INTEGRATED SYSTEMS

pSOSystem
PROGRAMMER'S REFERENCE

68K Processors

MRI Release

Copyright © 1996 Integrated Systems, Inc. All rights reserved. Printed in U.S.A.

Integrated Systems, Inc. • 3260 Jay Street • Santa Clara, CA 95054-3309
Support: 408-980-1500, x501 or 1-800-458-7767
FAX: 408-980-0400 (corporate); 408-980-1647 (support)
e-mail: psos_support@isi.com • Home Page: http://www.isi.com

Document Title: **pSOSystem Programmer's Reference**
Part Number: **000-5078-001**
Revision Date: **March 1996**

LICENSED SOFTWARE - CONFIDENTIAL/PROPRIETARY

This document and the associated software contain information proprietary to Integrated Systems, Inc., or its licensors and may be used only in accordance with the Integrated Systems license agreement under which this package is provided. No part of this document may be copied, reproduced, transmitted, translated, or reduced to any electronic medium or machine-readable form without the prior written consent of Integrated Systems.

Integrated Systems makes no representation with respect to the contents, and assumes no responsibility for any errors that might appear in this document. Integrated Systems specifically disclaims any implied warranties of merchantability or fitness for a particular purpose. This publication and the contents hereof are subject to change without notice.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS252.227-7013 or its equivalent. Unpublished rights reserved under the copyright laws of the United States.

TRADEMARKS

The following are trademarks of Integrated Systems, Inc.:

ESp, OpEN, pHILE+, pNA+, pREPC+, pRISM, pROBE+, pRPC+, pSOS, pSOS+, pSOS+m, pSOSim, pSOSystem, pX11+, SpOTLIGHT.

All other products mentioned are the trademarks, service marks, or registered trademarks of their respective holders.

Contents

About This Manual

Purpose vii
Audience vii
Organization vii
Related Documentation viii
Notation Conventions ix

1 System Services

bootpc 1-3
bootpd 1-5
FTP Client 1-11
FTP Server 1-19
Loader 1-23
mmulib 1-39
NFS Server 1-47
pSH+ 1-51
RARP 1-73

routed 1-75
Telnet Client 1-79
Telnet Server 1-85
TFTP Server 1-87

2 Interfaces and Drivers

NI 2-3
KI 2-21
DISI 2-31
DISIplus 2-57
SCSI 2-93
SLIP 2-101

3 Configuration Tables

Node 3-3
Multiprocessor 3-5
pSOS+ 3-9
pROBE+ 3-15
pHILE+ 3-25
pREPC+ 3-29
pNA+ 3-31
pRPC+ 3-39

4 Memory Usage

pSOS+ 4-3
pHILE+ 4-7
pREPC+ 4-9
pNA+ 4-11
pRPC+ 4-15

Appendix A: Assembly Language Information

pSOS+ A-3
I/O A-11
pHILE+ A-15

Index

About This Manual

Purpose

This manual is part of a documentation set that describes pSOSystem, the modular, high-performance real-time operating system environment from Integrated Systems.

This manual documents pSOSystem services and provides important reference material on device drivers, configuration tables, error codes, and memory usage.

For conceptual information or information on other areas of pSOSystem, refer to the other manuals of the pSOSystem basic documentation set, which include the *pSOSystem Installation Guide*, *pSOSystem Getting Started*, *pSOSystem System Concepts*, *pSOSystem System Calls*, and the *pROBE+ User's Guide*.

Audience

This manual is targeted primarily for embedded application developers who want to implement pSOSystem. Basic familiarity with UNIX terms and concepts is assumed.

Organization

This manual is organized as follows:

Section 1, "System Services," describes pSOSystem system services, such as boot ROMs, FTP Client and FTP Server, pSOSystem Loader, NFS Server, pSH+ command line interface, Telnet Client and Telnet Server, and TFTP Server.

Section 2, "Interfaces and Drivers," describes the pSOSystem Network Interface and Kernel Interface, as well as the interface to SCSI drivers.

Section 3, "Configuration Tables," describes each software component's configuration table, which contains parameters that characterize the hardware and application environment.

Section 4, "Memory Usage," describes formulas used to calculate the amount of RAM required by each pSOSystem software component.

Appendix A, "Assembly Language Information," gives information that is useful to system programmers who understand 68K assembly language.

Related Documentation

When using pSOSystem you might want to have on hand the other two manuals included in the basic documentation set:

- *pSOSystem Getting Started* - explains how to create and bring up pSOSystem-based applications. This manual also contains a number of tutorials.
- *pSOSystem System Concepts* - provides theoretical information about the operation of pSOSystem.
- *pSOSystem System Calls* - describes the system calls and C language interface to pSOS+, pHILE+, pREPC+, pNA+, pRPC+, and pX11+.
- *pROBE+ User's Manual* - describes how to use the pROBE+ System Debugger/Analyzer.

Based on the options you have purchased, you might also need to reference one or more of the following manuals:

- *C++ Support Package User's Manual* - describes how to implement C++ applications in a pSOSystem environment.
- *OpEN User's Manual* - describes how to install and use pSOSystem's OPEN (Open Protocol Embedded Networking) product.
- *SNMP User's Manual* - describes the internal structure and operation of SNMP, Integrated System's Simple Network Management Protocol product. This manual also describes how to install and use the SNMP MIB (Management Information Base) Compiler.
- *pSOSim User's Manual* - describes how to install and use pSOSim, a UNIX-based pSOS+ kernel simulator.
- *XRAY+ User's Manual* - describes how to use the XRAY+ Source-Level Cross Debugger.

