int n, int* work);

void gssum(float* x, int n, float* work);
```

Arguments

`x`	The input vector (or scalar). This vector will contain the result when the function completes.
`n`	The number of elements in the input array.
`work`	Not used; included for compatibility.

Description

These functions calculate the sum of `x` across all nodes. The result is returned in `x` to every node.

giand(), gland() — Global AND operation

Synopsis

```
SUBROUTINE GIAND(x, n, work)
INTEGER x(n)
INTEGER n
INTEGER work(n)

SUBROUTINE GLAND(x, n, work)
LOGICAL x(n)
INTEGER n
LOGICAL work(n)
```

Synopsis

```
void giand(int* x, int n, int* work);

void gland(int* x, int n, int* work);
```

Arguments

`x`	The input vector (or scalar). This vector will contain the result when the function completes.
`n`	The number of elements in the input array.
`work`	Not used; included for compatibility.

Description

These functions calculate the bitwise (`giand()`) or logical (`gland()`) AND of `x` across all nodes. The result is returned in `x` to every node.

gior(), glor() — Global OR operation

Synopsis

```
SUBROUTINE GIOR(x, n, work)
INTEGER x(n)
INTEGER n
INTEGER work(n)

SUBROUTINE GLOR(x, n, work)
LOGICAL x(n)
INTEGER n
LOGICAL work(n)
```

Synopsis

```
void gior(int* x, int n, int* work);

void glor(int* x, int n, int* work);
```

Arguments

`x`	The input vector (or scalar). This vector will contain the result when the function completes.
`n`	The number of elements in the input array.
`work`	Not used; included for compatibility.

Description

These functions calculate the bitwise (`gior()`) or logical (`glor()`) OR of `x` across all nodes. The result is returned in `x` to every node.

gixor(), glxor() — Global XOR (exclusive-OR) operation

Synopsis

```
SUBROUTINE GIXOR(x, n, work)
INTEGER x(n)
INTEGER n
INTEGER work(n)

SUBROUTINE GLXOR(x, n, work)
LOGICAL x(n)
INTEGER n
LOGICAL work(n)
```

Synopsis

```
void gixor(int* x, int n, int* work);

void glxor(int* x, int n, int* work);
```

Arguments

`x`	The input vector (or scalar). This vector will contain the result when the function completes.
`n`	The number of elements in the input array.
`work`	Not used; included for compatibility.

Description

These functions calculate the bitwise (`gixor()`) or logical (`glxor()`) XOR of `x` across all nodes. The result is returned in `x` to every node.

gsync() — Global synchronisation

Synopsis

```
SUBROUTINE GSYNC()
```

Synopsis

```
void gsync(void);
```

Description

This function synchronises node processes. When a process executes `gsync()` it blocks until all other processes have executed it.

Host Functions 4

The library provides support for a limited set of host functions, which interface to the resource management system to load the node processes. The following functions are only available in the host program.

Host specific functions

`mpsc_getnodes()`	Pre-allocate nodes' processing resource.
`killcube()`	Forcibly terminate all node processes.
`load()`	Start execution of a set of node processes.
`setpid()`	Set the host pid.
`waitall()`	Wait for all node processes to exit.

In addition the host can use any of the functions used on the node apart from the collective communication functions.

Restrictions

The host functions provided are restricted to allowing a single node program to be loaded on all nodes. Only a single pid is permitted (which must be zero).

Note that `getcube()` is not included in this implementation; see the similar function `mpsc_getnodes()`.

mpsc_getnodes() — Pre-allocate nodes' processing resource

Synopsis

```
SUBROUTINE MPSC_GETNODES(request,istatus)
CHARACTER *(*) request
INTEGER istatus
```

Synopsis

```
int mpsc_getnodes(const char* request);
```

Arguments

`istatus` returns 1 on success and 0 on failure.

The `request` argument is a string in which one or more of the following options are concatenated (note the similarity to the `allocate(1)` command):

`-b` *number*	Set the base processor, relative to the start of the partition.
`-i`	Allocate resource immediately; fail if the resource is in use rather than suspending execution until the resource is free.
`-n` *number* | `a`	Ask for `number` processors, or all (`-na`) processors in the partition.
`-p` *partition*	The name of the partition.

Description

This function is used by a host process to allocate resource for the node processes; it is a functional equivalent of `allocate(1)`.

Allocated resource is held by the host process until it terminates and is chargeable to that host for the whole period that it is held; it is also unavailable for use by other user's during the period.

Node processes are spawned onto the allocated resource by the `load(3x)` function. When resources have been pre-allocated `load(3x)` does not attempt to re-allocate the resource, but instead spawns the node processes over the whole of the allocated resource.

The `numnodes(3x)` function can be used by the host process after calling `mpsc_getnodes` (3x) to determine the number of processors that were allocated.

Example

Allocate all the nodes in the `parallel` partition:

```
call mpsc_getnodes("-p parallel -na", istatus)
print *, "Allocated ", numnodes()," from parallel"
call load ("example", -1, 0)
```

Or in C:

```
istatus = mpsc_getnodes("-p parallel -na");
printf("Allocated %d from parallel\n", numnodes());
load("example", -1, 0);
```

See Also

`allocate`(1), `load`(3x), `numnodes`(3x).

killcube() — Forcibly terminate node proceses

Synopsis

```
SUBROUTINE KILLCUBE(node, pid)
INTEGER node
INTEGER pid
```

Synopsis

```
void killcube(const int node, const int pid);
```

Arguments

`node` Specifies the set of nodes to be killed. The only valid value is -1.

`pid` Specifies the pid of the nodes to be killed. The only valid values are zero or -1

Description

`killcube()` sends a **SIGKILL**.signal to all of the node processes in the program and awaits their termination.

Notes:

- `killcube()` can only be used to terminate all nodes simultaneously.

load() — Load an executable image onto the node processors

Synopsis

```
CALL LOAD(exe,node,pid)
CHARACTER*(*) exe
INTEGER node
INTEGER pid
```

Synopsis

```
void load(const char * exe, const int node,
          const int pid);
```

Arguments

`exe`	Specifies the name of the image file to be loaded. This is searched for through the directories in the `PATH` environment variable
`node`	Specifies the set of nodes to be loaded. The only valid value is -1, meaning all nodes
`pid`	Specifies the pid for the processes to be created. The only valid value is zero.

Description

`load()` loads a set of nodes with the given executable and starts them running. The number of nodes chosen and their placement are determined by examining the resource management system environment variables at the time that `load()` is executed, or the resources which have already been allocated.

Relevant environment variables are:

`RMS_PARTITION`	The name of the partition.
`RMS_NPROCS`	The number of processors to be loaded.

Notes:

- The choice of nodes to load can be changed by the host program by using the `putenv()` call to modify the environment variables consulted by the resource management system prior to making the call to load.
- A host process can pre-allocate the nodes' resource by calling `mpsc_getnodes()`. When resources are pre-allocated the subsequent call to `load()` will not attempt to allocate its own resources.

See Also

`mpsc_getnodes`(3x), `allocate`(1).

setpid() | **Set the pid for the host node**

Synopsis

```
CALL SETPID(pid)
INTEGER pid
```

Synopsis

```
void setpid(const int pid);
```

Arguments

`pid` is the process id to be used by the host node. The only valid argument value is zero.

Description

This function is a no-op — it is provided solely for compatibility with other systems which require it to be present.

waitall() — **Allows the host to await termination of the nodes**

Synopsis

```
CALL WAITALL(node, pid)
INTEGER node
INTEGER pid
```

Synopsis

```
void waitall(const int node, const int pid);
```

Arguments

`node` Specifies the set of nodes to wait for; the only valid value is -1, meaning all nodes

`pid` Specifies the pid for the processes to be waited for. The only valid values are zero or -1.

Description

`waitall()` allows the host program to suspend itself until all of the node programs loaded by `load` have finished execution.

Example Programs 5

The programs in `/opt/MEIKOcs2/example/mpsc` describe a C and Fortran version of a simple libmpsc application.

The examples have been coded to illustrate both hosted and hostless programming models and methods of coding that allows the choice of model to be selected at either run-time or compile time. Also illustrated are examples of both blocking and non-blocking communications, global reduction, and global synchronisation.

Compilation

A makefile is included alongside the example programs. Before compiling or editing the example programs you should copy them into your home directory so that your work does not conflict with the work of others:

```
user@cs2 mkdir ~/mpsc
user@cs2 cp /opt/MEIKOcs2/example/mpsc/* ~/mpsc
user@cs2 cd ~/mpsc
```

To compile the C version of the example type:

```
user@cs2: make host htag tag
```

To compile the Fortran version type:

```
user@cs2: make fhost ftag
```

Running the Programs

Hosted applications are started by executing the host directly from you command shell, whereas hostless applications require a loader such as `prun`. This section shows examples of both methods.

Running Hosted Programs

The host process in a libmpsc application liaises with the CS-2 resource management system for the node's processing resource. You specify your resource requirement by setting one or more of the following environment variables:

Variable	Description
RMS_PARTITION	The name of your preferred partition. If you fail to set this variable your node processes are executed on the default partition specified by your System Administrator.
RMS_NPROCS	The number of node processes. If you fail to set this variable your node processes are executed on all nodes in the partition.
RMS_BASEPROC	Id of the first processor within the partition that will host the node process; usually the first processor in the partition (logical id 0) is used, or the first available processor.
RMS_VERBOSE	Set level of status reporting.
RMS_MEMORY	The minimum memory requirements for each process, suffixed by K or M (for kilobytes and megabytes respectively).
RMS_CORESIZE	Enable core dumping if this variable is set.

To specify, for example, that the host process spawns 4 node processes within the `parallel` partition you must set the following two variables before you execute the host process (the following example uses the C-shell):

```
user@cs2: setenv RMS_PARTITION parallel
user@cs2: setenv RMS_NPROCS 4
```

Having specified your resource requirements you start the application by executing the host program from your command shell. The following command line starts the C version of this example:

```
user@cs2: host
```

If you prefer the Fortran example execute `fhost` in place of `host`.

Running Hostless Programs

Hostless applications require a loader program, such as `prun(1)`, to load the node processes into a partition. You can specify your resource requirements by setting the environment variables described above, or you can specify them on `prun`'s command line. The following example uses `prun` to execute 4 processes in the `parallel` partition:

```
muser@cs2: prun -n4 -pparallel tag
```

If you prefer the Fortran example execute `ftag` in place of `tag`.

Description of the Hosted Application

The following sections describe the how the processes are initialised, including the host's interaction with the resource management system, and how they communicate.

Process Initialisation

A hosted application initially consists of just one process — the host. This process begins by calling the initialisation function `mpsc_init()`, which is used to attach the process to the Elan network and to initialise the underlying communication mechanisms (the Widget library TPORTs).

The host process spawns the node processes by calling `load()`. In the Fortran example, where a previous call to `mpsc_getnodes()` is used to pre-allocate the resource, the `load()` function spawns the node processes onto the allocated resource — it does not allocate any resource itself. In the case of the C example, where there is no previous call to `mpsc_getnodes()`, the `load()` function both allocates resource and spawns the node processes.

Note that the `load()` function in this implementation is not passed the number of node processes that are to be spawned; this is determined by either spawning the nodes over all the pre-allocated resource (where `allocate(1)` or `mpsc_getnodes(3x)` have been used) or by the resource management system environment variables.

After spawning the node processes the `load()` function suspends execution of the host until all of the nodes have successfully initialised. Embedded within both `load()` and the nodes' `mpsc_init()` is a barrier synchronisation that prevents the application from continuing until all processes are ready; this barrier synchronisation is a safeguard to ensure that no communications may take place before the underlying communication mechanisms are in place.

Process Communications

Two types of communication are used by the node processes; blocking and non-blocking.

The iterative loop within the node processes uses the non-blocking `isend()`/`irecv()` pair to handle communication between the node processes; use of non-blocking communications allow the node process to continue with useful work (in this case a simple summation) while waiting for the communication to complete. Completion of the communication is tested for by calls to `msgwait()`; this function will delay iteration of the loop until the communications have completed and the send and receive buffers are available for reuse. Note that the message type arguments are always set to 0; we have no interest in the source or the ordering of message in this case.

Communication with the host process is handled by blocking communications. Note that the node processes have been coded to allow their execution without a host process (in the C example the programming model is selected at compile time, in the fortran example the decision can be made at runtime — see later). The communications that are sent to the host are tagged with the sender's node id, which allows the host to receive the messages ordered by the sender's node id.

Global Operations

The node processes include an example of global reduction. Each process passes to `gisum()` a single integer (a vector of 1 element). `gisum()` synchronises all the processes (an implicit barrier synchronisation) and then calculates the sum of the vectors across all nodes. On completion the source vector is overwritten by the result.

Note that `gisum()` must be called by all the node processes; the implicit synchronisation within this function will suspend the calling process until all the node processes have also synchronised.

The example also includes an example of global synchronisation — an example of `gsync()`. This is used to synchronise all the node processes and to prevent any one node process from terminating before its peers have also completed. You can use `gsync()` to synchronise entry to any critical section of code.

Description of the Hostless Application

The hostless example uses the same node processes as the hosted application described above, except that they are loaded into a partition by a loader program, such as `prun`, and not by a libmpsc program.

All the node processes begin execution of `mpsc_init()` at the same time. This function initialises the process's communication mechanisms and includes an implicit barrier, which suspends the caller until all other node processes have also successfully execution their initialisation function.

In the C version of this example the decision to execute the application as a hostless application is made at compile time. Communications with a master process are removed from the source by preprocessor directives, and substituted by output to the console. To compile the program for execution as a hosted application include the `-DHOSTED` option on your compiler driver's command line; remove it for a hostless application. If you study the makefile that is supplied with the examples you will note that the only difference between the `tag` and `htag` targets is the inclusion of this compiler option.

The Fortran example uses a different approach; the model used for this example is selected at runtime by a call to `myhost()`. Here the return value from `myhost()` is compared with the return value from `numnodes()`; if the two values are the same then the node has a host (because the node id of the host will always be the highest node id in the application). A return value of -2 from `myhost()` also signifies that there is no host.

Error Messages 6

Message Format

The functions in the Tagged Message Passing and Global Reduction library (`libmpsc`) are built upon the functions in the Elan Widget library. Errors within `libmpsc` are reported via the Widget library exception handler; this writes diagnostic messages to the standard error device and kills the application.

The format of `libmpsc` messages is:

MPSC EXCEPTION @ *process* : *error_code* (*error_text*)
error message string

The *error message strings* are described later in this chapter. The *process* is the virtual process number of the process that detected the error; if the exception occurs before the process has attached to the network (i.e. before `mpsc_init()` is called) then this is shown as `----`. The *error code* (and its textual equivalent the *error text*) are one of:

Error Code	Error Text
1000	`Initialisation error`
1001	`No more message descriptors`
1002	`Bad pid`
1003	`Bad event`

Error Code	Error Text
1004	`No more dma descriptors`
1005	`Bad Node`
1006	`Invalid argument`
1007	`Bad tag`
1008	`Bad ptype (must be zero)`
1009	`Bad resource request`

Widget Library Exceptions

Functions in `libmpsc` are implemented on functions in the Elan Widget library. When an exception occurs within a Widget library function this is handled by the Widget library's own exception handler. The Widget library handler is similar to that used by `libmpsc` but produces errors in the form:

```
EW_EXCEPTION @ process : error_code (error_text)
error message string
```

These exceptions are fully described in *The Elan Widget Library*, Meiko document number S1002–10M104.

Note for Fortran Programmers

All errors apply to both C and Fortran implementations unless the description specifies a specific language. Often the error message repeats the parameters that were passed to the failed call; these will be the parameters that were passed to the underlying C implementation of the function, and may not be identical to those passed to the Fortran binding.

Error Messages

In the following list italicised text represents context specific text or values.

'*mpsc version*' incompatible with '*elan version*' ('*elan version*' expected)
Error type is 1000 (Initialisation error). Occurs in mpsc_init(); Elan library version incompatibility. This library was linked with an out of date version of `libelan`.

'*mpsc version*' incompatible with '*ew version*' ('*ew version*' expected)
Error type is 1000 (Initialisation error). Occurs in mpsc_init(); Elan Widget library incompatibility. This library was linked with an out of date version of `libew`.

Can't allocate *count* message descriptors
Error type is 1001 (No more message descriptors). Occurs in `irecv`(), `irecvx`(), `isend`(), and `isendrecv`(); a call to `calloc`() failed (insufficient memory). A descriptor is required for each pending non-blocking communication; tried to allocate a batch of additional descriptors for non-blocking communications but was unable. Maybe there are too many outstanding communications, are you clearing them with either `msgdone`() or `msgwait`()?

Can't allocate message port
Error type is 1000 (Initialisation error). Occurs in `load`() (in host processes) and `mpsc_init`() (on node processes); a call to `ew_allocate`()[1] failed, maybe because heap or swap space were exhausted.

cprobe (*type*)
Error type is 1007 (Bad tag). Occurs in `cprobe`(); the message type (*type*) must be greater than -1 in this implementation.

cprobex (*type*, *sender*, *ptype*, *info*)
Error type is 1007 (Bad tag). Occurs in `cprobex`(); the message type (*type*) must be greater than -1 in this implementation.

cprobex (*type*, *sender*, *ptype*, *info*)
Error type is 1008 (Bad ptype (must be zero)). Occurs in `cprobex`(); the process type (*ptype*) must be either 0 or -1 in this implementation.

crecv (*type*, *buf*, *len*)
Error type is 1007 (Bad tag). Occurs in `crecv`(); the message type (*type*) must be greater than -1.

1. ew_allocate() is a Widget library function.

crecvx (*type, buf, len, sender, ptype, info*)
Error type is 1007 (Bad tag). Occurs in `crecvx`(); the message type (*type*) must be greater than -1.

crecvx (*type, buf, len, sender, ptype, info*)
Error type is 1008 (Bad ptype (must be zero)). Occurs in `crecvx`(); the process type (*ptype*) must be 0 or -1 in this implementation.

csend (*type, buf, len, node, pid*)
Error type is 1002 (Bad PID). Occurs in `csend`() (with debugging enabled); the *pid* argument must be 0 in this implementation.

csend (*type, buf, len, node, pid*)
Error type is 1005 (Bad node). Occurs in `csend`(); the *node* argument is out of range; must be either a node id or -1.

csendrecv (*type, sbuf, slen, tonode, topid, rtype, rbuf, rlen*)
Error type is 1002 (Bad PID). Occurs in `csendrecv`() (with debugging enabled); the *pid* argument must be 0 in this implementation.

csendrecv (*type, sbuf, slen, tonode, topid, rtype, rbuf, rlen*)
Error type is 1005 (Bad node). Occurs in `csendrecv`(); the node argument (*tonode*) is out of range — must be a positive integer node id.

csendrecv (*type, sbuf, slen, tonode, topid, rtype, rbuf, rlen*)
Error type is 1007 (Bad tag). Occurs in `csendrecv`(); the reply message type (*rtype*) must be greater than -1.

Hosted MPSC initialised with *count* procs in host segment
Error type is 1000 (Initialisation error). Occurs in `load`(); a hosted MPSC application has been created but there is not 1 process in the host segment. This indicates an internal error that should be reported to Meiko.

Hosted MPSC initialised with *count* segments
Error type is 1000 (Initialisation error). Occurs in `load`(); a hosted MPSC application has been created but not within 2 segments. The host process should be running in a different segment to the node processes. This indicates an internal error that should be reported to Meiko.

iprobe (*type*)
Error type is 1007 (Bad tag). Occurs in `iprobe()`; the message type (*type*) must be greater than -1.

iprobex (*type*, *sender*, *ptype*, *info*)
Error type is 1007 (Bad tag). Occurs in `iprobex()`; the message type (*type*) must be greater than -1.

iprobex (*type*, *sender*, *ptype*, *info*)
Error type is 1008 (Bad ptype (must be zero)). Occurs in `iprobex()`; the process type (*ptype*) must be either 0 or -1 in this implementation.

irecv (*type*, *buf*, *len*)
Error type is 1007 (Bad tag). Occurs in `irecv()`; the message type (*type*) must be greater than -1.

irecvx (*type*, *buf*, *len*, *sender*, *ptype*, *info*)
Error type is 1007 (Bad tag). Occurs in `irecvx()`; the message type (*type*) must be greater than -1.

irecvx (*type*, *buf*, *len*, *sender*, *ptype*, *info*)
Error type is 1008 (Bad ptype (must be zero)). Occurs in `irecvx()`; the process type (*ptype*) must be 0 or -1 in this implementation.

isend (*type*, *buf*, *len*, *node*, *pid*)
Error type is 1002 (Bad PID). Occurs in `isend()` (with debugging enabled); the *pid* argument must be 0 in this implementation.

isend (*type*, *buf*, *len*, *node*, *pid*)
Error type is 1005 (Bad node). Occurs in `isend()`; the *node* argument is out of range.

isendrecv (*type*, *sbuf*, *slen*, *tonode*, *topid*, *rtype*, *rbuf*, *rlen*)
Error type is 1002 (Bad PID). Occurs in `isendrecv()` (with debugging enabled); the *pid* argument must be 0 in this implementation.

isendrecv (*type*, *sbuf*, *slen*, *tonode*, *topid*, *rtype*, *rbuf*, *rlen*)
Error type is 1005 (Bad node). Occurs in `isendrecv()`; the *node* argument (*tonode*) is out of range — must be a positive integer node id.

isendrecv (*type*, *sbuf*, *slen*, *tonode*, *topid*, *rtype*, *rbuf*, *rlen*)
Error type is 1007 (Bad tag). Occurs in `isendrecv()`; the reply message type (*rtype*) must be greater than -1.

killcube (*node*, *pid*) **node must be -1**
Error type is 1005 (Bad node). Occurs in `killcube()`; the *node* argument must be -1 in this implementation.

killcube (*node*, *pid*) **only valid on host**
Error type is 1005 (Bad node). Occurs in `killcube()`; a node process called `killcube()` (only host processes may call this function).

killcube (*node*, *pid*) **pid must be 0**
Error type is 1002 (Bad PID). Occurs in `killcube()`; the pid argument must be set to 0 in this implementation.

load exe name too long
Error type is 1006 (Invalid argument). Occurs in fortran binding for `load()`; an internal limit of 256 exists for the length of the executable's name.

load : no elan capability
Error type is 1006 (Invalid argument). Occurs in `load()`; a call to the Elan Widget library function `ew_getenvCap()` failed which may happen because of insufficient memory.

load (*"prog"*, *node*, *pid*) **node must be -1**
Error type is 1005 (Bad node). Occurs in `load()`; the node argument must be -1 in this implementation.

load (*"prog"*, *node*, *pid*) **pid must be 0**
Error type is 1002 (Bad PID). Occurs in `load()`; the pid argument must be set to 0 in this implementation.

mpsc_checkVersion(self)
Error type is 1000 (Initialisation error). Occurs in `mpsc_init()`; internal incompatibility of library source files.

mpsc_getnodes argument string too long
Error type is 1009 (Bad resource request). Occurs in `mpsc_getnodes()`; there is an internal limit of 256 characters on the resource request string.

mpsc_getnodes(*"resource"*)

Error type is 1009 (Bad resource request). Occurs in `mpsc_getnodes`(); the argument string is not a valid resource request.

nodedim(): invalid number of nodes *count*

Error type is 1006 (Invalid argument). Occurs in `nodedim`(); the number of node processes is not a power of 2.

setpid (*pid*) pid must be 0

Error type is 1002 (Bad PID). Occurs in `setpid`(); the specified pid was not 0. (This function is provided for compatibility only and performs no useful function).

waitall(*node, pid*) node must be -1

Error type is 1005 (Bad node). Occurs in `waitall`(); the node argument must be -1 in this implementation.

waitall (*node, pid*) only valid on host

Error type is 1006 (Invalid argument). Occurs in `waitall`(); a node process called `waitall`(); only host processes may call this function.

waitall (*node, pid*) pid must be 0 or -1

Error type is 1002 (Bad PID). Occurs in `waitall`(); the pid argument may only be set to 0 or -1 in this implementation.

Message Types A

Message types in the range 0 to 999,999,999 are assigned to a message at transmission time. Message types outside the above ranges are reserved for system use and should be avoided.

Functions that receive messages are able to specify the types of message that are to be received. The type variable is set according to the following conventions:

- If the type is a non-negative integer then a specific message type will be recognised; all other message types will be ignored, unless they are force types.
- If the type has a value of -1 then any message may be received.
- If the type is any negative number other than -1 then an exception is generated.

Computing
Surface

PVM User's Guide and Reference Manual

S1002–10M133.01

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*

Meiko's PVM product is based upon software and documentation that is subject to the following restrictions:

PVM 3.2 was funded in part by the U.S. Department of Energy, the National Science Foundation and the State of Tennessee.

Contents

Introduction 1

This chapter describes the features of the CS-2 implementation of PVM, and highlights the differences between standard PVM and Meiko's implementation (CS2-PVM).

Features of this Implementation

CS2-PVM allows PVM (version 3.2) applications to run on the CS-2 taking advantage of the high performance communication capability of the CS-2. In standard PVM most of the process control and message routing uses daemons, with one daemon running on each host. In the CS-2 implementation there are no PVM daemons. The process control functionality of the daemons is provided by the CS-2 Resource Management System. Message passing takes place directly using the tagged communication (tport) layer from the Elan Widget Library.

The Meiko resource manager cannot duplicate all of the functionality of the PVM daemons, so some of the calls that talk to the daemons are not supported in this implementation. In addition the absence of the daemons means that CS2-PVM cannot currently run in a mixed host environment; your applications are limited to the processing resource within the CS-2.

Programming Model

Meiko's implementation of PVM supports both *hosted* (master/slave) and *hostless* (SPMD) applications.

Hosted applications consist of two processes; a host and a number of identical node processes. The PVM application is initiated by executing the host process which is then responsible for spawning the node processes. All processes, including the host itself, use PVMs communication functions to cooperate and complete the task.

Hostless applications have a number of identical node processes that are started by using a loader program such as `prun`. These applications are coded as SPMD applications, in which one instance of the program acts as a master to a number of other node instances.

SPMD applications are unusual because they can be used as hosted or hostless programs. An instance of an SPMD application can be executed directly at your command shell, in which case it will spawn a number of copies of itself and then run as a host/node application. Alternatively a number of instances of an SPMD application can be started with a loader program, such as `prun`, in which case the spawning activity of the "host" instance is suppressed. This will be covered in more detail later.

Resource Allocation

All PVM applications must liaise with the CS-2 Resource Manager for processing resource. This liaison takes place within either the host process (for hosted applications) or the loader process (for hostless applications).

In either case the host/loader runs in your login partition as a sub-process of your command shell. The host/loader process calls upon functions in the resource management user interface library to liaise with the resource manager for the nodes' processing resource. In the case of a loader, such as `prun`, the liaison is via a direct calls to `rms_forkexecvp`() in `librms`. In the case of a PVM host process the liaison happens when the host process calls `pvm_spawn`(), which in turn calls `rms_forkexecvp`().

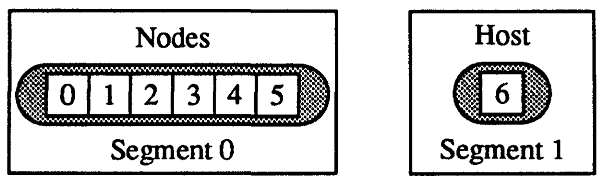
The resource management function uses the user's id and other criteria specified by your System Administrator to identify a suitable partition for the node processes. If you don't like the default resource you can specify your preferences by setting environment variables — the most useful variable is `RMS_PARTITION` which identifies your preferred partition, but there are others too (see the documentation for `rms_forkexecvp`()).

Process Communication

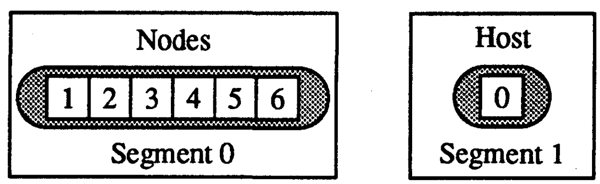
PVMs communication functions are built upon the tagged message port (TPORT) functions in the Elan Widget library. PVM applications are 2 segment CS-2 applications in which the host or loader program and the nodes run in separate segments. The two segments will usually run in separate partitions.

PVM processes have two numbering schemes associated with each process: there are the task-ids which are visible within the PVM application, and there are internal (virtual process) numbers that are used by the low level communication routines. You will need to understand the mapping from PVM tid to Elan virtual process numbers if you wish to include direct calls to the Elan Widget library within your PVM application.

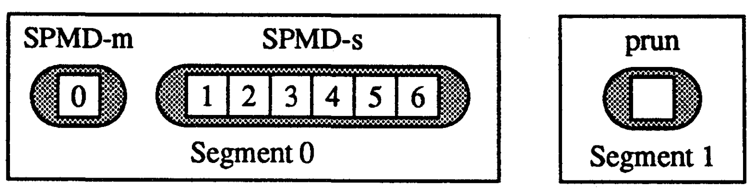
For the 6 processes in an example hosted PVM application the virtual process numbers are assigned as shown, with the node processes numbered from 0:



The PVM tids for the same example are allocated in a different order, with the host process numbered 0 and the nodes numbered from 1:



For a 6 process hostless applications the virtual process numbers and the tids are allocated in the same order as follows:



In general the allocation of each segment's processes to processors in a partition mirrors the allocation of the virtual process numbers; processes with low virtual process numbers are usually allocated to processors with lower Elan id's than those processes with high virtual process numbers.

Supported Functions

The following functions are defined in this library:

Process Control

The following functions are used to start and stop PVM processes.

`pvm_mytid`	Process initialisation.
`pvm_exit`	Process leaving PVM.
`pvm_spawn`	Start new PVM processes.

Information

These functions provide information about processes and the host environment.

`pvm_parent`	Returns the tid of the process that spawned this process.
`pvm_pstat`	Returns the status of the specified process.
`pvm_mstat`	Returns the status of a CS-2 partition.
`pvm_config`	Returns information about the current machine configuration.
`pvm_tasks`	Returns information about the tasks running on the CS-2.

Signalling

These functions enable a process to signal other processes in the application.

`pvm_sendsig`	Send a signal to a PVM process.
`pvm_kill`	Terminate a PVM process by sending a SIGTERM signal.

Error Handling

These functions enable error reporting.

`pvm_perror`	Print message describing the last error returned by a PVM function.
`pvm_serror`	Sets automatic error message printing on or off.

Message Buffers

These functions allow you to define message buffers.

`pvm_mkbuf`	Creates a new message buffer.
`pvm_initsend`	Clear default send buffer and specify message encoding.
`pvm_freebuf`	Disposes of a message buffer.
`pvm_getsbuf`	Returns the message buffer identifier for the active send buffer.
`pvm_getrbuf`	Returns the message buffer identifier for the active receive buffer.
`pvm_setsbuf`	Switches the active send buffer.
`pvm_setrbuf`	Switches the active receive buffer and saves the previous buffer.

Packing Message Buffers

These functions pack messages into message buffers.

`pvm_pk*` `pvm_packf`	Pack the active message buffer with arrays of prescribed data type.

Unpacking Message Buffers

These functions unpack messages from message buffers.

`pvm_unpk*` `pvm_unpackf`	Unpack the active message buffer into arrays of prescribed data type.

Sending and Receiving Data

These functions send and receive messages. Note that some functions block the calling process until the transaction is complete, whereas some allow the process to continue immediately (and require the transaction to be tested later).

`pvm_send`	Immediately sends the data in the active message buffer. This function is asynchronous; it does not suspend the calling process until a matching receive has been posted.
`pvm_mcast`	Multicasts the data in the active message buffer to a set of tasks.
`pvm_nrecv`	Non-blocking receive; fetches a message into a new active receive buffer if a message is available, but returns straight away even if the message has yet to arrive.
`pvm_recv`	Receive a message; this function will block the caller until a message is available.
`pvm_probe`	Check if a message has arrived.
`pvm_bufinfo`	Returns information about a message buffer.

Synchronisation

Synchronisation ensures that all processes enter critical sections of your code at the same time. Barriers are included within the definition of the PVM initialisation functions to ensure that the application does not begin until all processes have successfully initialised their communication mechanisms.

`pvm_barrier` Barrier synchronise all processes; suspend the calling process until other processes in the application have also called this function. `group/count` arguments are ignored in this implementation.

Unsupported Functions

The following functions are not supported in this implementation. Note that some functions are not defined (causing errors at program link time), some return an error ('not implemented'), and some may be called with no effect.

Most of the unsupported functions related to the group library and the interface to the pvmd daemons, neither of which are supported in this implementation.

Function	Behaviour
`pvm_addhosts`	Returns error (not implemented).
`pvm_advise`	May be called with no effect.
`pvm_bcast`	Not defined.
`pvm_delhosts`	Not defined.
`pvm_getinst`	Not defined.
`pvm_gettid`	Not defined.
`pvm_gsize`	Not defined.
`pvm_joingroup`	Not defined.
`pvm_lvgroup`	Not defined.
`pvm_notify`	Returns error (not implemented).
`pvm_recvf`	May be called with no effect.

The following function has a different meaning in this implementation:

Function	Behaviour
`pvm_barrier`	Barrier synchronisation of **all** processes.

Debugging

When the host of a hosted PVM application spawns the node processes under the control of a debugger (by specifying the `PvmTaskDebug` option to `pvm_spawn()`) the node processes are not executed directly but indirectly via a shell script.

By specifying the debug option `pvm_spawn()` locates a shell script called `debugger` in the directory `$HOME/pvm3/lib`[1] and passes it the name of the node task (as specified in the call to `pvm_spawn()`).

For example, consider the following call to `pvm_spawn()`, which identifies a node program in your current directory:

```
pvm_spawn("node",(char**)0,PvmTaskDebug,"",nproc,tids)
```

This causes `nproc` instances of `$HOME/pvm3/lib/debugger` to be started and passed as their first argument the name of the node process. If your preferred debugger is TotalView, the `debugger` script might be defined as follows:

```
#!/bin/csh -f
totalview $1
```

If you prefer DBX (in an X environment) you could use:

```
#!/bin/csh -f
exec xterm -n $1 -T $1 -ls -sb -sl100 -e dbx $1
```

1. This is the only occasion when Meiko's implementation of PVM requires a PVM subdirectory within your home directory.

PVM Console

There is no PVM console in the Meiko implementation. Many of the functions of the PVM console are available from resource management commands:

Console Commands	Meiko Alternatives
`conf`	`rinfo(1)` and `pandora(1)` can both be used to view the configuration of your machine (the partitions, their size, and their availability).
`add/delete`	Partition sizes can be changed by the System Administrator using `rcontrol(1m)` or `pandora(1)`.
`mstat`	The status of processors is available from `pandora(1)`.
`ps -a`	Use `ps(1)` or `gps(1)`.
`spawn`	Use `prun(1)` to spawn hostless applications, or execute the host of a hosted PVM application.
`kill/halt`	Use `gkill(1)` to terminate processes.

Performance Considerations

The host process (in a hosted PVM application) will normally execute in your login partition under the control of your command shell. In general the processors in the login partitions are heavily loaded and running tasks for more than one user. Applications in which the host process forms a key role in your application may therefore suffer significant and unpredictable performance variations. There are two solutions to this problem: either code the host process so that it does not take an active part in the overall application (i.e. limit it to a program loader), or code the application as a SPMD application so that all processes are executed together in a single partition.

The implementation of `pvm_spawn()` and `pvm_mytid()` include a barrier synchronisation. After spawning the node tasks, `pvm_spawn()` will suspend the host process until all the slave processes have executed `pvm_mytid()`. This implicit synchronisation is included to ensure that no process tries to communicate before the target process has initialised its CS-2 communication environment. To ensure that the application begins as quickly as possible all the node processes must include at the beginning of the program a call to `pvm_mytid()`.

Compilation of PVM Programs

PVM programs must be linked with the low level Elan communications libraries and the resource management library.

Use the following command line to compile C programs:

```
user@cs2: cc -o program -I/opt/MEIKOcs2/include \
-L/opt/MEIKOcs2/lib -R/opt/MEIKOcs2/lib program.c \
-lpvm3 -lrms -lew -lelan -lsocket -lnsl
```

Use the following command line to compile Fortran programs:

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

Note that the `-R` option specifies a search path to the run-time linker to locate dynamic libraries. If you fail to include this option you will get the following error:

```
ld.so.1:program:fatal:librms.so.2: can't open file: errno=2
Killed
```

To overcome this problem you must either recompile your application or include in your `LD_LIBRARY_PATH` environment variable the pathname for the Meiko library directory.

Notes for User of SunPro Fortran77

When using the SunPro F77 compiler you must specify both the Meiko library directory and the SunPro library directory after your compiler driver's `-R` option, or you can omit the `-R` option and set the `LD_RUN_PATH` environment variable before compilation to include just the Meiko library directory.

Header Files

Function prototypes and constants used by the PVM functions are defined in two header files, `pvm3.h` and `fpvm3.h`, which are used by the C and Fortran libraries respectively. Both files are in the directory `/opt/MEIKOcs2/include/PVM`.

You should include the appropriate file in your program by using the preprocessor's `#include` directive near the beginning of your program file.

Fortran programmers can use a filename suffix of .F for their program files which will instruct most compiler drivers to automatically pass your program through the pre-processor — see the example Fortran programs in `/opt/MEIKOcs2/example/PVM`.

Executing PVM Applications

You execute a hosted PVM application by executing the host process directly from your command shell. The host will liaise with the Resource Manager and spawn the node processes:

```
user@cs2: master
```

You execute a SPMD application by executing the program from your command shell. This program will then liaise with the Resource Manager and spawn additional copies of itself:

```
user@cs2: spmd
```

You execute a hostless application using `prun` or some other loader program. Note that the number of instances loaded by `prun` must be compatible with the number of processes specified to `pvm_spawn`(); the number of processes loaded by `prun` must always be 1 larger than the argument to `pvm_spawn`(). The following example loads 5 instances of the SPMD application:

```
user@cs2: prun -n5 -pparallel node
```

In all cases you specify your resource requirements with environment variables (`prun` will read these environment variables but also allows you to specify your requirements on the command line, as shown in the previous example). The following environment variables may be specified:

Variable	Meaning
`RMS_PARTITION`	The name of the partition that will host the node processes.
`RMS_BASEPROC`	The id of the first processor in the partition that you want to use (usually this is the first available processor)
`RMS_NPROCS`	The number of processors required in the target partition.
`RMS_MEMORY`	The minimum memory requirement for each processor, suffixed by K or M (for kilobytes and megabytes respectively).
`RMS_STDIOLOG`	Preserve IO from each process (don't delete temporary files) if this variable is set.
`RMS_VERBOSE`	Set level of status reporting.

For example, to specify that all node processes are spawned in the `parallel` partition you need to ensure that the `RMS_PARTITION` environment variable is set before you execute your PVM application. A C-shell user would set the variable as follows:

```
user@cs2: setenv RMS_PARTITION parallel
```

You can check the availability of your system and identify its partitions with the `rinfo` command.

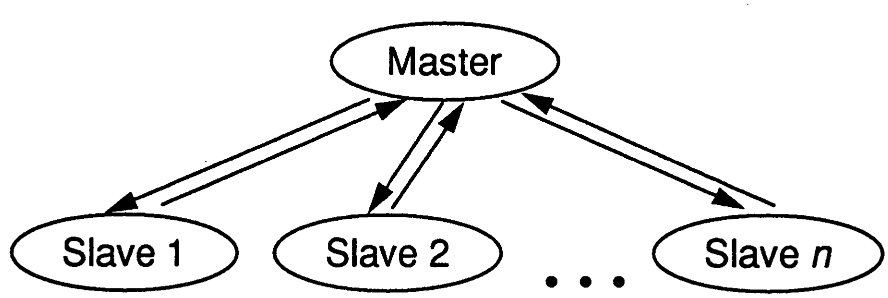
Example Programs 2

A number of example programs are distributed in `/opt/MEIKOcs2/example/PVM`. The following text describes how 2 of these programs are compiled and executed on the CS-2, and explains their interaction with the resource management system and the Elan Widget library.

Master/Slave Example

This example consists of two programs, a master and a slave. The example is started by executing the master program, which prompts for a number of slave processes. The slaves are spawned within a CS-2 partition and are passed a data vector from the master. Each slave returns a result to the master which is displayed on screen.

Figure 2-1 Master/Slave Communications



Compiling the Example

Before compiling or editing the example programs you should copy them into your home directory so that your work does not conflict with the work of others.

```
user@cs2 mkdir ~/PVM
user@cs2 cp /opt/MEIKOcs2/example/PVM/* ~/PVM
user@cs2 cd ~/PVM
```

Both programs can be compiled using the makefile that is distributed with the example programs. Type the following command to compile the C version of this example:

```
user@cs2 make master slave
```

The makefile executes the following compiler command lines (which you can type yourself if you prefer not to use `make`):

```
user@cs2 cc -I/opt/MEIKOcs2/include/PVM -o master1\
master1.c -L/opt/MEIKOcs2/lib -R/opt/MEIKOcs2/lib \
-lpvm3 -lrms -lew -lelan -lsocket -lnsl

user@cs2 cc -g -I/opt/MEIKOcs2/include/PVM -o slave1\
slave1.c -L/opt/MEIKOcs2/lib -R/opt/MEIKOcs2/lib \
-lpvm3 -lrms -lew -lelan -lsocket -lnsl
```

Starting the Example

You specify your resource requirements by setting environment variables. In the following C-shell example the parallel partition is identified as the target for the node processes:

```
user@cs2 setenv RMS_PARTITION parallel
```

You execute the example by executing the master program:

```
user@cs2 master1
How many slave programs (1-32)?
```

You can specify that up to 32 slave processes are spawned by the master, but note that the program will fail if you ask for more processes than can be supported by your nominated partition. If the partition is too small (or unavailable) you will get an appropriate error message from the resource management system. Note also that your program may be queued (and appear to hang) if the partition contains resource that is temporarily allocated to other tasks. Use `rinfo` to check the availability and size of your partitions.

The example should complete soon after it is started and confirm that a result was received from all the slaves.

Detailed Description of the Programs

This example defines a simple 2 segment application.

The master process performs the role of a program loader; it includes within it embedded calls to the resource management system which are used to allocate resource and execute the slave processes. The master process executes in your login partition on the processor that is hosting your command shell. The slave processes execute in some other partition (identified by the `RMS_PARTITION` environment variable).

The master process begins by executing `pvm_mytid()` (which for the master process actually does nothing but return the process tid).

After fetching a process count from the user a number of slave processes are spawned with `pvm_spawn()`. It is here that the master process interfaces with the resource management system — the request for resource and the execution of the slave processes is handled within `pvm_spawn()` by a call to `rms_forkexecvp()`[1] (a function in `librms` the resource management user interface library). `pvm_spawn()` also defines the underlying communication channels (implemented on Elan Widget Library TPORTs) and includes an implicit barrier that will delay execution of the master until all the slave processes are running and ready to communicate. This barrier is a safeguard to ensure that no inter-process communications may take place before the underlying communication mechanisms (TPORTs) are in place on all processes.

Initialisation of the communication channels within the slave processes is handled during the call to `pvm_mytid()`. This function attaches the slave process to the Elan network and uses the Widget library functions to initialise the TPORT communication channels. Only when **all** the slave processes have executed this function will they and the master be released from their barrier synchronisation.

The remainder of the example programs demonstrates PVMs message passing functions. The master builds a packet that is multicast to all the slaves. Each slave then performs some simple calculation, some one-to-one inter-process communications, and returns a result to the master (which is displayed on screen). All processes execute `pvm_exit()` before finishing.

SPMD Example

This example is essentially the same as the master/slave example described earlier, except in this example the code for both is defined by a single file. Using this method of coding allows the program to be executed as either a hosted or a hostless application.

---

1. Any of the environment variables supported by rms_forkexecvp() may be used to specify the requirements of your parallel application. The most useful variable is RMS_PARTITION, which identifies your preferred partition. See the documentation for rms_forkexecvp() for the full list of environment variables.

Hosted SPMD Application

To run as a hosted application you execute the program directly from your command shell. (As with the previous master/slave example you may prefer to specify your resource requirements for the node processes by setting the appropriate environment variables.)

```
user@cs2 setenv RMS_PARTITION parallel
user@cs2 spmd
me = 3 mytid = xxx
me = 2 mytid = yyy
me = 1 mytid = zzz
me = 0 mytid = 0
token ring done
```

The program begins with a call to `pvm_mytid()` which identifies this process as the first in the application and causes it to execute the host-specific code. The host's code includes a call to `pvm_spawn()` which spawns the node processes, initialises the host's communication ports, and barrier synchronises until the node processes are ready (i.e. until they have all successfully executed `pvm_mytid()`). Following the initialisation all processes (host and nodes) execute the same code and cooperate to complete the task.

Note that when using the hosted model the host process runs in your login partition and the node processes run in some other partition (which you will usually identify with the `RMS_PARTITION` environment variable).

Hostless SPMD Application

To run as a hostless application you load all instances of the parallel application by using a loader program, such as `prun`. When using `prun` all the processes are loaded into the same partition, and all begin executing at the same time.

The following example will spawn 4 instances of the SPMD program onto the `parallel` partition[1]:

```
user@cs2 prun -n4 -pparallel spmd
me = 3 mytid = xxx
me = 2 mytid = yyy
me = 1 mytid = zzz
me = 0 mytid = 0
token ring done
```

The process with tid 0 assumes the role of a master; a call to `pvm_mytid()` identifies the master process and causes it to branch into the master-specific part of the program. As with the hosted application the master program executes `pvm_spawn()`, but in this case the function's behaviour changes — it does not attempt to spawn the node processes (which have already been spawned by `prun`). When used within a hostless application `pvm_spawn()` initialises the master's communication mechanism, barrier synchronises with the remaining node processes, and returns to the caller the array of tids for the application.

The node processes begin executing immediately `prun` completes, however these processes will stop as soon as they reach the call to `pvm_mytid()` — remember that for node processes this function initialises the process's communication ports and then barrier synchronises.

When the barrier synchronisation in the master (`pvm_spawn()`) and nodes (`pvm_mytid()`) completes all processes resume execution. The master completes its initialisation and then continues by executing the same code as the nodes. All processes then cooperate to complete the task.

Note that when using the hostless model **all** processes (host and node) execute in the same partition, which is usually identified either as an argument to `prun` or by setting the `RMS_PARTITION` environment variable.

1. The SPMD program is assumed to specify 3 node processes to pvm_spawn().

Program Compilation

The program can be compiled with the supplied makefile (the same compilation procedure is used for either hosted or hostless methods of execution):

```
user@cs2 make spmd
```

Reference Manual 3

This chapter contains the reference manual pages for all the functions that are defined in this library. The manual pages are also available on-line for use with the `man` command.

Each function (or function group) is described on a separate page; the pages are ordered alphabetically.

pvm_intro — **Parallel Virtual Machine System Version 3.2**

Description

The CS-2 implementation of PVM makes the high performance communication capabilities of the CS-2 available to PVM application programs.

- CS2-PVM does not run in a mixed host environment.
- User programs are written in C, C++ or Fortran and access PVM through library routines (libpvm3.a and libfpvm3.a).
- The Meiko Resource Management System provides process control whereas the communication routines use the Elan widget tport layer.
- Both hosted (master/slave) and hostless (SPMD) applications are supported in this release.

The distinguishing features of this release (Meiko's 1.3 release) are:

Organisation

No PVM daemons (pvmd) need to be spawned. The functionality of pvmd is provided by the Resource Management System. The resource manager must be available before any PVM applications can be run. CS2-PVM currently cannot run in a mixed host environment.

Hosted vs Hostless

Both hosted (master/slave) and hostless (SPMD) applications are supported. Hosted applications are initiated by executing the host directly from your command shell; this then spawns (via `pvm_spawn`()) a number of identical node processes into a CS-2 partition. Hostless applications consist of a number of identical SPMD programs that are spawned using a program loader such as `prun`(1).

Compiling/running

PVM applications should be linked with libpvm3.a and libfpvm3.a for C and Fortran programs respectively. Additionally applications need to be linked with the resource management library (librms.a), the CS-2 communications libraries (libew.a and libelan.a), and the libsocket.a and libnsl.a libraries. For example:

```
user@cs2: cc -o master I/opt/MEIKOcs2/include \
-L/opt/MEIKOcs2/lib -R/opt/MEIKOcs2/lib master.c \
-lpvm3 -lrms -lew -lelan -lsocket -lnsl
```

See also the examples in `/opt/MEIKOcs2/example/PVM`.

Process control

Process control is provided by the Resource Management System, primarily to spawn (and terminate) PVM tasks. Typically a master task calls pvm_spawn() specifying the (slave) task name and the number of copies to be spawned. For example:

```
pvm_spawn("slave",(char**)0, 0, "", nproc, tids);
```

The master then negotiates with the resource manager to spawn the tasks and setup the CS-2 environment. By default tasks are spawned on the partition identified by your System Administrator. To spawn tasks on another partition use the environment variable RMS_PARTITION to specify the partition name. pvm_spawn() is restricted in that it can only be called once in an application. Note also that pvm_spawn() tries to synchronise with the slave/node tasks via pvm_mytid(); these tasks must therefore call pvm_mytid() before any other PVM calls. Likewise before exiting all tasks must call pvm_exit(), which synchronises tasks before they exit.

Message passing

pvm_send(), pvm_recv(), pvm_nrecv(), pvm_mcast() & pvm_probe() are all implemented on Elan Widget Library tports.

PVM console

PVM console is not supported, although the Resource Management System utility rinfo can provide similar functionality.

Debugging

The Resource Management Library allows tasks to be spawned under a debugger. When debugging the resource manager does not run spawned tasks directly but does instead executes a shell-script that spawns the task via a debugger. The following example spawns nproc instances of the script $HOME/pvm3/lib/debugger which can run the task under a debugger:

```
pvm_spawn("slave",(char**)0,PvmTaskDebug,"",nproc,tids);
```

The debugger script can run a task under any available debugger. For instance to debug this task with TotalView use the following script:

```
#!/bin/csh -f
totalview $1
```

or with DBX (in an X environment) use:

```
#!/bin/csh -f
exec xterm -n $1 -T $1 -ls -sb -sl100 -e dbx $1
```

Group library

The PVM group library is not supported, although the `pvm_barrier()` call is provided to allow all tasks to synchronise.

Other calls not supported

A number of other PVM calls are not supported. These include: `pvm_delhosts()`, `pvm_halt()`, and `pvm_notify()`.

See Also

PVM 3.2 User's Guide and Reference Manual

pvm_barrier() — Synchronise processes

Synopsis

```
int info = pvm_barrier( char *group, int count )
```

Synopsis

```
call pvmfbarrier( group, count, info )
```

Arguments

`group` Character string group name (ignored by this implementation).

`count` Integer specifying the number of group members that must call `pvm_barrier()` before they are all released (ignored by this implementation — all processes must call this function).

`info` Integer status code returned by the routine. Values less than zero indicate an error.

Description

`pvm_barrier()` blocks the calling process until *all* members of the group have called `pvm_barrier()`. This implementation does not support PVMs group mechanisms; `pvm_barrier()` may therefore only be used to synchronise all the processes in the application. Note that the `group` and `count` arguments are ignored and can be NULL. `pvm_barrier()` uses `ew_gsync()` from the Elan Widget library to synchronise tasks.

Examples

C:

```
info = pvm_barrier( NULL, NULL );
```

Fortran:

```
CALL PVMFBARRIER( 0, 0, INFO )
```

If `pvm_barrier()` is successful `info` will be 0. If some error occurs then `info` will be less than 0.

Errors

The following error conditions can be returned by `pvm_barrier()`;

`PvmSysErr` Resource management system (machine manager) was not started or has crashed.

See Also

`ew_gsync(3x)`

pvm_bufinfo()

Returns information about a message buffer

Synopsis

```
int info = pvm_bufinfo( int bufid, int *bytes,
                        int *msgtag, int *tid )
```

Synopsis