Notation Conventions

The following notation conventions are used in this manual:

- Function names (**q_receive**), filenames (**pdefs.h**), keywords (**int**), and operators (**!**) that must be typed exactly as shown are presented in bold.
- Italics indicate that a user-defined value or name (*drive*:*pathname*) can be substituted for the italicized word(s) shown. Italics also indicate emphasis, such as when important terms are introduced.
- Keynames [Enter] are shown within square brackets. Keynames separated by hyphens are typed together. For example, to type [Ctrl-Shift-E], hold down the [Ctrl] and [Shift] keys and type the letter E.
- `Code examples are shown in constant width.`

NAME

intro -- Introduction to Section 1: System Services

DESCRIPTION

This section describes the following pSOSystem system services:

- **Bootpc** Pseudo driver that requests BOOTSTRAP protocol information.
- **Bootpd** Implementation of the BOOTSTRAP protocol server.
- **FTP Client** Transfers files to and from a remote system.
- **FTP Server** Allows remote systems running FTP to transfer files to and from a pHILE+ device.
- **Loader** Allows run-time target loading and unloading of application programs.
- **MMU Library** Provides mapping tables for the Memory Management Unit.
- **NFS Server** Allows systems to share files in a networked environment.
- **pSH+** Interactive command line shell.
- **RARP** Reverse Address Resolution Protocol which can be used to identify a workstation's IP address, or obtained a dynamically assigned IP address from a domain name server (DNS).
- **Routed** Implementation of the Routing Information Protocol, or RIP.
- **Telnet Client** Supports communication with a remote system running a Telnet Server.
- **Telnet Server** Allows remote systems running the Telnet protocol to log into pSH+.
- **TFTP Server** Allows TFTP clients to read and write files interactively on pHILE+ managed disks.

NAME

bootpc -- BOOTP client

DESCRIPTION

With the **bootp** client feature, you can send a BOOTP request packet and get the necessary information for booting your target. As a minimum, this includes your IP address; it can also include the IP address of your router, the client's subnet mask, and the IP address of your domain name server (DNS).

NOTE: The bootpc is provided in the Network Utilities library as position dependent code.

BOOTP Client Code

The BOOTP client code uses User Datagram Protocol (UDP) and implements the following procedure:

```
get_bootp_params {
   long      (ni_entry)(),
   char      *bootp_file_name,
   char      *bootp_server_name,
   int       num_retries,
   int       flags,
   char      *ret_params
   };
```

ni_entry Network interface entry point. This parameter is set to the network interface entry procedure (for example, **NiLan**) in the **lan.c** file in the applicable board-support package.

bootp_file_name
The BOOTP filename is copied into the file field in the BOOTP request packet. It can be null, or its length can be up to 127 bytes.

bootp_server_name
The BOOTP server name is copied into the **sname** field in the BOOTP request packet. It can be null, or its length can be up to 63 bytes.

num_retries This parameter sets the number of retries for BOOTP requests. The retry interval is exponentially increased with the first retry interval of 1 second, the second retry interval of 2 seconds, and so on. If this parameter is set to 0, **get_bootp_params** uses BOOTP_RETRIES as the default value.

flags The BOOTP flags include the following:

RAW	Return a raw BOOTP reply packet.
COOK	Return extracted information from the BOOTP reply.
PSOSUP	Set this flag if the pSOS+ kernel is already running when **get_bootp_params** is called.

ret_params If the RAW flag is turned on, it should point to a data structure of type **bootppkt_t**, or it should point to a data structure of type **bootpparms_t**; both types are defined in the **bootpc.h** file. The result is copied into the area indicated by this parameter.

EXAMPLE of BOOTP Client Code

The following code segment provides an example of how to use the **get_bootp_params** procedure:

```
...

bootpparms_t bootp_parms;

extern long NiLan();

...

memset(&bootp_parms, 0, sizeof(bootpparms_t));

get_bootp_params(NiLan, 0, "ram.hex", 10, COOK, &bootp_parms);

...
```

In a **makefile**, you can add the **bootpc.lib** file in PSS_DRVOBJS, and include the following file:

```
$(PSS_ROOT)/drivers/bootpc/rules.mk
```

NAME

bootpd -- BOOTP daemon

DESCRIPTION

The **bootpd** server contained in pSOSystem's Networking Utilities product is an implementation of the BOOTSTRAP protocol server and is based on RFC951 and RFC1395.

NOTE: The bootpd is provided in the Network Utilities library as position dependent code.

It provides additional features by implementing: (a) a tag field (**ps**) in the **bootpd** configuration database, which identifies the forwarding server to which **bootpd** requests from a specific hardware device can be forwarded; (b) an optional default parent **bootpd** server address (**parentIP**) in the **bootpd** configuration table to which unresolved BOOTP requests can be forwarded.

bootpd creates a daemon task, BTPD, to handle BOOTP requests from clients. When BTPD starts, it reads configuration information from a user-supplied string, which it stores in its hash tables. When a BOOTP request comes in, if there is a match in the BTPD configuration database, BTPD first verifies whether the forwarding server field (**ps**) is set for the matching address. If it is set, the request is forwarded to the specified server regardless of other fields. Otherwise, BTPD processes the request and may send back a reply packet, if appropriate. If no match is found in BTPD's configuration database and a parent **bootpd** server is supplied when BTPD starts, BTPD forwards requests to its parent server.

The **bootpd** server in pSOSystem always ignores the server name field in BOOTP requests.

System/Resource Requirements

To use the **bootpd** server, you must have the following components installed:

- pSOS+ Real-Time Kernel.
- pNA+ TCP/IP Network Manager.
- pREPC+ Run-Time C Library.
- (Optional) pHILE+ File System Manager (not required for a **bootpd** server that only forwards requests).

In addition, **bootpd** requires the following system resources:

- Two Kbytes of user stack and two Kbytes of system stack.
- Two UDP sockets. One is used to receive BOOTP requests and send/forward BOOTP replies. The other is used to set an ARP cache entry in certain cases.

- The static memory requirement is four Kbytes. The dynamic memory size is affected by the server's database entries.

Starting the Routing Daemons

In order to use **bootpd** in an application, you need to link the pSOSystem network utilities library. **bootpd** is started with **bootpd_start(bootpdcfg_t*)**. The following code fragment gives an example of a database string and shows how to start **bootpd**:

```
#include <netutils.h>

char *bootp_table =
"scg.dummy:\
           sm=255.255.255.0:\
           td=3.0:\
           hd=/tftpboot:\
           bf=null:\
           dn=isi.com:\
           hn:\n\
subnetscg.dummy:\
           tc=scg.dummy:\
           gw=192.103.54.14:\
           ps=1.2.3.4:\n\
board1:\
           tc=subnetscg.dummy:\
           ht=ethernet:\
           ha=08003E20F810:\
           ip=192.103.54.229:\
           bf=ram.hex:\
           bs=123:\
           ps@:\
";

void start_bootpd_server()
{
static bootpdcfg_t bootpd_cfg;

     bootpd.priority = 200;
     bootpd_cfg.flags = BOOTPD_SYSLOG;
     bootpd_cfg.bootptab = bootp_table;
     bootpd_cfg.parentIP.s_addr = htonl(0xC0D8E61D);
```

```
    if (bootpd_start(&bootpd_cfg))
        printf("bootpd_start: failed to start\n");
}
```