```
call pvmfbufinfo( bufid, bytes, msgtag, tid, info )
```

Arguments

`bufid`	Integer specifying a particular message buffer identifier.
`bytes`	Integer returning the length in bytes of the entire message.
`msgtag`	Integer returning the message label. Useful when the message was received with a wildcard msgtag.
`tid`	Integer returning the source of the message. Useful when the message was received with a wildcard tid.
`info`	Integer status code returned by the routine. Values less than zero indicate an error.

Description

`pvm_bufinfo()` returns information about the requested message buffer. Typically it is used to determine facts about the last received message such as its size or source. `pvm_bufinfo()` is especially useful when an application is able to receive any incoming message, and the action taken depends on the source tid and the msgtag associated with the message that comes in first.

If `pvm_bufinfo()` is successful `info` will be 0. If some error occurs then `info` will be less than 0.

Example

C:

```
bufid = pvm_recv( -1, -1 );
info = pvm_bufinfo( bufid, &bytes, &type, &source );
```

Fortran:

```
CALL PVMFRECV( -1, -1, BUFID )
CALL PVMFBUFINFO( BUFID, BYTES, TYPE, SOURCE, INFO )
```

Errors

The following error conditions can be returned by `pvm_bufinfo()`.

`PvmNoSuchBuf`	specified buffer does not exist.
`PvmBadParam`	invalid argument.

See Also

`pvm_recv(3)`

pvm_config() — **Returns information about the present virtual machine configuration**

Synopsis

```
int info = pvm_config( int *nprocs, int *narch,
                       struct hostinfo **hostp )
struct hostinfo {
   int   hi_tid;
   char *hi_name;
   char *hi_arch;
   int   hi_speed;
};
```

Synopsis

```
call pvmfconfig( nproc, narch, dtid, name, arch,
                 speed, info )
```

Arguments

`nprocs`	Integer returning the number of processors in the partition.
`narch`	Integer returning the number of different data formats being used (always -1 for the CS-2).
`hostp`	Pointer to an array of structures which contain information about each host including its name, architecture, and relative speed.
`dtid`	Integer returning pvmd task ID (always -1 for the CS-2).
`name`	Character string returning name of this node.
`arch`	Character string returning name of host architecture; this is "cs2"
`speed`	Integer returning relative speed of this host. Default value is 1000.
`info`	Integer status code returned by the routine. Values less than zero indicate an error.

Description

`pvm_config()` returns information about a CS-2 partition.

The C function returns information about the entire partition in one call. The Fortran function returns information about one host per call and cycles through all the hosts; if `pvmfconfig()` is called `nproc` times the entire partition will be represented.

If `pvm_config()` is successful `info` will be 0. If some error occurs then `info` will be < 0.

This function is useful for determining the number of processors there are in a partition.

Example

C:

```
info = pvm_config( &nproc, &narch, &hostp );
```

Fortran:

```
 Do i=1, NPROC
    CALL PVMFCONFIG( NPROC, NARCH, DTID(i), HOST(i), ARCH(i),
&   SPEED(i),INFO)
 Enddo
```

See Also

pvm_tasks(3)

pvm_exit()

Tells the resource management system that this process is leaving PVM

Synopsis

`int info = pvm_exit( void )`

Synopsis

`call pvmfexit( info )`

Arguments

`info` Integer status code returned by the routine. Values less than zero indicate an error.

Description

`pvm_exit()` tells the resource management system that this process is leaving PVM. This routine does not kill the process, which can continue to perform tasks just like any other serial process.

In hosted applications `pvm_exit()` calls `rms_waitpid()` in the master task to wait until all slave tasks have exited.

Examples

C:

```
/* Program done */
pvm_exit();
exit();
```

Fortran:

```
CALL PVMFEXIT(INFO)
STOP
```

Errors

The following error condition can be returned by `pvm_exit()`:

`PvmSysErr` Resource management error (machine manager unavailable)

See Also

`rms_waitpid`(3x)

pvm_freebuf() — Disposes of a message buffer

Synopsis

```
int info = pvm_freebuf( int bufid )
```

Synopsis

```
call pvmffreebuf( bufid, info )
```

Arguments

`bufid` Integer message buffer identifier.

`info` Integer status code returned by the routine. Values less than zero indicate an error.

Description

`pvm_freebuf()` frees the memory associated with the message buffer identified by `bufid`. Message buffers are created by `pvm_mkbuf()`, `pvm_initsend()`, and `pvm_recv()`. If `pvm_freebuf()` is successful `info` will be 0. If some error occurs then `info` will be < 0.

`pvm_freebuf()` can be called for a send buffer created by `pvm_mkbuf()` after the message has been sent and is no longer needed.

Receive buffers typically do not have to be freed unless they have been saved in the course of using multiple buffers, but note that `pvm_freebuf()` can be used to destroy receive buffers as well. Messages that arrive but are no longer needed can be destroyed so they will not consume buffer space.

Typically multiple send and receive buffers are not needed and the user can simply use the `pvm_initsend()` routine to reset the default send buffer.

There are several cases where multiple buffers are useful. One example where multiple message buffers are needed involves libraries or graphical interfaces that use PVM and interact with a running PVM application but do not want to interfere with the application's own communication.

When multiple buffers are used they generally are made and freed for each message that is packed. In fact, `pvm_initsend()` simply does a `pvm_freebuf()` followed by a `pvm_mkbuf()` for the default buffer.

Examples

C:

```
bufid = pvm_mkbuf(PvmDataDefault);
  :
info = pvm_freebuf(bufid);
```

Fortran:

```
CALL PVMFMKBUF( PVMDEFAULT, BUFID )
 :
CALL PVMFFREEBUF( BUFID, INFO )
```

Errors

These error conditions can be returned by `pvm_freebuf()`:

`PvmBadParam`	giving an invalid argument value.
`PvmNoSuchBuf`	giving an invalid bufid value.

See Also

`pvm_mkbuf(3)`, `pvm_initsend(3)`, `pvm_recv(3)`.

pvm_getrbuf() — Returns the message buffer identifier for the active receive buffer

Synopsis `int bufid = pvm_getrbuf( void )`

Synopsis `call pvmfgetrbuf( bufid )`

Arguments

`bufid` Integer returning message buffer identifier for the active receive buffer.

Description

`pvm_getrbuf()` returns the message buffer identifier `bufid` for the active receive buffer or 0 if there is no current buffer.

Examples

C:

```
bufid = pvm_getrbuf();
```

Fortran:

```
CALL PVMFGETRBUF( BUFID )
```

See Also

`pvm_getsbuf(3)`

pvm_getsbuf() — **Returns the message buffer identifier for the active send buffer**

Synopsis

```
int bufid = pvm_getsbuf( void )
```

Synopsis

```
call pvmfgetsbuf( bufid )
```

Arguments

`bufid` Integer returning message buffer identifier for the active send buffer.

Description

`pvm_getsbuf()` returns the message buffer identifier `bufid` for the active send buffer or 0 if there is no current buffer.

Examples

C:

```
bufid = pvm_getsbuf();
```

Fortran:

```
CALL PVMFGETSBUF( BUFID )
```

See Also

`pvm_getrbuf(3)`

pvm_initsend() — Clear default send buffer and specify message encoding

Synopsis
```
int bufid = pvm_initsend( int encoding )
```

Synopsis
```
call pvmfinitsend( encoding, bufid )
```

Arguments

`encoding` Integer specifying the next message's encoding scheme.

Options in C are:

Encoding value		**MEANING**
PvmDataDefault	0	XDR
PvmDataRaw	1	no encoding
PvmDataInPlace	2	data left in place

Option names are shortened in Fortran to:

Encoding value		**MEANING**
PVMDEFAULT	0	XDR
PVMRAW	1	no encoding
PVMINPLACE	2	data left in place

`bufid` Integer returned containing the message buffer identifier. Values less than zero indicate an error.

Description

`pvm_initsend()` clears the send buffer and prepares it for packing a new message. The encoding scheme used for the packing is set by encoding, which for CS2-PVM defaults to `PvmDataRaw` since all CS-2 nodes are homogeneous.

`PvmDataInPlace` encoding specifies that data be left in place during packing. The message buffer only contains the sizes and pointers to the items to be sent. When `pvm_send()` is called the items are copied directly out of the user's memory. This option decreases the number of times a message is copied at the expense of requiring the user to not modify the items between the time they are packed and the time they are sent. The PvmDataInPlace is not implemented in the version 3.2.

If pvm_initsend() is successful then bufid will contain the message buffer identifier. If some error occurs then bufid will be < 0.

Examples

C:

```
bufid = pvm_initsend( PvmDataDefault );
info = pvm_pkint( array, 10, 1 );
msgtag = 3;
info = pvm_send( tid, msgtag );
```

Fortran:

```
CALL PVMFINITSEND( PVMRAW, BUFID )
CALL PVMFPACK( REAL4, DATA, 100, 1, INFO )
CALL PVMFSEND( TID, 3, INFO )
```

Errors

These error conditions can be returned by pvm_initsend():

PvmBadParam	giving an invalid encoding value
PvmNoMem	Malloc has failed. There is not enough memory to create the buffer.

See Also

pvm_mkbuf(3)

pvm_kill() — Terminates a specified PVM process

Synopsis

```
int info = pvm_kill( int tid )
```

Synopsis

```
call pvmfkill( tid, info )
```

Arguments

`tid` Integer task identifier of the PVM process to be killed (not yourself).

`info` Integer status code returned by the routine. Values less than zero indicate an error.

Description

`pvm_kill()` sends a terminate (SIGTERM) signal to the PVM process identified by `tid`. If `pvm_kill()` is successful `info` will be 0. If some error occurs then `info` will be < 0.

`pvm_kill()` is not designed to kill the calling process. To kill yourself in C call `pvm_exit()` followed by `exit()`. To kill yourself in Fortran call `pvmfexit()` followed by `stop`.

Examples

C:

```
info = pvm_kill( tid );
```

Fortran:

```
CALL PVMFKILL( TID, INFO )
```

Errors

These error conditions can be returned by `pvm_kill()`:

`PvmBadParam`	giving an invalid `tid` value.
`PvmSysErr`	internal error.

See Also

`pvm_exit(3)`, Meiko Resource Management System document set.

pvm_mcast()

Multicasts the data in the active message buffer to a set of tasks

Synopsis

```
int info = pvm_mcast( int *tids, int ntask, int msgtag )
```

Synopsis

```
call pvmfmcast( ntask, tids, msgtag, info )
```

Arguments

`ntask`	Integer specifying the number of tasks to be sent to.
`tids`	Integer array of length `ntask` containing the task IDs of the tasks to be sent to.
`msgtag`	Integer message tag supplied by the user. `msgtag` should be ≥0. It allows the user's program to distinguish between different kinds of messages.
`info`	Integer status code returned by the routine. Values less than zero indicate an error.

`pvm_mcast()` multicasts a message stored in the active send buffer to `ntask` tasks specified in the `tids` array. The message is not sent to the caller even if listed in the array of `tids`. The content of the message can be distinguished by `msgtag`. If `pvm_mcast()` is successful `info` will be 0. If some error occurs then `info` will be < 0.

The receiving processes can call either `pvm_recv()` or `pvm_nrecv()` to receive their copy of the multicast. `pvm_mcast()` is asynchronous and computation on the sending processor resumes as soon as the message is safely on its way to the receiving processors. This is in contrast to synchronous communication, during which computation on the sending processor halts until the matching receive is executed by the receiving processor.

On the CS-2 `pvm_mcast()` uses the high speed interconnect via the tport layer in the Elan Widget library.

Examples

C:

```
info = pvm_initsend( PvmDataRaw );
info = pvm_pkint( array, 10, 1 );
msgtag = 5;
info = pvm_mcast( tids, ntask, msgtag );
```

Fortran:

```
CALL PVMFINITSEND( PVMDEFAULT )
CALL PVMFPACK( REAL4, DATA, 100, 1, INFO )
CALL PVMFMCAST( NPROC, TIDS, 5, INFO )
```

Errors

These error conditions can be returned by `pvm_mcast()`:

`PvmBadParam`	giving a `msgtag` < 0.
`PvmSysErr`	Resource management system error.
`PvmNoBuf`	no send buffer.

See Also

`EW_TPORT`(3x), Meiko Elan Widget library documentation set.

pvm_mkbuf() — Creates a new message buffer.

Synopsis

```
int bufid = pvm_mkbuf( int encoding )
```

Synopsis

```
call pvmfmkbuf( encoding, bufid )
```

Arguments

`encoding` — Integer specifying the next message's encoding scheme.

Options in C are:

Encoding value		**MEANING**
PvmDataDefault	0	XDR
PvmDataRaw	1	no encoding
PvmDataInPlace	2	data left in place

Option names are shortened in Fortran to:

Encoding value		**MEANING**
PVMDEFAULT	0	XDR
PVMRAW	1	no encoding
PVMINPLACE	2	data left in place

`bufid` — Integer returned containing the message buffer identifier. Values less than zero indicate an error.

Description

`pvm_mkbuf()` creates a new message buffer and sets its encoding status to encoding. If `pvm_mkbuf()` is successful then `bufid` will be the identifier for the new buffer, which can be used as a send buffer. If some error occurs then `bufid` will be < 0.

Encoding in CS2-PVM defaults to `PvmDataRaw` since all CS-2 nodes are homogeneous.

`PvmDataInPlace` encoding specifies that data be left in place during packing. The message buffer only contains the sizes and pointers to the items to be sent. When `pvm_send()` is called the items are copied directly out of the user's memory. This option decreases the number of times a message is copied at the expense of requiring the user to not modify the items between the time they are packed and the time they are sent. The PvmDataInPlace option is not implemented in this version 3.2.

pvm_mkbuf() is required if the user wishes to manage multiple message buffers and should be used in conjunction with pvm_freebuf(). pvm_freebuf() should be called for a send buffer after a message has been sent and is no longer needed.

Receive buffers are created automatically by the pvm_recv() and pvm_nrecv() routines and do not have to be freed unless they have been explicitly saved with pvm_setrbuf().

Typically multiple send and receive buffers are not needed and the user can simply use the pvm_initsend() routine to reset the default send buffer.

There are several cases where multiple buffers are useful. One example where multiple message buffers are needed involves libraries or graphical interfaces that use PVM and interact with a running PVM application but do not want to interfere with the application's own communication.

When multiple buffers are used they generally are made and freed for each message that is packed.

Examples

C:

```
bufid = pvm_mkbuf( PvmDataRaw );
/* send message */
info = pvm_freebuf( bufid );
```

Fortran:

```
  CALL PVMFMKBUF( PVMDEFAULT, MBUF )
* SEND MESSAGE HERE
  CALL PVMFFREEBUF( MBUF, INFO )
```

Errors

These error conditions can be returned by `pvm_mkbuf()`:

`PvmBadParam`	giving an invalid encoding value.
`PvmNoMem`	Malloc has failed. There is not enough memory to create the buffer.

See Also

`pvm_initsend(3)`, `pvm_freebuf(3)`

pvm_mstat() **Returns the status of a partition on the CS-2**

Synopsis

```
int mstat = pvm_mstat( char *host )
```

Synopsis

```
call pvmfmstat( host, mstat )
```

Arguments

`host` Character string containing the host name. This is ignored on the CS-2 and a NULL value can be passed.

`mstat` Integer returning machine status:

Value	Meaning
PvmOk	host is OK
PvmHostFail	partition is down

Description

`pvm_mstat()` returns the status `mstat` of a partition on the CS-2; the partition is specified by the RMS_PARTITION environment variable or (if the environment variable is not set) it will be the default partition specified by your System Administrator.

Examples

C:

```
mstat = pvm_mstat( NULL );
```

Fortran:

```
CALL PVMFMSTAT( 0, MSTAT )
```

Errors

These error conditions can be returned by `pvm_mstat()`;

`PvmSysErr`	Internal error.
`PvmHostFail`	partition is down.

See Also

`pvm_config(3)`, Meiko Resource Management System document set.

pvm_mytid()

Returns the tid of the calling process

Synopsis

```
int tid = pvm_mytid( void )
```

Synopsis

```
call pvmfmytid( tid )
```

Arguments

`tid` Integer returning the task identifier of the calling PVM process. Values less than zero indicate an error.

Description

`pvm_mytid()` enrols this process into PVM on its first call. `pvm_mytid()` returns the `tid` of the calling process and can be called multiple times in an application.

Any PVM system call (not just `pvm_mytid()`) will enrol a task in PVM if the task is not enrolled before the call.

When executed by node processes `pvm_mytid()` includes an implicit barrier (a call to `ew_baseInit()`) that will block the calling process until all other processes in the application have also executed the barrier. This means that a node process is delayed until all the other nodes have initialised, and until the host process has called `pvm_spawn()`. For host processes `pvm_mytid()` simply returns a `tid` (the barrier does not occur until the host executes `pvm_spawn()`).

Examples

C:

```
tid = pvm_mytid();
```

Fortran:

```
CALL PVMFMYTID( TID )
```

Errors

This error condition can be returned by `pvm_mytid()`:

`PvmSysErr` Resource management system error.

See Also

`pvm_parent(3)`, `ew_baseInit(3x)`, `ew_gsync(3x)`, Meiko Elan Widget library documentation set.

pvm_nrecv() — **Non-blocking receive**

Synopsis