The BOOTP Daemon Configuration Table and Database String

The **bootpd** server requires a user-supplied configuration table, defined as follows:

```
struct {
   unsigned long priority;       /* Priority of BTPD task */
   unsigned long flags;          /* Optional flags */
   char *bootptab;               /* The bootpd database string */
   struct in_addr parentIP;      /* Parent BOOTP server IP address */
   unsigned long reserved[2];    /* Reserved for future */
   } bootpdcfg_t;
typedef struct bootpdcfg_t;
```

priority This defines the priority at which the BTPD daemon task starts executing.

flags This specifies the following **bootpd** server options:

BOOTPD_SYSLOG This displays logging information on the pREPC+ standard error channel.

bootptab This is a pointer to a string that contains the **bootpd** configuration database. The string is defined as follows:

```
"hostname:\
tg=value:\
...\
tg=value:\n\
hostname:\
tg=value:\
...\
tg=value:"
```

where *hostname* is the actual name of a BOOTP client and *tg* is a two-character tag symbol. Most tags must be followed by an equals sign and a *value*, as above. Some may also appear in a boolean form with no value (i.e. *tg*:). For a list of currently recognized tags, see “Two-Character Tag Symbols,” on page 1-8.

parentIP This is the IP address in network byte order of this server's parent server, to whom this server can forward BOOTP requests.

Two-Character Tag Symbols

The following tags are currently recognized by the **bootpd** server:

bf	Bootfile
bs	Bootfile size in 512-octet blocks
cs	Cookie server address list
dn	Domain name
ds	Domain name server address list
gw	Gateway address list
ha	Host hardware address
hd	Bootfile home directory
hn	Send client's hostname to client
ht	Host hardware type (see Assigned Numbers RFC)
im	Impress server address list
ip	Host IP address
lg	Log server address list
lp	LPR server address list
ns	IEN-116 name server address list
rl	Resource location protocol server address list
rp	Root path to mount as root
sa	TFTP server address client should use
ps	BOOTP server address forwarding server should use
sm	Host subnet mask
sw	Swap server address
tc	Table continuation (points to similar "template" host entry)
td	TFTP root directory used by TFTP servers
to	Time offset in seconds from UTC (Universal Time Coordinate)
ts	Time server address list
vm	Vendor magic cookie selector

There is also a generic tag, **T** *n*, where *n* is an RFC1084 vendor field tag number. Thus, it is possible to immediately take advantage of future extensions to RFC1084 without being forced to modify **bootpd** first. Generic data may be represented as either a stream of

hexadecimal numbers or as a quoted string of ASCII characters. The length of the generic data is automatically determined and inserted into the proper field(s) of the RFC1084-style BOOTP reply.

The following tags take a whitespace-separated list of IP addresses: **cs**, **ds**, **gw**, **im**, **lg**, **lp**, **ns**, **rl**, and **ts**. The **ip**, **sa**, **ps**, **sw**, and **sm** tags each take a single IP address. All IP addresses are specified in standard Internet "dot" notation and may use decimal, octal, or hexadecimal numbers (octal numbers begin with 0, hexadecimal numbers begin with '0x' or '0X').

The **ht** tag specifies the hardware type code as either an unsigned decimal, octal, or hexadecimal integer, or as one of the following symbolic names: **ethernet** or **ether** for 10Mb Ethernet, **ethernet3** or **ether3** for 3Mb experimental Ethernet, **ieee802**, **tr**, or **token-ring** for IEEE 802 networks, **pronet** for Proteon ProNET Token Ring, or **chaos**, **arcnet**, or **ax.25** for Chaos, ARCNET, and AX.25 Amateur Radio networks, respectively. The **ha** tag takes a hardware address, which must be specified in hexadecimal; optional periods and/or a leading '0x' may be included for readability. The **ha** tag must be preceded by the **ht** tag (either explicitly or implicitly; see **tc** below).

The **td** tag is used to inform **bootpd** of the root directory used by **tftpd**. The **hd** tag is actually relative to the root directory specified by the **td** tag. For pHILE+ files, the **td** tag should always be there to include the volume name. For Sun files, the **td** tag is optional. For example, if your BOOTP client bootfile is **/tftpboot/bootimage** on volume 3 in your system, then specify the following in the **bootptab** string:

:td=3.0:hd=/tftpboot:bf=bootimage:

The hostname, home directory, and bootfile are ASCII strings that may be optionally surrounded by double quotes ("). The client's request and the values of the **hd** and **bf** symbols determine how the server fills in the **bootfile** field of the BOOTP reply packet.

If the client specifies an absolute pathname (an absolute pathname in pHILE+ begins with a volume name followed by a complete path) and that file exists on the server machine, that pathname is returned in the reply packet. If the file cannot be found, the request is discarded; no reply is sent. If the client specifies a relative pathname, a full pathname is formed by prepending the value of the **hd** tag and testing for existence of the file. If the **hd** tag is not supplied in the configuration file or if the resulting boot file cannot be found, then the request is discarded.

Clients that specify null boot files always elicit a reply from the server. The exact reply depends again upon the **hd** and **bf** tags. If the **bf** tag gives an absolute pathname and the file exists, that pathname is returned in the reply packet. Otherwise, if the **hd** and **bf** tags together specify an accessible file, that filename is returned in the reply. If a complete filename cannot be determined or the file does not exist, the reply will contain a zeroed-out **bootfile** field.

In all these cases, existence of the file means that, in addition to actually being present, the file must have read access to public, since this is required by **tftpd** to permit the file

transfer. Also, all filenames are first tried as *filename.hostname* and them simply as *filename*, thus providing for individual per-host bootfiles.

The **sa** tag may be used to specify the IP address of the particular TFTP server you wish the client to use. In the absence of this tag, **bootpd** tells the client to perform TFTP to the same machine bootpd is running on.

The **ps** tag may be used to specify the IP address of a peer BOOTP server address to which the BOOTP request will forward.

The time offset **to** may be either a signed decimal integer specifying the client's time zone offset in seconds from UTC, or the keyword **auto**, which sets the time zone offset to 0. Specifying the **to** symbol as a boolean has the same effect as specifying **auto** as its value.

The bootfile size **bs** may be either a decimal, octal, or hexadecimal integer specifying the size of the bootfile in 512-octet blocks, or the keyword **auto**, which causes the server to automatically calculate the bootfile size at each request. As with the time offset, specifying the **bs** symbol as a boolean has the same effect as specifying **auto** as its value.

The vendor magic cookie selector (the **vm** tag) may take one of the following keywords: **auto** (indicating that vendor information is determined by the client's request), **rfc1048** or **rfc1084** (which always forces an RFC1084-style reply).

The **hn** tag is strictly a boolean tag; it does not take the usual equals-sign and value. It's presence indicates that the hostname should be sent to RFC1084 clients. **bootpd** attempts to send the entire hostname as it is specified in the configuration file; if this will not fit into the reply packet, the name is shortened to just the host field (up to the first period, if present) and then tried. In no case is an arbitrarily-truncated hostname sent (if nothing reasonable will fit, nothing is sent).

Often, many host entries share common values for certain tags (such as name servers, etc.). Rather than repeatedly specifying these tags, a full specification can be listed for one host entry and shared by others via the **tc** (table continuation) mechanism. Often, the template entry is a dummy host that does not actually exist and never sends BOOTP requests. Note that **bootpd** allows the **tc** tag symbol to appear anywhere in the host entry. Information explicitly specified for a host always overrides information implied by a **tc** tag symbol, regardless of its location within the entry. The value of the **tc** tag may be the hostname or IP address of any host entry previously listed in the configuration file.

Sometimes it is necessary to delete a specific tag after it has been inferred via **tc**. This can be done using the construction tag @ which removes the effect of tag. For example, to completely undo an IEN-116 name server specification, use "**:ns@:**" at an appropriate place in the configuration entry. After removal with @, a tag is eligible to be set again through the **tc** mechanism.

Host entries are separated from one another by newlines in the configuration string; a single host entry may be extended over multiple lines if the lines end with a backslash (\). It is also acceptable for lines to be longer than 80 characters. Tags may appear in any order, with the following exceptions: the hostname must be the very first field in an entry, and the hardware type must precede the hardware address.

NAME

FTP Client -- Transfer files to and from a remote system

DESCRIPTION

The FTP (File Transfer Protocol) Client contained in the Network Utilities product, transfers files to and from a remote system. The remote system must run an FTP server program that conforms to the ARPANET File Transfer Protocol. The FTP Client runs as an application under pSH+ and is invoked with the following command:

pSH+ > ftp [*remote_system*]

where ***remote_system*** is a remote system IP address.

If no arguments are given, FTP Client enters *command mode* (indicated by the `ftp>` prompt). In command mode, FTP accepts and executes commands described under “FTP Commands” on page 1-12.

If the command contains arguments, FTP executes an **open** command with those arguments. See “FTP Commands” on page 1-12 for a description of **open** and the other FTP commands.

The normal abort sequence, [CTRL]-C does not work during a transfer.

NOTE: The ftp client is provided in the Network Utilities library as position dependent code.

Configuration and Startup

The FTP Client requires the following:

- pSOS+ or pSOS+m Real-Time Kernel
- pREPC+ Run-Time C Library
- pHILE+ File System Manager
- pSH+ interactive shell command
- Four Kbytes of user stack space
- Four Kbytes of supervisor stack space

pSH+ starts FTP Client by calling **ftp_main()**. pSOSystem includes a pre-configured version of pSH+ and FTP Client, but to add FTP Client to pSH+, an entry for it must be made in the pSH+ list of user applications. The following shows an example of a user application list containing FTP and Telnet:

```
struct appdata_t appdata[] = {
    {"ftp", "file transfer application", ftp_main, "ft00", 250,
      4096, 4096,1, 0},
    {"telnet", "telnet application", telnet_main, "tn00", 250,
      4096, 4096, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0}
};
```

You can define the other elements in the preceding example ("**ft00**", and so on).

FTP Commands

The following commands can be entered at the FTP prompt (`ftp>`). File naming conventions and descriptions of transfer parameters follow these command descriptions.

! [*command*] Run **command** as a shell command on the local machine.

account [*passwd*] Provide a supplemental password required by a remote system for access to resources after a successful login. If no argument is included, you are prompted for an account password in a non-echoing input mode.

append local_file [*remote_file*]
Append a local file to a file on the remote machine. If ***remote_file*** is unspecified, the local filename is used to name the remote file. File transfer uses the current settings for representation type, file structure, and transfer mode.

ascii Set the representation type to network ASCII (the default type).

bell Sound a bell after each file transfer command completes.

binary Set the representation type to image.

bye Terminate the FTP session to the remote server and exit FTP. An EOF also terminates the session and causes an exit.

cd *remote_directory* Change the working directory on the remote machine to ***remote_directory***.

cdup Change the working directory on the remote machine to the parent of the current working directory on the remote machine.

close Terminate the FTP session with the remote server and return to the command interpreter.

cr Toggle [RETURN] stripping during network ASCII-type file retrieval. Records are denoted by a [RETURN] or [LINEFEED] sequence during a network ASCII-type file transfer. When **cr** is on (the default), [RETURN] characters are stripped from this sequence to conform to the UNIX system single-LINEFEED record delimiter. Records on non-UNIX system remote hosts may contain

single [LINEFEED] characters; when a network ASCII-type transfer is made, the [LINEFEED] characters can be distinguished from a record delimiter only when **cr** is off.

delete *remote_file* Delete the file ***remote_file*** on the remote machine.

dir [***remote_directory***] [***local_file***]
Print a listing of the directory contents in the directory, the *remote directory*, and, optionally, the local file. If no directory is specified, the current working directory on the remote machine is used. If no local file is specified or if the local file is specified by a dash (-), output goes to the terminal.

disconnect Synonymous to **close**.

get remote_file [***local_file***]
Retrieve the remote file and store it on the local machine. If the local filename is not specified, it receives the same name it has on the remote machine. When no name is specified, the program-generated name can be altered because of the current **case**, **ntrans**, and **nmap** settings. The current settings for representation type, file structure, and transfer mode apply during file transfers.

glob Toggle globbing (filename expansion) for **mdelete**, **mget** and **mput**. If globbing is off, filenames are taken literally.

Globbing for **mput** is done the same as with the **csh** UNIX command. For **mdelete** and **mget**, each remote filename is expanded separately on the remote machine, and the lists are not merged.