```
int bufid = pvm_nrecv( int tid, int msgtag )
```

Synopsis

```
call pvmfnrecv( tid, msgtag, bufid )
```

Arguments

`tid`	Integer task identifier of sending process supplied by the user.
`msgtag`	Integer message tag supplied by the user. `msgtag` should be ≥ 0.
`bufid`	Integer returning the value of the new active receive buffer identifier. Values less than zero indicate an error

Description

`pvm_nrecv()` checks to see if a message with label `msgtag` has arrived from `tid` and also clears the current receive buffer, if any. If a matching message has arrived `pvm_nrecv()` immediately places the message in a new active receive buffer, and returns the buffer identifier in `bufid`.

If the requested message has not arrived then `pvm_nrecv()` immediately returns with a 0 in `bufid`. If some error occurs `bufid` will be < 0.

A -1 in `msgtag` or `tid` matches anything. This allows the user the following options. If `tid` = -1 and `msgtag` is defined by the user, then `pvm_nrecv()` will accept a message from any process which has a matching `msgtag`. If `msgtag` = -1 and `tid` is defined by the user, then `pvm_nrecv()` will accept any message that is sent from process `tid`. If `tid` = -1 and `msgtag` = -1, then `pvm_nrecv()` will accept any message from any process.

The PVM model guarantees the following about message order. If task 1 sends message A to task 2, then task 1 sends message B to task 2, message A will arrive at task 2 before message B. Moreover, if both messages arrive before task 2 does a receive, then a wildcard receive will always return message A.

`pvm_nrecv()` is non-blocking in the sense that the routine always returns immediately either with the message or with the information that the message has not arrived yet.

`pvm_nrecv()` can be called multiple times to check if a given message has arrived yet. In addition the blocking receive `pvm_recv()` can be called for the same message if the application runs out of work it could do before the data arrives.

If pvm_nrecv() returns with the message then the data in the message can be unpacked into the user's memory using the unpack routines.

On the CS-2, pvm_nrecv() uses the high-speed interconnect via the tport layer in the Elan Widget library.

Example

C:

```
tid = pvm_parent();
msgtag = 4;
arrived = pvm_nrecv( tid, msgtag );
if (arrived > 0)
   info = pvm_upkint( tid_array, 10, 1 );
else
   /* go do other computing */
```

Fortran:

```
   CALL PVMFNRECV( -1, 4, ARRIVED )
   IF (ARRIVED .gt. 0) THEN
      CALL PVMFUNPACK( INTEGER4, TIDS, 25, 1, INFO )
      CALL PVMFUNPACK(REAL8, MATRIX, 100, 100, INFO)
   ELSE
* GO DO USEFUL WORK
   ENDIF
```

Errors

These error conditions can be returned by pvm_nrecv():

PvmBadParam	giving an invalid tid value or msgtag.
PvmSysErr	Resource management system error.

See Also

pvm_recv(3), pvm_unpack(3), pvm_send(3), pvm_mcast(3), EW_TPORT(3x), Meiko Elan Widget library documentation set.

pvm_pack **Pack the active message buffer with arrays of prescribed data type**

Synopsis

```
int info = pvm_packf( const char *fmt, ... )
int info = pvm_pkbyte( char *xp,int nitem,int stride )
int info = pvm_pkcplx( float *cp,int nitem,int stride)
int info = pvm_pkdcplx( double *zp, int nitem,
                        int stride)

int info = pvm_pkdouble( double *dp, int nitem,
                         int stride)

int info = pvm_pkfloat(float *fp,int nitem,int stride)
int info = pvm_pkint( int *ip,int nitem,int stride )
int info = pvm_pkuint( unsigned int *ip, int nitem,
                       int stride)

int info = pvm_pkushort( unsigned short *ip,int nitem,
                         int stride )

int info = pvm_pkulong( unsigned long *ip,int nitem,
                        int stride )

int info = pvm_pklong( long *ip,int nitem,int stride )
int info = pvm_pkshort(short *jp,int nitem,int stride)
int info = pvm_pkstr( char *sp )
```

Synopsis

```
call pvmfpack( what, xp, nitem, stride, info )
```

Arguments

`fmt`	Printf-like format expression specifying what to pack. (See discussion).
`nitem`	The total number of items to be packed (not the number of bytes).
`stride`	The stride to be used when packing the items. For example, if `stride` = 2 in `pvm_pkcplx()`, then every other complex number will be packed.
`xp`	Pointer to the beginning of a block of bytes. Can be any data type, but must match the corresponding unpack data type.

`cp`	Complex array at least `nitem*stride` items long.
`zp`	Double precision complex array at least `nitem*stride` items.
`dp`	Double precision real array at least `nitem*stride` items long.
`fp`	Real array at least `nitem*stride` items long.
`ip`	Integer array at least `nitem*stride` items long.
`jp`	Integer*2 array at least `nitem*stride` items long.
`sp`	Pointer to a null terminated character string.
`what`	Integer specifying the type of data being packed.

what options:

STRING	0	REAL4	4
BYTE1	1	COMPLEX8	5
INTEGER2	2	REAL8	6
INTEGER4	3	COMPLEX16	7

`info` Integer status code returned by the routine. Values less than zero indicate an error.

Description

Each of the pvm_pk*() routines packs an array of the given data type into the active send buffer. The arguments for each of the routines are a pointer to the first item to be packed, `nitem` which is the total number of items to pack from this array, and `stride` which is the stride to use when packing.

An exception is `pvm_pkstr()` which by definition packs a NULL terminated character string and thus does not need `nitem` or `stride` arguments. The Fortran routine `pvmfpack( STRING, ... )` expects `nitem` to be the number of characters in the string and `stride` to be 1.

If the packing is successful, `info` will be 0. If some error occurs then `info` will be < 0.

A single variable (not an array) can be packed by setting `nitem` = 1 and `stride` = 1.

The routine `pvm_packf()` uses a printf-like format expression to specify what and how to pack data into the send buffer. All variables are passed as addresses if `count` and `stride` are specified otherwise, variables are assumed to be values. A BNF-like description of the format syntax is:

```
format : null | init | format fmt
init : null | '%' '+'
fmt : '%' count stride modifiers fchar
fchar : 'c' | 'd' | 'f' | 'x' | 's'
count : null | [0-9]+ | '*'
stride : null | '.' ( [0-9]+ | '*' )
modifiers : null | modifiers mchar
mchar : 'h' | 'l' | 'u'
```

Formats:	+	means initsend – must match an int (`how`) in the param list.
	c	pack/unpack bytes
	d	integers
	f	float
	x	complex float
	s	string
Modifiers:	h	short (int)
	l	long (int, float, complex float)
	u	unsigned (int)

Messages should be unpacked exactly like they were packed to ensure data integrity. Packing integers and unpacking them as floats will often fail because a type encoding will have occurred transferring the data between heterogeneous hosts. Packing 10 integers and 100 floats then trying to unpack only 3 integers and the 100 floats will also fail.

Example

C:

```
info = pvm_initsend( PvmDataDefault );
info = pvm_pkstr( "initial data" );
info = pvm_pkint( &size, 1, 1 );
info = pvm_pkint( array, size, 1 );
info = pvm_pkdouble( matrix, size*size, 1 );
msgtag = 3 ;
info = pvm_send( tid, msgtag );
int count, *iarry;
double darry[4];
pvm_packf("%+ %d %*d %4lf",PvmDataRaw,count,count,iarry,darry);
```

Fortran:

```
CALL PVMFINITSEND(PVMRAW, INFO)
CALL PVMFPACK( INTEGER4, NSIZE, 1, 1, INFO )
CALL PVMFPACK( STRING, 'row 5 of NXN matrix', 19, 1, INFO)
CALL PVMFPACK( REAL8, A(5,1), NSIZE, NSIZE , INFO )
CALL PVMFSEND( TID, MSGTAG, INFO )
```

Errors

The following error conditions can be returned by these functions:

`PvmNoMem` Malloc has failed. Message buffer size has exceeded the available memory on this host.

`PvmNoBuf` There is no active send buffer to pack into. Try calling `pvm_initsend()` before packing message

See Also

`pvm_unpack`(3), `pvm_initsend`(3)

pvm_parent()

Returns the tid of the process that spawned the calling process

Synopsis

```
int tid = pvm_parent( void )
```

Synopsis

```
call pvmfparent( tid )
```

Arguments

`tid` Integer returns the task identifier of the parent of the calling process. If the calling process was not created with `pvm_spawn()`, then `tid` = `PvmNoParent`.

Description

The routine `pvm_parent()` returns the `tid` of the process that spawned the calling process. If the calling process was not created with `pvm_spawn()`, then `tid` is set to `PvmNoParent`.

For hosted PVM applications the host process has the `tid` set to `PvmNoParent`. For hostless applications, the process that assumes the role of the master has the `tid` set to `PvmNoParent`.

Examples

C:

```
tid = pvm_parent();
```

Fortran:

```
CALL PVMFPARENT( TID )
```

Errors

The following error conditions can be returned by `pvm_parent()`:

`PvmNoParent`	The calling process was not created with `pvm_spawn()`.
`PvmSysErr`	Resource management system error.

pvm_perror()

Prints message describing the last error returned by a PVM call

Synopsis

`int info = pvm_perror( char *msg )`

Synopsis

`call pvmfperror( msg, info )`

Arguments

`msg` Character string supplied by the user which will be prepended to the error message of the last PVM call.

`info` Integer status code returned by the routine.Values less than zero indicate an error.

Description

`pvm_perror()` returns the error message of the last PVM call. The user can use `msg` to add additional information to the error message, for example, its location.

Examples

C:

```
if ( pvm_send( tid, msgtag ) ) pvm_perror();
```

Fortran:

```
CALL PVMFSEND( TID, MSGTAG )
IF( INFO .LT. 0 ) CALL PVMFPERROR( 'Step 6', INFO )
```

pvm_probe() — Check if message has arrived

Synopsis

```
int bufid = pvm_probe( int tid, int msgtag )
```

Synopsis

```
call pvmfprobe( tid, msgtag, bufid )
```

Arguments

`tid`	Integer task identifier of sending process supplied by the user.
`msgtag`	Integer message tag supplied by the user. `msgtag` should be ≥ 0.
`bufid`	Integer returning the value of the new active receive buffer identifier. Values less than zero indicate an error.

Description

`pvm_probe()` checks to see if a message with label `msgtag` has arrived from `tid`. If a matching message has arrived `pvm_probe()` returns a buffer identifier in `bufid`. This `bufid` can be used in a `pvm_bufinfo()` call to determine information about the message such as its source and length.

If the requested message has not arrived, then `pvm_probe()` returns with a 0 in `bufid`. If some error occurs `bufid` will be < 0.

A -1 in `msgtag` or `tid` matches anything. This allows the user the following options. If `tid` = -1 and `msgtag` is defined by the user, then `pvm_probe()` will accept a message from any process which has a matching `msgtag`. If `msgtag` = -1 and `tid` is defined by the user, then `pvm_probe()` will accept any message that is sent from process `tid`. If `tid` = -1 and `msgtag` = -1, then `pvm_probe()` will accept any message from any process.

`pvm_probe()` can be called multiple times to check if a given message has arrived yet. After the message has arrived, `pvm_recv()` must be called before the message can be unpacked into the user's memory using the unpack routines.

On the CS-2, `pvm_probe()` uses the high-speed interconnect via the tport layer in the Elan Widget library.

Examples

C:

```
tid = pvm_parent();
msgtag = 4 ;
arrived = pvm_probe( tid, msgtag );
if ( arrived )
   info = pvm_bufinfo( arrived, &len, &tag, &tid );
else
   /* go do other computing */
```

Fortran:

```
 CALL PVMFPROBE( -1, 4, ARRIVED )
 IF ( ARRIVED .GT. 0 ) THEN
    CALL PVMFBUFINFO( ARRIVED, LEN, TAG, TID, INFO )
 ELSE
* GO DO USEFUL WORK
 ENDIF
```

Errors

These error conditions can be returned by pvm_probe():

PvmBadParam	giving an invalid tid value or msgtag.
PvmSysErr	Resource Management System error.

See Also

pvm_nrecv(3), pvm_recv(3), pvm_unpack(3), EW_TPORT(3x), Meiko Elan Widget library documentation set.

pvm_pstat() — **Returns the status of the specified PVM process**

Synopsis

```
int status = pvm_pstat( tid )
```

Synopsis

```
call pvmfpstat( tid, status )
```

Arguments

`tid`	Integer task identifier of the PVM process in question.
`status`	Integer returns the status of the PVM process identified by `tid`. Status is `PvmOk` if the task is running, `PvmNoTask` if not, and `PvmBadParam` if the `tid` is bad.

Description

`pvm_pstat()` returns the status of the process identified by `tid`.

Examples

C:

```
tid = pvm_parent();
status = pvm_pstat( tid );
```

Fortran:

```
CALL PVMFPARENT( TID )
CALL PVMFPSTAT( TID, STATUS )
```

Errors

The following error conditions can be returned by `pvm_pstat()`:

`PvmBadParam`	Bad Parameter most likely an invalid `tid` value.
`PvmSysErr`	Internal error.
`PvmNoTask`	Task not running.

See Also

Meiko Resource Management System document set.

pvm_recv() — Receive a message

Synopsis

```
int bufid = pvm_recv( int tid, int msgtag )
```

Synopsis

```
call pvmfrecv( tid, msgtag, bufid )
```

Arguments

`tid`	Integer task identifier of sending process supplied by the user.
`msgtag`	Integer message tag supplied by the user. `msgtag` should be ≥0.
`bufid`	Integer returns the value of the new active receive buffer identifier. Values less than zero indicate an error.

Description

`pvm_recv`() blocks the process until a message with label `msgtag` has arrived from `tid`. `pvm_recv`() then places the message in a new active receive buffer, which also clears the current receive buffer.

A -1 in `msgtag` or `tid` matches anything. This allows the user the following options. If `tid` = -1 and `msgtag` is defined by the user, then `pvm_recv`() will accept a message from any process which has a matching `msgtag`. If `msgtag` = -1 and `tid` is defined by the user, then `pvm_recv`() will accept any message that is sent from process `tid`. If `tid` = -1 and `msgtag` = -1, then `pvm_recv`() will accept any message from any process.

The PVM model guarantees the following about message order. If task 1 sends message A to task 2, then task 1 sends message B to task 2, message A will arrive at task 2 before message B. Moreover, if both messages arrive before task 2 does a receive, then a wildcard receive will always return message A.

If `pvm_recv`() is successful, `bufid` will be the value of the new active receive buffer identifier. If some error occurs then `bufid` will be < 0.

`pvm_recv`() is blocking which means the routine waits until a message matching the user specified `tid` and `msgtag` values arrives. If the message has already arrived then `pvm_recv`() returns immediately with the message.

Once `pvm_recv`() returns, the data in the message can be unpacked into the user's memory using the unpack routines.

On the CS-2, `pvm_recv`() uses the high-speed interconnect via the tport layer in the Elan Widget library.

Examples

C:

```
tid = pvm_parent();
msgtag = 4 ;
bufid = pvm_recv( tid, msgtag );
info = pvm_upkint( tid_array, 10, 1 );
info = pvm_upkint( problem_size, 1, 1 );
info = pvm_upkfloat( input_array, 100, 1 );
```

Fortran:

```
CALL PVMFRECV( -1, 4, BUFID )
CALL PVMFUNPACK( INTEGER4, TIDS, 25, 1, INFO )
CALL PVMFUNPACK( REAL8, MATRIX, 100, 100, INFO )
```

Errors

These error conditions can be returned by `pvm_recv()`:

`PvmBadParam`	giving an invalid `tid` value, or `msgtag` < -1.
`PvmSysErr`	Resource management system error.

See Also

`pvm_nrecv`(3), `pvm_unpack`(3), `pvm_probe`(3), `pvm_send`(3), `pvm_mcast`(3), `EW_TPORT`(3x).

pvm_send() — Immediately sends the data in the active message buffer

Synopsis

`int info = pvm_send( int tid, int msgtag )`

Synopsis

`call pvmfsend( tid, msgtag, info )`

Arguments

`tid`	Integer task identifier of destination process.
`msgtag`	Integer message tag supplied by the user. `msgtag` should be ≥ 0.
`info`	Integer status code returned by the routine.

`pvm_send()` sends a message stored in the active send buffer to the PVM process identified by `tid`. `msgtag` is used to label the content of the message. If `pvm_send()` is successful, `info` will be 0. If some error occurs then `info` will be < 0.

The `pvm_send()` routine is asynchronous. Computation on the sending processor resumes as soon as the message is safely on its way to the receiving processor. This is in contrast to synchronous communication, during which computation on the sending processor halts until the matching receive is executed by the receiving processor.

The PVM model guarantees the following about message order. If task 1 sends message A to task 2, then task 1 sends message B to task 2, message A will arrive at task 2 before message B. Moreover, if both messages arrive before task 2 does a receive, then a wildcard receive will always return message A.

On the CS-2, `pvm_send()` uses the high-speed interconnect via the tport layer in the Elan Widget library.

Examples

C:

```
info = pvm_initsend( PvmDataDefault );
info = pvm_pkint( array, 10, 1 );
msgtag = 3 ;
info = pvm_send( tid, msgtag );
```

Fortran:

```
CALL PVMFINITSEND( PVMRAW, INFO )
CALL PVMFPACK( REAL8, DATA, 100, 1, INFO )
CALL PVMFSEND( TID, 3, INFO )
```

Errors

These error conditions can be returned by `pvm_send()`:

`PvmBadParam`	giving an invalid `tid` or a `msgtag`.
`PvmSysErr`	Resource management system error
`PvmNoBuf`	no active send buffer. Try `pvm_initsend()` before send.

See Also

`pvm_initsend(3)`, `pvm_pack(3)`, `pvm_recv(3)`, `EW_TPORT(3x)`, Meiko Elan Widget library documentation set.

pvm_sendsig() — Sends a signal to another PVM process

Synopsis