Expansion of a directory name is likely to be very different from expansion of the name of an ordinary file: the exact result depends on the remote operating system and FTP server. The result can be previewed by executing the following:

mls ***remote_files*** -

The **mget** and **mput** commands are not meant to transfer entire directory subtrees of files: instead, transfer directory subtrees of files by transferring a **tar** (UNIX command) archive of the subtree (using the image representation type as set by the **binary** command).

hash Toggle hash-sign (#) printing for each data block transferred.

help [***command***] Print information about the command. With no argument, **ftp** lists the known commands.

lcd [***directory***] Change the working directory on the local machine. If no directory is specified, the user's home directory is used.

ls [***remote_directory***] [***local_file***]
Print a listing of the contents of a directory on the remote machine. If ***remote_directory*** is unspecified, the current working directory is used. If no local file is specified or if ***local_file*** is a dash (-), the output goes to the terminal.

mdelete [***remote_files***]
Delete the specified ***remote_files*** on the remote machine.

mdir ***remote_files local_file***
The **mdir** command is like **dir**, except that **mdir** supports specification of multiple remote files. If interactive prompting is on, **ftp** prompts you to verify that the last argument is the local file targeted to receive **mdir** output.

mget ***remote_files*** Expand the ***remote_files*** on the remote machine and execute a **get** for each filename thus produced. See **glob** for details the filename expansion. Resulting filenames are then processed according to **case**, **ntrans**, and **nmap** settings. Files are transferred into the local working directory, which can be changed by executing **lcd** ***directory***. New local directories can be created with **!mkdir** ***directory***.

mkdir [***directory_name***]
Make a directory on the remote machine.

mls ***remote_files local_file***
The **mls** command resembles **ls**(1V), except that **mls** supports specification of multiple remote files. If interactive prompting is on, **ftp** prompts you to verify that the last argument is the local file targeted to receive **mls** output.

mode [***mode_name***] Set the transfer mode to ***mode_name***. The only valid mode name is **stream**, which corresponds to the default stream mode.

mput ***local_files*** Expand wild cards in the list of local files given as arguments and do a **put** for each file in the resulting list. See **glob** for details on filename expansion.

nlist [***remote_directory***] [***local_file***]
Print an abbreviated listing of the contents of a directory on the remote machine. If ***remote_directory*** is unspecified, the current working directory is used. If no local file is specified or if ***local_file*** is a dash (-), the output goes to the terminal.

open host [*port*] Establish a connection to the specified host FTP server. A port number is optional. If ***port*** is specified, **ftp** attempts to contact an FTP server at that ***port***. If the **auto-login** option is on (the default), **ftp** also attempts to automatically log the user into the FTP server (refer to the description of **user**).

prompt Toggle interactive prompting. Interactive prompting during multiple file transfers allows you to selectively retrieve or store files. Prompting is on by default. If prompting is off, an **mget** or **mput** transfers all files, and an **mdelete** deletes all files.

put *local_file* [*remote_file*]
Store a local file on the remote machine. If ***remote_file*** is unspecified, the local filename is used to specify the remote file. File transfer uses the current settings for representation type, file structure, and transfer mode.

pwd Print the name of the current working directory on the remote machine.

quit Synonymous to **bye**.

quote *arg1 arg2...* Send the arguments specified verbatim to the remote FTP server. A single FTP reply code is expected.

recv *remote_file* [*local_file*]
Synonymous to **get**.

remotehelp [*command_name*]
Request help from the remote FTP server. If a ***command_name*** is specified, it also goes to the server.

rename *from to* Rename the file specified by ***from*** on the remote machine to have the name specified by ***to***.

reset Clear reply queue. This command synchronizes command/reply sequencing with the remote FTP server. Synchronization may be necessary if the remote server violates FTP protocol.

rmdir *directory_name* Delete a directory on the remote machine.

runique Toggle storing of files on the local system with unique filenames. The generated unique filename is reported. The **runique** command does not affect local files generated from a shell command. By default **runique** is OFF.

If a file already exists with the same name as the target local filename for a **get** or **mget**, a.1 is appended to the name. If the resulting name matches another existing filename, a **.2** is appended to the original name. If the additions reach **.99**, an error message is printed, and the transfer does not take place.

send local_file [*remote_file*]
Synonymous to **put**.

sendport Toggle the use of PORT commands. By default, **ftp** attempts to use a PORT command when it establishes a connection for each data transfer. The use of PORT commands can prevent delays during multiple file transfers. If the PORT command fails, **ftp** uses the default data port. When the use of PORT commands is disabled, no attempt is made to use PORT commands for each data transfer. This is useful for certain FTP implementations that ignore PORT commands but incorrectly indicate they have been accepted.

status Show the current status of FTP.

sunique A toggle for storing of files on a remote machine under unique filenames. For successful file storage, the remote FTP server must support the STOU command. The remote server reports the unique name. The default state is OFF.

tenex Set the representation type to the value needed for communication with TENEX machines.

type [*type_name*] Set the representation type to ***type_name***. Valid type names are as follows:

ascii	For network ASCII.
binary or **image**	For image.
tenex	For local byte size of eight bits (used to talk to TENEX machines).

If no type is specified, the current type is printed. The default type is network ASCII.

user username [*password*] [*account*]
Identify the user to the remote FTP server. If the password is not specified and the server requires it, **ftp** prompts for the password after it disables local echo. If an account field is unspecified and the FTP server requires one, the user prompts for an account field.

If the remote server does not require an account input for login and if it is nevertheless specified, an account command is relayed to the remote server after the login sequence is completed. Unless **ftp** is invoked with **auto-login** disabled, this process is done automatically upon initial connection to the FTP server.

verbose Toggle verbose mode. In verbose mode, all responses from the FTP server are displayed to the user. If verbose mode is on, statistics about the efficiency of the transfer are reported when a file transfer completes. By default, verbose mode is on if FTP commands come from a terminal (and off otherwise).

? [***command***] Synonymous to **help**.

A command argument can have embedded spaces if the argument is enclosed in quote marks (").

If a required command argument is absent, **ftp** prompts for that argument.

File Naming Conventions for FTP Command Arguments

Arguments for some commands in the preceding list can be local files. Local files specified as arguments to FTP commands are processed according to the following rules:

- If the specified filename is a dash (-), the standard input (for reading) or standard output (for writing) is used.
- If the filename is not a dash and if globbing is enabled, local filenames are expanded according to the rules used in the **csh** UNIX command. (See also the **glob** command.) If the FTP command expects a single local file (for example, with a **put** command), only the first filename generated by the globbing operation is used.
- For **mget** and **get** commands that have unspecified local filenames, the local filename is the same as the remote filename. The resulting filename can then be altered if **runique** is on.
- For **mput** and **put** commands with unspecified remote filenames, the remote filename is the local filename. The resulting filename can then be altered by the remote server if **sunique** is on.

File Transfer Parameters

FTP command specification (described in the preceding pages) includes three parameters that can affect a file transfer. The three parameters are the *representation type*, the *file structure*, and the *transfer mode*. The representation type can be one of the following:

- Network ASCII
- EBCDIC
- Image

The network ASCII and EBCDIC types also have a subtype. This subtype specifies whether vertical format control ([NEWLINE] characters, form feeds, and so on) are to be processed in one of the following ways:

- Passed through (*nonprint*)
- Provided in Telnet format (*TELNET format controls*)

FTP supports the network ASCII (subtype non-print only) and image types.

Next, the file structure can be one of **file** (no record structure), **record**, or **page**. FTP supports only **file**.

Lastly, the transfer mode can be either **stream**, **block**, or **compressed**. FTP supports only **stream**.

FTP Client Bugs

Correct execution of many commands depends on correct operation by the remote server. An error in the treatment of carriage returns in the 4.2 BSD code handling transfers with a representation type of network ASCII has been corrected. This correction can result in incorrect transfers of binary files to and from 4.2 BSD servers using a representation type of network ASCII. Avoid this problem by using the image type.

NAME

FTP Server -- Allow remote systems running FTP to transfer files to/from a pHILE+ device

DESCRIPTION

FTP Server allows remote systems that are running the ARPANET File Transfer Protocol to transfer files to and from a pHILE+ device. FTP Server is implemented as a daemon task named **ftpd**. The **ftpd** daemon listens for connection requests from clients and creates server tasks for each FTP session that a client establishes.

NOTE: The ftpd task is provided in the Network Utilities library as position dependent code.

Configuration and Startup

FTP Server requires the following:

- pSOS+ Real-Time Kernel.
- pHILE+ File System Manager.
- pNA+ TCP/IP Network Manager.
- pREPC+ Run-Time C Library.
- Eight Kbytes of user stack and two Kbytes of supervisor stack per session.
- One TCP socket, which is used to listen for client session requests, and two additional TCP sockets per session.
- Eight Kbytes of dynamic storage, which a pREPC+ **malloc()** system call allocates
- A user-supplied configuration table.

The user-supplied FTP Server Configuration Table defines application-specific parameters, and the following is a template for this table. This template exists in the **include/netutils.h** file.

```
struct ftpcfg_t {
        long task_prio;          /* priority for ftpd task */
        long max_sessions;       /* max # of concurrent sessions */
        char *vol_name;          /* name of the login volume */
        char **hlist;            /* list of trusted clients */
        struct ulist_t *ulist;   /* list of permitted users */
        long reserved[2];        /* must be 0 */
        };
```

Definitions for the FTP Server Configuration Table entries are as follows:

task_prio Defines the priority at which the daemon task **ftpd** starts executing.

max_sessions Defines the maximum number of concurrently open sessions.

vol_name Defines the name of the volume to use when a client logs into pSOSystem.

hlist Contains a pointer to a list of the IP addresses of the trusted clients. If this field is 0, FTP Server accepts a connection from any client.

ulist Points to a list of structures that contain login information of permitted users. If this field is 0, all users are allowed to log in. The following is a template for one of these structures:

```
struct ulist_t {
    char *login_name;   /* user name */
    char *login_passwd; /* user password */
    long reserved[4];   /* must be 0 */
    };
```

The following is an example structure with three entries:

```
struct ulist_t ulist[] {
    {"guest", "psos0", 0, 0, 0, 0},
    {"scg",   "andy0", 0, 0, 0, 0},
    {0,       0,       0, 0, 0, 0}
    }
```

reserved Reserved for future use, and each must be 0.

FTP Server comes as one object module and must be linked with a user application. Calling the function **ftpd_start(ftpdcfg)** at any time after pSOSystem initialization (when ROOT is called) starts it. The parameter **ftpdcfg** is a pointer to the FTP Server Configuration Table. If FTP Server starts successfully, **ftpd_start()** returns 0, otherwise it returns a non-zero value.

EXAMPLE

The following code fragment shows an example configuration table and the call that starts FTP Server. The complete example code exists in the **apps/netutils/root.c** file.

```
#include <ftpdcfg.h>
start_ftp_server() {
        /* FTP server configuration table */
        static ftpcfg_t ftpcfg =
        {
        250,            /* Priority for ftpd task */
        4,              /* Maximum number of concurrent sessions */
        "4.0",          /* Name of the login volume */
        0,              /* List of trusted clients */
        0,              /* List of permitted users */
        0, 0            /* Must be 0 */
        };
        /* start the FTP server */
        if (ftpd_start(&ftpcfg))
        printf("ftpd_start: failed to start\n");
        }
```

NAME

Loader -- Allow run-time target loading and unloading of application programs

DESCRIPTION

The pSOSystem loader included in the pSOSystem base package, provides a programmatic interface for controlling run-time target loading and unloading of application programs from a variety of I/O interfaces. The loader is supplied as a library of functions that can be called from a user application.

Powerful loader applications can be written using just three functions (**load**, **unload**, and **release**). The loader library has been designed to depend only on pSOS+ system services; it does not depend on any other components. However, you may need to include other components like pHILE+ and pNA+, depending on the type of I/O interface being used to load applications.

The loader supports the loading of object files residing on pHILE+ media (pHILE+ volumes, CD_ROM volumes, MS-DOS volumes, or remote file systems mounted through NFS). The loader also supports loading from any device driver that conforms to the interface standard defined by pREPC+. Additional requirements for device drivers are described in “Guidelines for Writing Device Drivers” on page 1-37. A pseudo device driver that uses TFTP (Trivial File Transfer Protocol) to transfer files from a remote host is also provided with pSOSystem. You can use the TFTP device driver in conjunction with the loader.

The loader can load object files that are either in Motorola S-record (SREC) format or in the IEEE-695 format generated by MRI compiler tools. The following types of object files are supported by the loader:

- SREC object files containing absolute (position-dependent) code.
- SREC object files containing position-independent code.
- IEEE-695 object files containing absolute code.
- IEEE-695 object files containing relocatable code.