```
int info = pvm_sendsig( int tid, int signum )
```

Synopsis

```
call pvmfsendsig( tid, signum, info )
```

Arguments

`tid`	Integer task identifier of PVM process to receive the signal.
`signum`	Integer signal number.
`info`	Integer status code returned by the routine.

Description

`pvm_sendsig()` sends the signal number `signum` to the PVM process identified by `tid`. If `pvm_sendsig()` is successful, `info` will be 0. If some error occurs then `info` will be < 0.

`pvm_sendsig()` should only be used by programmers with Unix signal handling experience. Many library functions (and in fact the PVM library functions) cannot be called in a signal handler context because they do not mask signals or lock internal data structures.

On the CS-2 signals are sent using the `rms_sigsend()` routine from the resource management user interface library.

Examples

C:

```
tid = pvm_parent();
info = pvm_sendsig( tid, SIGKILL );
```

Fortran:

```
CALL PVMFBUFINFO( BUFID, BYTES, TYPE, TID, INFO );
CALL PVMFSENDSIG( TID, SIGNUM, INFO )
```

Errors

These error conditions can be returned by `pvm_sendsig()`:

`PvmSysErr`	Internal error.
`PvmBadParam`	giving an invalid tid value.

See Also

Meiko Resource Management System document set.

pvm_serror()

Sets automatic error message printing on or off

Synopsis

```
int oldset = pvm_serror( int set )
```

Synopsis

```
call pvmfserror( set, oldset )
```

Arguments

`set` Integer defining whether detection is to be turned on (1) or off (0).

`oldset` Integer defining the previous setting of `pvm_serror()`.

Description

`pvm_serror()` sets automatic error message printing for all subsequent PVM calls by this process. Any PVM routines that return an error condition will automatically print the associated error message. The argument `set` defines whether this detection is to be turned on (1) or turned off (0) for subsequent calls. In the future a value of (2) will cause the program to exit after printing the error message. `pvm_serror()` returns the previous value of `set` in `oldset`.

Examples

C:

```
info = pvm_serror( 1 );
```

Fortran:

```
CALL PVMFSERROR( 0, INFO )
```

Errors

This error condition can be returned by `pvm_serror()`:

`PvmBadParam` giving an invalid set value.

pvm_setrbuf() — Switches the active receive buffer and saves the previous buffer

Synopsis

```
int oldbuf = pvm_setrbuf( int bufid )
```

Synopsis

```
call pvmfsetrbuf( bufid, oldbuf )
```

Arguments

`bufid`	Integer specifying the message buffer identifier for the new active receive buffer.
`oldbuf`	Integer returning the message buffer identifier for the previous active receive buffer.

`pvm_setrbuf()` switches the active receive buffer to `bufid` and saves the previous active receive buffer `oldbuf`. If `bufid` is set to 0 then the present active receive buffer is saved and no active receive buffer exists.

A successful receive automatically creates a new active receive buffer. If a previous receive has not been unpacked and needs to be saved for later, then the previous `bufid` can be saved and reset later to the active buffer for unpacking.

The routine is required when managing multiple message buffers. For example switching back and forth between two buffers. One buffer could be used to send information to a graphical interface while a second buffer could be used to send data to other tasks in the application.

Examples

C:

```
rbuf1 = pvm_setrbuf( rbuf2 );
```

Fortran:

```
CALL PVMFSETRBUF( NEWBUF, OLDBUF )
```

Errors

These error conditions can be returned by `pvm_setrbuf()`;

`PvmBadParam`	giving an invalid `bufid`.
`PvmNoSuchBuf`	switching to a non-existent message buffer.

See Also

`pvm_setsbuf(3)`

pvm_setsbuf() — Switches the active send buffer

Synopsis

```
int oldbuf = pvm_setsbuf( int bufid )
```

Synopsis

```
call pvmfsetsbuf( bufid, oldbuf )
```

Arguments

`bufid` — Integer message buffer identifier for the new active send buffer. A value of 0 indicates the default receive buffer.

`oldbuf` — Integer returning the message buffer identifier for the previous active send buffer.

Description

`pvm_setsbuf()` switches the active send buffer to `bufid` and saves the previous active send buffer `oldbuf`. If `bufid` is set to 0 then the present active send buffer is saved and no active send buffer exists.

The routine is required when managing multiple message buffers. For example switching back and forth between two buffers. One buffer could be used to send information to a graphical interface while a second buffer could be used send data to other tasks in the application.

Examples

C:

```
sbuf1 = pvm_setsbuf( sbuf2 );
```

Fortran:

```
CALL PVMFSETSBUF( NEWBUF, OLDBUF )
```

Errors

These error conditions can be returned by `pvm_setsbuf()`:

`PvmBadParam` — giving an invalid `bufid`.

`PvmNoSuchBuf` — switching to a non-existent message buffer.

See Also

`pvm_setrbuf(3)`

pvm_spawn() — Starts new PVM processes

Synopsis

```
int numt = pvm_spawn(char *task, char **argv,
                     int flag, char *where,
                     int ntask, int *tids)
```

Synopsis

```
call pvmfspawn( task, flag, where, ntask, tids, numt )
```

Arguments

`task` Character string containing the executable file name of the PVM process to be started. The executable must already reside on the host on which it is to be started. The default location PVM looks in is the current directory.

`argv` Pointer to an array of arguments to the executable with the end of the array specified by NULL. If the executable takes no arguments, then the second argument to `pvm_spawn()` is NULL.

`flag` Integer specifying spawn options. In C, `flag` should be the sum of:

Option	Value	Meaning
`PvmTaskHost`	1	`where` specifies a particular host (Not applicable to CS-2)
`PvmTaskArch`	2	`where` specifies a type of architecture (Not applicable to CS-2)
`PvmTaskDebug`	4	Start up processes under debugger
`PvmTaskTrace`	8	Processes will generate PVM trace data. *

In Fortran, `flag` should be the sum of:

Option	Value	Meaning
`PVMHOST`	1	`where` specifies a particular host (Not applicable to CS-2)
`PVMARCH`	2	`where` specifies a type of architecture (Not applicable to CS-2)
`PVMDEBUG`	4	Start up processes under debugger
`PVMTRACE`	8	Processes will generate PVM trace data. *

`where`	Character string specifying where to start the PVM process. On the CS-2 this parameter is currently ignored.
`ntask`	Integer specifying the number of copies of the executable to start up.
`tids`	Integer array of length `ntask` returning the tids of the PVM processes started by this `pvm_spawn()` call.
`numt`	Integer returning the actual number of tasks started. Values less than zero indicate a system error. A positive value less than `ntask` indicates a partial failure. In this case the user should check the `tids` array for the error code(s).

** future extension*

Description

`pvm_spawn()` starts up `ntask` copies of the executable named `task`. `pvm_spawn()` passes selected variables in the parents environment to children tasks. If set, the envar PVM_EXPORT is passed. If PVM_EXPORT contains other variable names (separated by ':') then they will be passed too. For example:

```
setenv DISPLAY myworkstation:0.0
setenv MYSTERYVAR 13
setenv PVM_EXPORT DISPLAY:MYSTERYVAR
```

On return the array `tids` contains the PVM task identifiers for each process started. `numt` will be the actual number of tasks started. If a system error occurs then `numt` will be < 0. `pvm_spawn()` may be called only once.

CS2-PVM negotiates with the Meiko Resource Management System to provide process control. For hosted applications `pvm_spawn()` calls `rms_forkexec()` to spawn `numt` copies of the task on a partition. The partition is identified by the environment variable RMS_PARTITION, or defaults to the partition specified by the System Administrator. For hostless SPMD applications that are loaded onto a partition with `prun(1)` or some other loader, the `pvm_spawn()` executed by the master process does not attempt to create additional processes, as they will already be up and running having been loaded by `prun`.

`pvm_spawn()` tries to synchronise with the slave/node tasks via `pvm_mytid()`. `pvm_spawn()` (on the master/host process) and `pvm_mytid()` (running on the slaves/nodes) both include a barrier synchronisation that prevents any process from continuing until all the others are ready. This ensures that no communcia-tions can be initiated until the underlying communication mechanisms of all processes are in place.

If `PvmTaskDebug` is set then the resource management system will start the task(s) in a debugger. In this case, instead of executing `task args` it executes `$HOME/pvm3/lib/debugger task args`. The `debugger` is a shell script that can run the task under a debugger such as dbx or TotalView. Note that host-less applications cannot spawn a debugger in this way.

Example

C:

```
numt = pvm_spawn("node",(char**)0,0,"",numt,tids );
numt = pvm_spawn("node",(char**)0,PvmTaskDebug,"",numt,tids);
```

Fortran:

```
CALL PVMFSPAWN( 'node',PVMDEFAULT,'0',3,TID(1),NUMT )
FLAG = PVMDEBUG
CALL PVMFSPAWN( 'node', FLAG, '0', 3, TID(1), NUMT )
```

Errors

These error conditions can be returned by `pvm_spawn()` either in `numt` or in the `tids` array:

`PvmBadParam`	giving an invalid argument value.
`PvmNoFile`	specified executable cannot be found. The default location PVM looks in is the current working directory.
`PvmNoMem`	malloc failed. Not enough memory on host.
`PvmSysErr`	Resource management system error.
`PvmOutOfRes`	out of resources.

See Also

Meiko Resource Management System document set, `rms_forkexec`(3x).

pvm_tasks() — Returns information about the tasks running on the CS-2

Synopsis

```
int info = pvm_tasks( int where, int *ntask, struct
                      taskinfo **taskp )
struct taskinfo {
   int ti_tid;
   int ti_ptid;
   int ti_host;
   int ti_flag;
   char *ti_a_out;
} taskp;
```

Synopsis

```
call pvmftasks( where, ntask, tid, ptid, dtid, flag,
                aout, info )
```

Arguments

`where` Integer specifying what tasks to return information about. The options are:

- `0` for all the tasks on the virtual machine
- `pvmd` tid for all tasks on a given host (not applicable to CS-2)
- `tid` for a specific task

`ntask` Integer returning the number of tasks being reported on.

`taskp` Pointer to an array of structures which contain information about each task including its task ID, parent tid, status flag, and the name of this task's executable file. The status flag values are: waiting for a message, and running.

`tid` Integer returning task ID of one task

`ptid` Integer returning parent task ID

`dtid` Integer returning pvmd task ID of host task is on.

`flag` Integer returning status of task

`aout` Character string returning the name of spawned task. Manually started tasks return blank.

`info` Integer status code returned by the routine. Values less than zero indicate an error.

Description

pvm_tasks() returns information about tasks presently running on a partition on the CS-2. The C function returns information about the entire machine in one call. The Fortran function returns information about one task per call and cycles through all the tasks. Thus, if where = 0, and pvmftasks is called ntask times, all tasks will be represented. If pvm_tasks() is successful, info will be 0. If some error occurs then info will be < 0.

Examples

C:

```
info = pvm_tasks( 0, &ntask, &taskp );
```

Fortran:

```
   Do i=1, NTASK
      CALL PVMFTASKS( DTID, NTASK, TID(i), PTID(i), DTID(i),
&     FLAG(i), AOUT(i), INFO )
   EndDo
```

Errors

The following error conditions can be returned by pvm_tasks():

PvmBadParam	invalid value for where argument.
PvmSysErr	Resource management system error.

See Also

pvm_config(3), Meiko Resource Management System document set.

pvm_unpack()

Unpack the active message buffer into arrays of prescribed data type

Synopsis

```
int info = pvm_unpackf( const char *fmt, ... )
int info = pvm_upkbyte(char *xp,int nitem,int stride)
int info = pvm_upkcplx(float *cp,int nitem,int stride)
int info = pvm_upkdcplx(double *zp,int nitem,int stride)
int info = pvm_upkdouble(double *dp,int nitem,int stride)
int info = pvm_upkfloat(float *fp,int nitem,int stride)
int info = pvm_upkint( int *ip, int nitem, int stride)
int info = pvm_upkuint( unsigned int *ip, int nitem,
                        int stride)

int info = pvm_upkushort( unsigned short *ip, int nitem,
                          int stride)

int info = pvm_upkulong( unsigned long *ip, int nitem,
                         int stride)

int info = pvm_upklong(long *ip,int nitem,int stride)
int info = pvm_upkshort(short *jp,int nitem,int stride)
int info = pvm_upkstr( char *sp )
```

Synopsis

```
call pvmfunpack( what, xp, nitem, stride, info )
```

Arguments

`fmt`	Printf-like format expression specifying what to pack. (See discussion).
`nitem`	The total number of items to be packed (not the number of bytes).
`stride`	The stride to be used when packing the items. For example, if `stride` = 2 in `pvm_upkcplx()`, then every other complex number will be unpacked.
`xp`	Pointer to the beginning of a block of bytes. Can be any data type, but must match the corresponding pack data type.
`cp`	Complex array at least `nitem*stride` items long.
`zp`	Double precision complex array at least `nitem*stride` items.
`dp`	Double precision real array at least `nitem*stride` items long.
`fp`	Real array at least `nitem*stride` items long.

`ip`	Integer array at least `nitem*stride` items long.
`jp`	Integer*2 array at least `nitem*stride` items long.
`sp`	Pointer to a null terminated character string.
`what`	Integer specifying the type of data being packed.

what options:

STRING	0	REAL4	4
BYTE1	1	COMPLEX8	5
INTEGER2	2	REAL8	6
INTEGER4	3	COMPLEX16	7

`info`	Integer status code returned by the routine. Values less than zero indicate an error.

Description

Each of the `pvm_upk*()` routines unpacks an array of the given data type from the active receive buffer. The arguments for each of the routines are a pointer to the array to be unpacked into, `nitem` which is the total number of items to unpack, and `stride` which is the stride to use when unpacking.

An exception is `pvm_upkstr()` which by definition unpacks a NULL terminated character string and thus does not need `nitem` or `stride` arguments. The Fortran routine `pvmfunpack(` STRING, ...) expects `nitem` to be the number of characters in the string and `stride` to be 1.

If the unpacking is successful, `info` will be 0. If some error occurs then `info` will be < 0.

A single variable (not an array) can be unpacked by setting `nitem` = 1 and `stride` = 1.

The routine pvm_unpackf() uses a printf-like format expression to specify what and how to unpack data from the receive buffer. All variables are passed as addresses. A BNF-like description of the format syntax is:

```
format : null | init | format fmt
init : null | '%' '+'
fmt : '%' count stride modifiers fchar
fchar : 'c' | 'd' | 'f' | 'x' | 's'
count : null | [0-9]+ | '*'
stride : null | '.' ( [0-9]+ | '*' )
modifiers : null | modifiers mchar
mchar : 'h' | 'l' | 'u'
```

Formats:	+	means initsend – must match an int (how) in the param list.
	c	pack/unpack bytes
	d	integers
	f	float
	x	complex float
	s	string
Modifiers:	h	short (int)
	l	long (int, float, complex float)
	u	unsigned (int)

Messages should be unpacked exactly like they were packed to ensure data integrity. Packing integers and unpacking them as floats will often fail because a type encoding will have occurred transferring the data between heterogeneous hosts. Packing 10 integers and 100 floats then trying to unpack only 3 integers and the 100 floats will also fail.

Example

C:

```
info = pvm_recv( tid, msgtag );
info = pvm_upkstr( string );
info = pvm_upkint( &size, 1, 1 );
info = pvm_upkint( array, size, 1 );
info = pvm_upkdouble( matrix, size*size, 1 );

int count, *iarry;
double darry[4];
pvm_unpackf("%d", &count);
pvm_unpackf("%*d %4lf", count, iarry, darry);
```

Fortran:

```
CALL PVMFRECV( TID, MSGTAG );
CALL PVMFUNPACK( INTEGER4, NSIZE, 1, 1, INFO )
CALL PVMFUNPACK( STRING, STEPNAME, 8, 1, INFO )
CALL PVMFUNPACK( REAL4, A(5,1), NSIZE, NSIZE , INFO)
```

Errors

The following error conditions maybe produced by these functions:

`PvmNoData`	Reading beyond the end of the receive buffer. Most likely cause is trying to unpack more items than were originally packed into the buffer.
`PvmBadMsg`	The received message can not be decoded. Try setting the encoding to PvmDataDefault (see `pvm_mkbuf`()).
`PvmNoBuf`	There is no active receive buffer to unpack.

See Also

`pvm_pack`(3)

Computing
Surface

The Elan Library

S1002–10M131.01

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

Elan Library 1

This chapter describes the Elan Library; the lowest level functional interface to the Elan communications processor and foundation for the Elan Widget library and other higher level communications libraries.

Compilation

Applications using the functions in this library must be linked with `libelan.a` which is installed in the directory `/opt/MEIKOcs2/lib`. In addition Elan library programs reference header files from the standard header file directory (`/usr/include`) and `/opt/MEIKOcs2/include`. A suitable compile command line for Elan programs is:

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

libelan — Elan library

Synopsis

```
#include <elan/elan.h>
```

`libelan` provides the lowest level of access to the Elan Communications Processor.

Parallel Programming

Parallel programs executing under the resource management system will usually use the functions provided by the Elan Widget library or higher level communication libraries (CSN, PVM etc.) to initialise each process. This is because the processes must execute on the resources provided by the partition managers, and support for this is not included in `libelan`.

Parallel programs may however use the low level communication primitives provided by `libelan` to implement high performance or application specific communication protocols. The DMA and event handling routines will therefore be of principle interest to parallel application programmers.

Capabilities

Access to the Elan is controlled via capabilities. A capability describes a physical section of the machine, as a range of processors, and an Elan context number across that range. Capabilities can be created both by the resource management code, and by user applications. When a program tries to communicate the capability is validated to ensure that it is only communicating with other processes holding the same capabilities. This provides the protection mechanism between programs and users.

A capability is defined by the following data structures, defined in the header file `<elan/elanvp.h>`:

```
typedef struct elan_userkey
{
    int         key_vals[4];
} ELAN_USERKEY;
```

```
typedef struct elan_capability
{
    ELAN_USERKEY cap_userkey;
    int cap_context;
    int cap_process;
    int cap_entries;
    int cap_lowElanId;
    int cap_highElanId;
    int cap_routeTable;
} ELAN_CAPABILITY;
```

A process can attach to the Elan using a particular capability. Other processes on potentially different processors can then access this process's memory using the Elan so long as they also hold the same capability.

The 128-bit random key `cap_userkey` ensures that capabilities cannot be forged, `cap_entries` specifies the number of processes, `cap_lowElanId` and `cap_highElanId` specify the range over which the capability is valid and `cap_routeTable` specifies which route table is to be used.

Elan DMA's

The Elan supports a number of different ways of accessing a remote nodes memory, the most common is the DMA processor. The DMA processor is responsible for performing bulk data transfers; it transfers data from the source to the destination by writing into the remote process's address space. At the completion of the data transfer *events* can be set at the source and destination; these are the synchronisation mechanism used by the Elan.

Each DMA is specified by a *descriptor*. The Elan maintains a queue of descriptors which have been submitted, and successively takes *descriptors* of the queue and generates the network transactions to transfer the data. If the DMA is for a large amount of data then the Elan will break the transfer into a number of packets and may reschedule to progress other DMA *descriptors* on the queue.

Events

Events form the synchronisation mechanism for the Elan. Normally an event will be *set* when a data transfer completes. Elan events comprise of two words and must be aligned on a double word boundary. Events are of two types, simple events and queued events (queued events are not considered in this document). Simple events can be in one of three states

State	Description
CLEAR	The event has not been set, and has nothing waiting on it. This is the state that events must be initialised to.
SET	The event has been set. Should anything try to wait or deschedule on the event then it will continue without descheduling and the event will be cleared.
WAITING	Something is descheduled on the event. There are a number of different things which can wait on a event; these are: local/ remote events, threads, DMA's, signals. When the event is set the waiting item will be started and the event will be cleared.

The `libelan` library provides functions for polling for an event to be set, suspending the process on an event, delivering a signal to the process when the event is set, and suspending local events or DMA's on the event. The most common use of events is as a way of indicating that a DMA has completed.

elan_init(), elan_fini(), _elan_fini() Elan library initialisation/finalisation

Synopsis

```
#include <sys/types.h>
#include <elan/elan.h>
void *elan_init (void);
void elan_fini (void *ctx);
void _elan_fini (void *ctx);
```

Description

`elan_init()` provides a handle to access the Elan device driver. **This function is not intended for direct use by parallel applications; the initialisation functions in the Elan Widget library perform this task (see ew_init(3x) and ew_attach(3x)).**

`elan_init()` returns an opaque pointer which can be used in all subsequent calls to `libelan`. The function also checks the revision number of the Elan silicon and reports the following error if it is incompatible.

```
elan: elan is incorrect version 91f != 92f
```

`elan_init()` will return NULL when there are too many processes currently using the Elan, or if there is no virtual address space available to map-in the Elan device.

`elan_fini()` and `_elan_fini()` are used when the process no longer needs to access the Elan. `_elan_fini()` is solely used for a child of a process that has vfork'ed, in that it does not free the opaque structure pointed at by `ctx`. Both functions will implicitly detach the process from the Elan and destroy any capabilities created on this context.

Example

```
void *ctx;

if (!(ctx = elan_init())) {
   fprintf(stderr, "Failed to initialise Elan context");
   exit(1);
}
```

elan_version(), elan_checkVersion() libelan version checking

Synopsis

```
#include <sys/types.h>
#include <elan/elan.h>
#define ELAN_VERSION
char *elan_version (void);
int elan_checkVersion (char *version);
```

Description

`ELAN_VERSION` is a macro which gives the version string of the `libelan` with which the application was compiled.

`elan_version()` returns the version string of the `libelan` with which an application was linked.

`elan_checkVersion()` provides a check that the version of `libelan` against which an application was compiled is compatible with the version with which it was linked. It returns a non-zero value if `version` is a compatible version of the library.

Example

```
if (!elan_checkVersion (ELAN_VERSION))
{
   fprintf (stderr, "libelan version error\n");
   fprintf (stderr, "  Compiled with '%s'\n", ELAN_VERSION);
   fprintf (stderr, "  Linked with   '%s'\n", elan_version ());
   exit (1);
}
```

elan_create(), elan_destroy(), elan_nullcap()   Create/modify/destroy an Elan capability

Synopsis

```
#include <sys/types.h>
#include <elan/elan.h>
int elan_create (void *ctx, ELAN_CAPABILITY *cap);
void elan_destroy (void *ctx, ELAN_CAPABILITY *cap);
void elan_nullcap(ELAN_CAPABILITY* cap);
```

Description

`elan_create()` creates or modifies a capability in the Elan device driver; any process which holds the same capability may then subsequently attach to the Elan or communicate with the attached process via the Elan. **This function is not intended for direct use by parallel applications; the initialisation functions in the Elan Widget library perform this task (see ew_init(3x) and ew_attach(3x)).**

The capability argument `cap` is usually an un-initialised instanced of an ELAN_ CAPABILITY, as returned by `elan_nullcap`(3x). The following fields will be initialised by this function if they were previously unassigned:

```
cap_lowElanId     node-id
cap_highElanId    node-id
cap_context       free-context-number
```

The fields of a capability can be modified by subsequent calls to `elan_create`() if the `ctx` parameter is the one used to create the capability in the first place. `elan_create`(3x) returns a value of 0 on failure.

`elan_destroy`() destroys capabilities previously created by `elan_create`(). Any process trying to attach with that capability will be refused. If a process is already attached the context will become free when that process detaches. If the capability argument to `elan_destroy`() is NULL then all capabilities created using this `ctx` will be destroyed. This is done implicitly when the process exits or calls `elan_fini`().

Example

```
void *ctx;
ELAN_CAPBILITY *cap;

cap = (ELAN_CAPABILITY*) malloc(sizeof(ELAN_CAPABILITY));
```

```
ctx = elan_init();
elan_nullcap(cap);

if(elan_create(ctx, cap) < 0) {
   fprintf(stderr, "Failed to create capability\n");
   exit(1);
}
```

elan_attach(), elan_detach() — Attach to, or detach from, the Elan

Synopsis

```
#include <sys/types.h>
#include <elan/elan.h>
int elan_attach (void *ctx, ELAN_CAPABILITY *cap);
void elan_detach (void *ctx);
```

Description

`elan_attach()` is used to attach the process with `ctx` into the Elan. **This function is not intended for direct use by parallel applications; the initialisation functions in the Elan Widget library perform this task (see ew_init(3x) and ew_attach(3x)).**

`elan_attach()` will map the whole of the process's address space into the Elan and allows any process that also holds the capability `cap` to access the process's memory through the Elan.

The fields of the capability are checked against the capabilities that have been previously created with `elan_create()`. Should the capability not be found or not match then `elan_attach()` will fail. On failure a value of -1 is returned and set `errno` as follows

EBUSY	`elan_attach()` has already been called by this process, or another process has already attached with this capability.
EACCES	`cap->cap_userkey` did not match the one specified by `elan_create()`.
EINVAL	The `cap->cap_context`, `cap->cap_lowElanId` or `cap->cap_highElanId` did not match the ones specified by `elan_create()`.
ENOMEM	`cap->cap_userkey` did not match the one specified by elan_create().

`elan_detach()` is used to detach the process from the Elan group that it had previously attached to. After calling `elan_detach()` the process will not be able to communicate with other processes using the Elan. The Elan state will be preserved, and may be reinstated by calling `elan_attach()`.

elan_addvp(), elan_removevp() Add/remove virtual process segments

Synopsis

```
#include <sys/types.h>
#include <elan/elan.h>
int elan_addvp (void *ctx, ELAN_CAPABILITY *cap);
int elan_removevp (void *ctx, int process);
```

Description

`elan_addvp()` adds a section of virtual process numbers to the context. **This function is not intended for direct use by parallel applications; the initialisation functions in the Elan Widget library perform this task (see ew_init(3x) and ew_attach(3x)).**

The virtual process numbers that are used to communicate are in the range `cap_process` to `cap_process+cap_entries-1`, and these map to the physical location of the processes as defined by `cap_lowElanId`, `cap_highElanId`, and `cap_context`.

The capability is validated against that held by the destination process when the first packet is opened. Should it not match then the program will take an invalid process exception.

If `cap_process` is specified as ELAN_CAP_UNITITALISED then a value will be chosen such that the range does not overlap with previously added segments.

elan_addrt() — Add a broadcast virtual process

Synopsis

```
#include <elan/elan.h>
int elan_addrt (void *ctx, int process, int entries);
```

Description

`elan_addrt()` adds a virtual process that can be used to broadcast across the processes `[process,process+entries-1]`. **This function is not intended for direct use by parallel applications; the ew_createBcastVp(3x) function in the Elan Widget library performs this task.**

Packets opened to this virtual process will use the hardware broadcast supported by the Elan/Elite network. The range of processes to broadcast over must have been previously specified by a single call to `elan_addvp`(3x) — which for parallel programs is performed by the `ew_attach`(3x) Elan Widget function.

It is not permissible to broadcast across multiple segments of an application.

The function returns the virtual process number to use for the broadcast. On error the function returns ELAN_INVALID_PROCESS, and will set `errno` appropriately.

EINVAL	The process has not called `elan_attach`(), the range of processes does not match a previous segment defined by `elan_addvp`(3x), or `entries` is less than 0.
ENOMEM	There is insufficient space in the Elan route tables to create this route.

elan_dma() — Queue a DMA descriptor on the Elan

Synopsis

```
#include <elan/elan.h>
void elan_dma (void *ctx, ELAN_DMA *dma);
```

Description

elan_dma() queues a DMA on the Elan.

The DMA is defined by the following descriptor, defined in <elan/dma.h>. Note that descriptors must be 32-bit aligned, and so must be created either by memalign(), or with the Elan Widget ew_allocate() function. The DMA descriptor must not be altered until the DMA has completed.

```
typedef struct elan_dma
{
    union elan_dma_type            dma_u;
    unsigned int                   dma_size;
    void                          *dma_source;
    void                          *dma_dest;
    volatile struct elan_event    *dma_destEvent;
    unsigned int                   dma_destProc;
    volatile struct elan_event    *dma_sourceEvent;
    unsigned int                   dma_pad;
} ELAN_DMA;

#define dma_type        dma_u.type
```

Field	Description
dma_u	The transaction type. The DMA_TYPE() macro, defined in <elan/dma.h>, simplifies the setting of this field. This is described below.
dma_size	Size of the transfer.
dma_source	A pointer to the source data in the sending process's address space.
dma_dest	A pointer to the receivers data buffer in the receiver's address space.
dma_destEvent	The event to set at the receiving processor when the DMA has completed.

Field	Description
dma_destProc	The process number of the receiving process.
dma_sourceEvent	The event to set at the sending process when the DMA has completed.
dma_pad	Unused.

The DMA type can be set with the DMA_TYPE() macro. This takes three arguments: one of the transaction types defined in <elan/transaction.h>, a mode of operation, and an integer retry-on-error count. The mode of operation is either DMA_NORMAL or DMA_SECURE; in secure mode DMA transfers are not acknowledged all DMA network packets have arrived, whereas normally they are acknowledged as the first arrives. The transaction type is used to describe the alignment of the data and with the dma_size field to determine the size of the transfer; it is one of:

- TR_TYPE_BYTE — 8 bit data object (C type char).
- TR_TYPE_SHORT — 16 bit data object (C type short).
- TR_TYPE_WORD — 32 bit data object (C type int).
- TR_TYPE_DWORD – 64 bit data object (C type long long).

The Elan will perform the data transfer and set the completion events. The descriptor should not be changed until either of the completion events have been set. Note that you can use a DMA of size 0 to set remote events without transferring data.

The virtual process that the DMA will transfer data to is defined by previous calls to elan_addvp(3x), or elan_addrt(3x) for this context. Typically, for parallel applications, these will be called indirectly by Elan Widget library functions.

Example

Send 1024 bytes to process 1, transferring the data from `mybuffer` (sender's address space) to `destbuffer` (recipient's address space). Set events to awake both the sender and the recipient when the transfer completes.

```
/* Build the DMA descriptor */
dmaDesc->dma_type = DMA_TYPE(TR_TYPE_BYTE, DMA_NORMAL, 8);
dmaDesc->dma_size = 1024;
dmaDesc->dma_source = &mybuffer;
dmaDesc->dma_dest = &destbuffer;
dmaDesc->dma_destEvent = &destevent;
dmaDesc->dma_destProc = 1;
dmaDesc->dma_sourceEvent = &myevent;

/* Initiate DMA; the event signifies completion. */
elan_dma(ew_ctx, dmaDesc);
elan_waitevent(ew_ctx, myevent, ELAN_POLL_EVENT);
```

Example

Set the remote event at address `destevent` in the address space of process 1:

```
dmaDesc->dma_type = DMA_TYPE(TR_TYPE_BYTE, DMA_NORMAL, 1);
dmaDesc->dma_size = 0;
dmaDesc->dma_source = NULL;
dmaDesc->dma_dest = NULL;
dmaDesc->dma_destEvent = &destevent;
dmaDesc->dma_destProc = 1;

/* Set the remote event. */
elan_dma(ew_ctx, dmaDesc);
```

elan_setevent(), elan_waitevevent() Set or wait for an event

Synopsis

```
#include <elan/elan.h>

ELAN_CLEAREVENT(ELAN_EVENT *event);

void elan_waitevent (void *ctx, ELAN_EVENT *event,
                     int how);

void elan_setevent (void *ctx, ELAN_EVENT *event);
```

Description

ELAN_CLEAREVENT() is a macro which initialises an event. It is normally only required for initialising events which have been dynamically allocated or declared on the stack.

elan_setevent() sets an event. If something was waiting on the event then the Elan will schedule it. If nothing is waiting then the event will be left in the set state.

elan_waitevent() waits for the event to be set; when the event is set elan_waitevent returns after clearing the event. If the event is set before the call to elan_waitevent() the function returns immediately (after clearing the event).

The parameter how determines whether the event is polled until it is ready or whether the process deschedules and voluntarily relinquishes the processor. There are two macros defined <elan/event.h> for use with the how field: ELAN_POLL_EVENT and ELAN_WAIT_EVENT. If the process deschedules it will take some time from the event being set until the process returns from the call to elan_setevent() call; this is because the kernel needs to reschedule the process. If a communication is expected to complete quickly then the event is best polled.

An environment variable `ELAN_WAITEVENT_MODE` allows the `elan_wait-event()` function to provide information if the event is not set. It is a bit mask defined as follows:

Bit 0	Flash mode. The front-panel LEDs display a cycling pattern if the event is not set.
Bit 1	Abort mode. The program prints a message and executes the `abort()` system call if the event is not set.

Example

The following call to `elan_waitevent()` will deschedule the calling process until the event `myevent` is set. The context `ew_ctx` is initialised by start-up functions in the Elan Widget library.

```
ELAN_EVENT myevent;

...