The term *relocatable* refers to object files that contain relocation information. Such object files are produced as intermediate files during the compilation/linking process. The relocatable object (**.o**) files are produced by various MRI tools, as follows:

- All object files produced by the assembler are relocatable.
- Object files produced by the C compiler with the -**c** option specified are relocatable.
- Object files produced by the MRI linker with the incremental linking (-**i**) option specified are relocatable.

The ability to load relocatable files with the loader provides extra flexibility. For example, you can generate position-dependent code but defer the decision of where to place the code in target memory until runtime.

NOTE: Relocatable files must not contain unresolved external symbol references.

Loader Configuration

The following files are associated with the loader:

sys/libc/loader.lib	MRI library file containing the loader.
include/loader.h	Header file that contains typedefs, defines, and function prototypes for the functions provided in the loader library.
configs/std/ldcfg.c	Configuration file for customizing the loader. It controls what modules get linked with the user loader application.
apps/loader/README	Contains detailed on-line instructions for generating and running a sample loader application.
apps/loader/Makefile	Contains the rules to build a sample loader application and is used by the UNIX **make** utility.
apps/loader/sys_conf.h	The pSOSystem configuration file.
apps/loader/*.[csh]	Source programs for the sample application demonstrating how to use loader functions in applications. To run this sample application, you may need to configure pREPC+, pHILE+, pRPC+, pNA+, and/or the TFTP pseudo device driver in the system.
apps/loader/loadable/*	Source files for a simple pSOSystem application that is intended to be loaded by the sample loader application.
configs/std/configpi.mk	System makefile for generating position-independent loadable applications.
configs/std/configre.mk	System makefile for generating relocatable loadable applications.
configs/std/beginapi.s	Application startup file for use by position- independent loadable applications (similar to the **begina.s** file provided with pSOSystem).

The loader contains two user-configurable modules. One supports the loading of Motorola S-records and the other supports the loading of MRI IEEE-695 object files. It is possible to generate a loader application that contains any one or both of the modules. By default, both of the modules are enabled. The **apps/loader/sys_conf.h** file contains the following two **#define** statements:

```
#define LD_SREC_MODULE YES    /* Motorola S-record support */
#define LD_IEEE_MODULE YES    /* MRI IEEE-695 support */
```

To exclude a particular module from getting linked to the loader application, change **YES** to **NO** for the module you want to exclude.

sys_conf.h also contains the following **#define**, which determines the maximum number of loading operations that can be handled simultaneously by the code in the loader library:

```
#define LD_MAX_LOAD 8    /* Max number of simultaneously active load */
```

You must make any necessary changes to **LD_SREC_MODULE**, **LD_IEEE_MODULE**, or **LD_MAX_LOAD** by modifying **sys_conf.h** before generating the loader application.

In addition to the files listed above, a host-executable utility called **ld_prep** is present under the various **bin**/<**host**> subdirectories in the pSOSystem directory tree. You must include the proper subdirectory in your **PATH**, depending on the host environment you are using for pSOSystem application development.

For example, if using a Sun SPARCstation as the development platform, modify your path as follows:

```
set path = ($path $PSS_ROOT/bin/sparc) # if csh is the working shell
```

or

```
PATH=$PATH:$PSS_ROOT/bin/sparc # if sh or ksh is the working shell
```

where **PSS_ROOT** is an environment variable specifying the pathname of the pSOSystem root directory.

Copy the files under the **apps/loader** directory to a working directory of your choice before making any modifications or generating the sample application. (The UNIX **cp -r** or MS-DOS **xcopy** commands can be used for this purpose.) The **README** file contains detailed information about the sample application. It also contains instructions for generating and running this application. You must follow these instructions to compile and run the sample loader application. Run this application and view the sample code to help familiarize yourself with the loader.

Concepts and Operation

The loader is useful in situations where you are dealing with multiple applications (running simultaneously on a target), and they can be partitioned so that no two applications share symbol references with each other. There can be many reasons for partitioning applications into multiple executable files and/or using dynamic loading. Some of these are as follows:

- All of the applications, taken together, are too big to fit in target memory. In certain cases, you may want to load/unload the applications on an as-needed basis.
- Depending on the hardware configuration, you may want to configure and load certain applications at runtime.
- In the development environment, you may want to load a new version of an "already-running" application without bringing down the whole system.

- In a situation where it is difficult or impossible to determine the final load address of an application, you may want to delay this decision until runtime.

Typically, you will make the loader run as part of the root task. This task remains resident on the target and loads other applications as and when needed. This is one of the suggested approaches and, as demonstrated by the sample application, the loader functions can be used in many other ways.

For ease of explanation, assume the presence of a single task called the *loader task*, which takes care of loading other application tasks, called *loadable applications*. The loader task is linked with pSOSystem and gets loaded on the target system using the standard method for bootstrapping the system. First, an outline is provided of a simple method for writing the loader task using the functions provided by the loader library, **loader.lib**.

The **load()** function is provided for loading loadable applications. These applications may remain resident on the target forever, or only temporarily. Loadable applications can be unloaded using the **unload()** function. A call to **unload()** frees up any memory allocated by **load()** for the run-time image of a loadable application; it also frees up any state information associated with the loadable application [saved by the loader library during the call to **load()**].

If the loadable application is to remain resident on the target forever, the **release()** function must be called to free up any state information associated with the loadable application. A call to **release()** does not free up the memory occupied by the run-time image of the loadable application, and the application can keep running without any hindrance. A detailed description of these functions is provided later in this section.

You open the file containing the loadable application using either the pHILE+ **open_f()** call or the **de_open()** function, which opens the device driver through which the loadable application will be read. The file descriptor returned by **open_f()** or the device number of the device driver must be passed to the **load()** function as the first argument (**fd**). You must also specify whether the first argument refers to a pHILE+ file descriptor or a device number. This is done by setting either **LD_DESC_PHILE** or **LD_DESC_DEV** in the second argument (**flags**) passed to **load()**.