ELAN_CLEAREVENT(&myevent);
elan_waitevent(ew_ctx, &myevent, ELAN_WAIT_EVENT);
```

elan_waiteventevent(), elan_waitdmaevent() Wait a DMA on an event

Synopsis

```
#include <elan/elan.h>
void elan_waitdmaevent (void *ctx, ELAN_DMA *dma,
                        ELAN_EVENT *event);
void elan_waiteventevent (void *ctx,
                          ELAN_EVENT *chained,
                          ELAN_EVENT *event);
```

Description

`elan_waitdmaevent()` suspends a DMA pending the event. When the event is set then the DMA descriptor pointed at by `dma` will be queued on the Elan. The event will then be left clear. If the event was set when `elan_waitdmaevent()` was called then the DMA descriptor is queued immediately and the event is left cleared.

This mechanism allows you to chain DMA's together and to suspend on a single event to wait for them all to complete. The DMA's would execute sequentially and chain through each other, setting a single event when they have all completed.

`elan_waiteventevent()` allows an event to wait on another event; when the `event` is set the event pointed to by `chained` is set. The event pointed to by `event` will be left clear. This function allows you to implement *alting* for one of many different communications to complete.

elan_runthread() — **Schedule a thread to run on the Elan**

Synopsis

```
#include <elan/elan.h>
void elan_runthread (void *ctx, void (*fn)(),
                     caddr_t stack, int stacksize,
                     int nargs ...);
```

Description

`elan_runthread()` schedules a thread to run on the Elan's thread processor. The thread executes the function `fn` passing it `nargs` parameters. The thread executes using the stack specified by `stack` and `stacksize`.

The function `fn` should be compiled using the Elan threads processor compiler, and it can call any of the inline intrinsic functions to execute the Elan instructions for scheduling and preparing packets. A description of programming styles for the Elan threads processor is beyond the scope of this document.

elan_clock() — Read the elan nano-second clock

Synopsis

```
#include <elan/elan.h>
void elan_clock (void *ctx, ELAN_TIMEVAL *tv);
```

Description

`elan_clock()` reads the nano-second realtime (wallclock) clock on the Elan. It returns the current time in the structure pointed to by `tv`. The structure has the following members

```
typedef struct elan_timeval
{
    lond tv_nsec;
    long tv_sec;
} ELAN_TIMEVAL;
```

Examples 2

Introduction

Two examples are included in this chapter showing how the Elan Library's DMA and event functionality can be embedded within an Elan Widget Library application and a CSN message passing application.

Using with the Elan Widget Library

In this example the Elan library functions are sandwiched between Elan Widget Library initialisation and clean-up functions.

The Elan Widget library is a layer above the Elan Library; it provides a set of higher level parallel programming constructs that augment the basic capabilities of the Elan/Elite hardware. For many applications the Widget Library's performance and generality will be sufficient. Where gains in performance are vital time critical components of the Widget Library application may be implemented with Elan Library functions.

In the following example the Elan Widget library is used to handle the process initialisation and the creation of the Global Data Objects[1]. The Elan library's DMA and Event functionality is used to handle the inter-process communication.

1. Global Objects are data structures that exist at the same virtual address on all processes.

For a description of the Widget library see *The Elan Widget Library*, Meiko document number S1002–10M104.

Program Description

Process Initialisation

The program is initialised with the Widget library function `ew_baseInit()`. This function performs process initialisation, attachment to the Elan network, and definition of virtual process addresses. It also defines some useful parallel programming objects which are packaged within an `ew_base` structure; in this example we will use the `segGroup` (group of processes in this application) and `alloc` (area of global memory) definitions.

The DMA descriptor, data buffer, and the event structure are allocated as global objects from within the alloc region defined by the Widget library. The use of global objects is fundamental to the simplicity of this example; by defining the buffer and event as global objects they will exist at the same virtual address on all processes, allowing the sending process to address the receiver's data buffer and event without explicit handshaking.

Having defined the global objects the processes barrier synchronise using the Widget function `ew_gsync()`. This ensures that none of the processes proceed until the global objects have been defined (and prevents, in this example, the sender from initiating a transfer into unallocated memory).

Elan DMA/Event Functionality

The process with virtual process number 0 will be the sending process, so this initialises the DMA descriptor to describe the transfer. A block of memory will be transferred from the buffer in the sender's address space to the buffer in the recipients address space (the buffer is initialised with a pattern so the integrity of the received data can be verified).

The type of DMA transfer is described by the macro `DMA_TYPE()`. In this example the transfer size of the DMA refers to a number of bytes (`TR_TYPE_BYTE`), the op-code is `DMA_NORMAL`, and the fail-retry count is set to 8. The op-code is used to specify when the DMA is flagged as complete; with `DMA_NORMAL` the recipient acknowledges receipt as soon as the first DMA network packet is received (with DMA_SECURE the acknowledge is sent after the last packet is received).

Both a source and destination event are specified so that both processes are notified when the DMA has completed. The source and destination event structures exist at the same virtual address space in both processes, so the same address is specified in both fields of the DMA descriptor.

Process 0 initiates the DMA with elan_dma(), using the context that is initialised with the Widget library. The process is delayed until the event is set — because the DMA will complete quickly it is more efficient to poll the event (ELAN_POLL_EVENT) than to suspend the process and wait for it (ELAN_WAIT_EVENT).

Process 1 simply waits until its own event is set signifying completion of the DMA. Checking the receiver's data buffer will confirm the same data pattern as the sender.

Finalisation

Both processes synchronise and then free their global objects.

Compilation and Execution

To compile the program use the following command line:

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

You can run the program with prun (in this case in the parallel partition):

```
user@cs2: prun -n2 -p parallel elandma
Process 0 now transferring 1024 bytes by DMA
Data received and verified by process 1
```

The Program

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

#define DMASIZE 1024

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


main()
{
   int me, nproc, i;
   ELAN_DMA *dmaDesc;
   ELAN_EVENT* event;
   EW_ALLOC* alloc;
   unsigned char* buffer;

   /*********** Widget library initialisation functions ****************/

   ew_baseInit();

   nproc = ew_base.segGroup->g_size;
   me = ew_base.segGroup->g_self;
   alloc = ew_base.alloc;

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

   if(!(dmaDesc = (ELAN_DMA*) ew_allocate(alloc, EW_ALIGN, sizeof(ELAN_DMA))) ||
      !(buffer = (unsigned char*) ew_allocate(alloc, EW_ALIGN, DMASIZE)) ||
      !(event = (ELAN_EVENT*) ew_allocate(alloc, EW_ALIGN, sizeof(ELAN_EVENT))))
   {
      fprintf(stderr, "Failed to allocate\n");
      exit(1);
   }

   ew_gsync(ew_base.segGroup);

   /******************** End of Initialisation *********************/
```

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

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

ELAN_CLEAREVENT(event);

if(me == 0) {
   /* Processor 0 is the 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 = buffer;
   dmaDesc->dma_destEvent = event;
   dmaDesc->dma_destProc = 1;
   dmaDesc->dma_sourceEvent = event;

   /* Initiate DMA; the event signifies completion. */
   printf("Process %d now transferring %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 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);
}
/***************** End of Elan Library Functions ****************/
```

```
    /*************** Widget library clean-up *******************/

    ew_gsync(ew_base.segGroup);

    ew_free((void*) event);
    ew_free((void*) dmaDesc);
    ew_free((void*) buffer);
    exit(0);
}
```

Using with the CSN Library

In this example the Elan library's DMA and event functions are sandwiched between CSN initialisation and clean-up functions. The CSN library is an example of a message passing library — the concepts illustrated here will be equally applicable to other messages passing systems.

The CSN library is a layer above the Elan Widget library (which in turn is built upon the Elan library). It provides a high level message passing interface to the Elan/Elite hardware. For performance critical sections of an application it may be desirable to make direct reference to either Widget library functions or the Elan library.

In the following example the CSN library is used to handle the process initialisation and synchronisation. The addresses of remote data structures are explicitly communicated to the sending process by using the CSN message passing functions. These addresses are then used as the target for a remote DMA transfer.

For a description of the CSN interface see the *CSN Communications Library*, Meiko document number S1002–10M106.

Program Description

The processes initialise with `csn_init()` and get their virtual process id and the number of processes in the application from `cs_getinfo()`.

The DMA descriptor, event data structure, and the data buffer are created in each process's local heap. There are two points to note here. Firstly the DMA descriptor must be 32 bit aligned. The second point is that the sender of the DMA transfer must explicitly obtain the address of the remote data buffer and event; compare this with the previous Elan Widget example in which each process allocates space with `ew_allocate()` and can assume that each process's data structure will exist at the same address[1].

1. A CSN program could use the Elan Widget allocation functions to create global objects and thus avoid the need for explicit communication of buffer addresses.

Both processes in this example open a transport; process 1 uses it's transport to communicate to process 0 the address of it's event structure and data buffer. Having obtained the remote addresses process 0 can use the Elan library DMA/event functionality to transfer a block of initialised data directly into the receiver's address space — using the same code as the previous Widget library example.

Compilation and Execution

To compile the program use the following command line:

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

You can run the program with `prun` (in this case in the `parallel` partition):

```
user@cs2: prun -n2 -p parallel csndma
Process 0 now transferring 1024 bytes by DMA
Data received and verified by process 1
```

The Program

The use of Elan functions in this program is identical to the Widget library example described earlier, except the address of the remote data buffer and event is that obtained by the CSN communications.

```
#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));
}

   /***************** End of CSN Initialisation **************/
```

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

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

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);
}
/****************** End of Elan functions ****************/
```

```
    /***************** CSN library clean-up ************************/

    free(buffer);
    free(dmaDesc);
    free(event);
    csn_exit(0);
}
```

Computing Surface

Group Routing

S1002–10M124.01

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

Group Routing 1

Introduction

This document briefly outlines the implementation of Group Routing on the Meiko CS-2 (Solaris 2.X) operating system. The design of group routing presented here is a logical extension of the scheme devised by Lawrence Livermore National Laboratories (LLNL).

The Solaris kernel maintains a routing table that is built at runtime via the actions of daemons and explicit route commands. This table holds all the TCP/IP routing information. Conceptually this table is a list of ordered pairs:

<address template 1>	<gateway address>
<address template 2>	<gateway address>
<address template 3>	<gateway address>
...	...
<any address>	<default gateway>

The address templates can represent several different types of route; broadcasts, loopback, networks, subnets, and hosts.

When a user issues a system call that causes a packet to be sent out on the network, the system looks at the *destination* address of the packet. This address is compared sequentially against all the address templates in the routing table. If a match is found then the packet will be sent to the corresponding gateway address.

If no match is found then the packet will be sent to the default gateway, if such a route has been configured. Otherwise the packet is dropped and an error is reported to the system call.

With Group Routing the route table is augmented:

<address template 1>	<gateway address>	<gid list>
<address template 1>	<gateway address>	<gid list>
<address template 1>	<gateway address>	<gid list>
...	...	...
<any address>	<default gateway>	<gid list>

`gid list` is a list of group ids. This list may be either "positive", which allows all listed groups to access that route, or "negative", which denies access to the listed groups. The kernel lookup algorithm is extended so that a route is only found if the destination address matches the address template **and** the sender is allowed to use that route (as specified by the gid list). A user is permitted access to a route if any of their gid's match (i.e. their real gid or any of their supplemental gids). Senders with a root uid are always permitted access.

Three Solaris commands have also been extended to support the group routing; the `route(1m)` command is used to add the group lists into the route table, the `netstat(1m)` command is used to display the route table and associated gid lists, and the `ifconfig(1m)` command is used to assign a gid to network interfaces — the latter command is used when data must be forwarded from an external network where the sender's gid cannot otherwise be determined.

Implementation

There are six types of IP traffic that need to be considered:

1. IP packets originating from the local node.
2. IP packets originating externally and requiring forwarding.
3. IP broadcast packets originating locally.
4. IP broadcast packets originating externally and requiring forwarding.
5. IP multicast packets originating from the local node.
6. IP multicast packets originating externally and requiring forwarding.

Warning – group routing is only relevant to out-going packets, all in-coming packets destined for the local node are not validated.

Packets Originating from the Local Node

Packets from the local node are the most obvious in terms of implementing the group routing strategy. By amending the kernel routing tables to include a list of group ids (gids), the standard IP routing algorithm can be amended to match the sender's group id as well as the target IP address. This allows the Administrator to define exactly which routes a particular group of users can use. The kernel's routing tables contain several different types of entry: broadcasts, networks, subnets, gateways, and hosts. All these types of route entry will be subject to group routing, allowing the Administrator to control access to individual hosts as well as complete networks.

Warning – the sender's gid is stored when the stream is opened and is not updated during the lifetime of the communication. The group routing is not updated if the sender's process changes group.

External Packets Requiring Forwarding

The control of packets that originate externally to a node is more difficult but is fundamental to the operation of the CS-2.

CS-2 machines are built from many processing elements each running a separate instance of the Solaris kernel. All processing elements within the CS-2 are interconnected by the Elan/Elite network; some of the processing elements, called *gateway nodes*, will also be connected to local networks.

IP forwarding must be functional at the gateway nodes, however a forwarding gateway node has no way of determining the original sender's group id. For packets originating within the CS-2 (that is, those arriving via the Elan/Elite network) it is guaranteed that group routing was performed at the source node; it is therefore safe to forward these packets without further checking. For external networks this assumption cannot be made. Rather than inhibit the forwarding of these packets, which would be too restrictive for most applications, group ids are assigned to each network interface and are inherited by incoming packets. This strategy allows the same routing checks to be used as for the local packets, and also allows the System Administrator to effectively partition network segments — packets arriving from a network interface can be prevented from being forwarded to other networks.

For example, a CS-2 may be connected to 4 external networks: NET_A, NET_B, NET_C, and NET_D. By creating new group ids to represent these networks a matrix of routing permissions can be implemented:

	NET_A	NET_B	NET_C	NET_D
NET_A	Y	Y	N	N
NET_B	Y	Y	N	N
NET_C	N	N	Y	Y
NET_D	N	N	Y	Y

The above table shows that users can use the CS-2 to route between networks A and B (and B to A), and between C and D; users on networks A or B cannot route into networks C or D. By default through routing will not be allowed. The default gid assigned to network interfaces is `nobody` — only by adding `nobody` to an outgoing route, or `+everyone`, will packets be forwarded through the CS-2 from these interfaces.

Warning – security can be compromised by routing external networks through non-gateway CS-2 nodes. All through-routing should pass direct from the incoming gateway node to the outgoing gateway node.

Broadcast Packets Originating Locally

Broadcast packets originating locally to the node should ideally be treated in the same way as non-broadcast packets, however the broadcast routes are created dynamically by the kernel and cannot be changed or deleted by the `route` command.

To give the System Administrator control over broadcast routes a default group list is used. The default group list is the access list associated with any routes that have not been explicitly given group routing information. For security reasons the default group list is defined to allow access to no-one. The kernel has been modified to allow this default list to be amended via the `route` command (see the reference to default routes in Section *route(1m)* on page 8).

External Broadcast Packets Requiring Forwarding

This type of packet is treated in the same way as *External Packets Requiring Forwarding*, described above.

Local and External Multicast Packets

To simplify the initial group routing implementation multicast packets, either originating locally or externally, are disallowed. The CS-2 will not perform any multicast forwarding, and will only allow the superuser to send multicast packets.

Group Routing Administration 2

Start of day configuration

By default the kernel will boot with group routing enabled. In order to configure group routing a new file called `/etc/groutes` is executed when the system is rebooted. If this file is not present and executable then group routing will be disabled and the machine will resort to the normal TCP/IP routing scheme. If present this file should contain all the `route` and `ifconfig` commands necessary to enable normal user access to the machine. As a minimum it must configure the Elan network adaptor (`elanip0`) to have a group id of `root`, and also allow `+everyone` access to the Elan network.

Defaults Summary

- To allow system maintenance and normal daemon operation the `root` gid will bypass all group routing checks.
- All routes have a default gidlist that will apply unless explicitly specified by the `route` command. For security reasons the default gidlist is `-everyone`, which excludes everyone but `root`.
- All network interfaces have a default gid that will apply unless explicitly specified by the `ifconfig` command. For security reasons the default gid is `nobody`.

Commands

Two commands are used to administer the group routing strategy. They are Meiko extended versions of the standard Solaris commands `ifconfig(1m)` and `route(1m)`. A third command, `ndd(1m)`, allows group routing to be enabled or disabled.

ifconfig(1m)

The synopsis for the extended `ifconfig(1m)` command is:

```
ifconfig interface [ address_family ] [ address [ dest_address ] ]
[ netmask mask ] [ broadcast address ] [ up ] [ down ]
[ trailers ] [ -trailers ] [ arp ] [ -arp ] [ private ]
[ -private ] [ metric n ] [ mtu n ] [ auto-revarp ] [ plumb ]
[ group groupname ]
```

Where *groupname* is a valid group name in the `/etc/groups` file or NIS map. By default all adaptors are initialised with a gid of `nobody`. The gid `root` is a special group which bypasses all group routing checks.

The following example usage of `ifconfig` applies a gid of `root` to the Elan network interface:

```
cs2-0# ifconfig elanip0 group root
```

route(1m)

The synopsis for the extended `route(1m)` command is:

```
route [ -fn ] [ -g +|-gidlist ] add | delete [ host | net ]
destination [ gateway [ metric ] ]
```

Where *gidlist* is a comma separated list of one or more group names (from `/etc/groups` or NIS map). There must be no whitespace in this list, either after the initial +/- or between each group name. The initial +/- defines whether the list is an access or deny list. If + then only the groups listed will be allowed access to that route; if - then only the groups listed will be denied access to that route. Only one group list per command is valid. There is a special group name called `everyone` that can be used to define lists that include or exclude all groups — for example, `+everyone` will allow all groups access, and `-everyone` will deny all groups access (except root).

Warning – the group list flag must appear before the add/delete part of the command. This is better suited to the original command syntax and command line validation. This is not compatible with the LLNL specification.

All route entries with an undefined group list use the default group list, which is `-everyone`. The System Administrator can change this default by specifying `default` as both the destination and gateway addresses; note that the metric shown in the following command line is ignored:

```
cs2-0# route -g +everyone add default default 0
```

This is not the same as setting the group list for a default route (where only the destination is specified as `default`).

The route command may also be used to change the group list for routes that already exist. The following example changes the group list for the local network `meiko-net` on the machine `spin`.

```
cs2-0# route -g +meiko,staff add meiko-net spin 0
```

This causes the old group list to be deleted and be replaced by the new list. Only the group list is changed, all the other route parameters are left untouched.

netstat(1m)

The `netstat (1m)` command has been extended to display the gid lists associated with each route. To display this information the following command line should be used. This will dump out the kernel IP route table and the corresponding group lists in symbolic format, as shown below. Note that only the first 16 groups of each route's gid list will be displayed.

```
root@cs2-0# netstat -rv

IRE Table:

Destination    Mask            Gateway      Device    MxFrg Rtt  Ref Flg Out  In/Fwd  Groups

localhost   255.255.255.255 localhost    lo0        8232* 512    0  UH 3107       0 -everyone
godiva-net  255.255.255.0   godiva0-le0             1500* 512    0  UG    0       0 -everyone
cs2-net     255.255.255.0   cs2-0        elanip0  69554* 512    3  U     0       0 -everyone
meiko-net   255.255.255.0   cs2-0-le0    le0        1500* 512    2  U    29       0 -everyone
224.0.0.0   240.0.0.0       cs2-0        elanip0  69554* 512    3  U     0       0 -everyone
default     255.0.0.0       telstar                 1500* 512    0  UG    0       0 -everyone
```

ndd(1m)

Group routing can be enabled and disabled using the `ndd` command on the IP module. If the parameter `ip_group_routing` is non-zero then group routing is enabled.

```
ndd -set /dev/ip ip_group_routing 1     # enable group routing
ndd -set /dev/ip ip_group_routing 0     # disable group routing
```

The `ip_ire_status` function has also been modified to display the group lists associated with each route entry.