The **load()** function reads in the loadable application using either the **read_f()** function of pHILE+ or the **de_read()** function, whatever the case may be. It determines the object file format of the loadable application and invokes the appropriate module (SREC or IEEE) to convert the object file into a binary image suitable for execution. The exact behavior of **load()** depends on the type of code (*position independent*, *absolute*, or *relocatable*), as follows:

- **Loading Absolute Code**
  Any object files containing absolute code (that is, position-dependent and non-relocatable code) are loaded at the address specified at the time of linking. The **load()** function does not allocate any memory for loading the run-time image. It is the responsibility of the calling task to make sure that it is safe to load the

application at the address to which it was linked. You cannot override the default addresses, as it does not make sense to load absolute code at a location to which it was not linked.

- **Loading Position-Independent Code**
  When loading object files containing position-independent code, the load addresses that were specified during the time of linking are ignored. **load()** allocates the memory needed to load the binary image from pSOS+ Region 0 (**RN#0**). It is possible to override the addresses selected by **load()**.

- **Loading Relocatable Code**
  Object files containing relocatable code are also treated like files containing position-independent code. By default, the needed memory is allocated from Region 0, and it is possible to override the default load addresses. Internally, the **load()** function does the necessary processing to relocate an otherwise position-dependent code using the relocation information present in the object file.

Loading absolute code is a one-step process. Similarly, loading position-independent or relocatable code at the default load address chosen by the loader is also a one-step process. You simply call **load()** with the **LD_LOAD_DEF** flag set in the **flags** argument.

If for some reason you want to control where the various parts (sections) of an object file get loaded into target memory, a two-step process must be followed:

1. Call **load()** with the **LD_GET_INFO** flag set in the **flags** argument. The **load()** function reads in the object file information from the header present therein and returns a pointer to this information in the third argument (**of_info**) passed to **load()**. You can modify the load addresses (part of the information returned through **of_info**) of one or more of the sections.

2. Call **load()** again with the **LD_LOAD_MOD** flag set in the **flags** argument and with the modified object file information (pointed to by ***of_info**) passed as the third argument.

Once the **load()** call returns, you are free to close the object file (or device driver) by calling **close_f()** (or **de_close()**). At this point the binary image of the loadable application has been loaded into memory. If the binary image corresponds to a pSOS+ task, you can create and start the task at any time. In most cases, the entry point to the task can be obtained from the object file information returned by **load()**. If the entry point is not known, it is set to zero by **load()**. You must call the **t_create()** and **t_start()** system services of pSOS+ with appropriate arguments to create and start the task, respectively.

Once the task is running there are two possibilities, as explained earlier:

- You want the task to keep running and you never intend to stop it and unload it from the memory. In this case, the **release()** function must be called with the object file information returned by **load()** as the only argument. Once **release()** is called, you must not reference the object file information.

- The other possibility is that after the loaded task completes its job, you may want to delete the task and free up the memory it was using. In such cases, you must call **unload()**, with the only argument to **unload()** being the object file information returned by the earlier call to **load()**. Once **unload()** is called, you must not reference the object file information. It is your responsibility to delete the task being unloaded in a graceful manner, so that it unlocks any locked resources and frees up any allocated resources before it gets deleted and unloaded. The sample loadable applications provide examples for your reference.

The Loader API

Following is a template for the three data structures used by the pSOSystem loader. The first is the **OF_INFO** structure, a pointer to which is returned by **load()** and also gets passed to **unload()** and **release()**. The second is the **SECN_INFO** structure, which is contained in the **OF_INFO** structure. The third is the **TASK_INFO** structure, which is also contained in the **OF_INFO** structure. These structures are defined in the **<include/loader.h>** file.

```
typedef struct OF_INFO{
      int desc;              /* Object file descriptor */
      char format[5];        /* Object file format */
      char code_type;        /* Code Type (Absolute/Relocatable) */
      char filler[2];        /* Reserved, do not use */
      int nsecns;            /* Number of sections */
      SECN_INFO *secn_info;  /* Section information */
      TASK_INFO task_info;   /* Info needed to create & start task */
      } OF_INFO;
```

desc Used by the loader to identify the loaded object file. This is a read-only element. You must not modify it.

format Four-character null-terminated string that identifies the object file format of the file loaded by the loader. The values returned in this field are **SREC** or **IEEE**, which correspond to Motorola S-record or MRI IEEE-695 formats, respectively. This is a read-only element.

code_type Can take one of two values: **LD_ABSOLUTE** or **LD_RELOCATABLE**. **LD_ABSOLUTE** implies that the code is position dependent, and **LD_RELOCATABLE** implies that the code is either position independent or the object file contains relocation information and can be loaded anywhere in target memory. This is a read-only element.

nsecns Tells the number of independently loadable sections of the object file. A file in **SREC** format always has one section. This is a read-only element.

secn_info Points to an array of **SECN_INFO** structures that has **nsecns** elements. As described below, the **SECN_INFO** structure contains information regarding each of the separately loadable sections of the object file.

task_info Structure of type **TASK_INFO**. The information contained herein may be used by the loader to create and start the task, once the loader has loaded the object file into target memory.

The second structure, **SECN_INFO**, contains information regarding the individually loadable sections of an object file and is defined as follows:

```
typedef struct secn_info{
        char name[LD_SECNAMELEN];   /* Name of the section */
        unsigned long type;         /* Type of the section */
        unsigned long size;         /* Size of the section */
        unsigned long base;         /* Section load address */
        } SECN_INFO;
```

name Describes the name of the section. This field is compiler dependent and is supplied for your information. The loader does not make use of this field. This is a read-only element.

type Describes the type of the section. This field is This field is compiler dependent and is supplied for your information. The loader does not make use of this field. This is a read-only element.

size Specifies the size of the section in bytes. This is a read-only element.

base Specifies the address in memory where the section will be loaded. You can modify **base**, as explained in the description of the **load()** function later in this section, to control the placement of the section in memory if the **code_type** is **LD_RELOCATABLE**.

Modifications done to any other fields of this structure in between any two loader calls are ignored by the loader.

The third structure, **TASK_INFO**, contains information necessary to create and start a task. This structure is not used by the loader but is intended to be used by the application to create the task and start it after it has been loaded using the pSOS+ **t_create()** and **t_start()** system services. This information is obtained from the object file by **load()** and can be stored in the object file by running **ld_prep** on the object file and specifying the appropriate values for various task-specific parameters (see the man page for **ld_prep** for further details). **TASK_INFO** is defined as follows:

```
typedef struct task_info{
        char name[4];               /* Name of the task to be created */
        unsigned long priority;     /* Task priority */
        unsigned long sstack_sz;    /* Supervisor stack size */
        unsigned long ustack_sz;    /* User stack size */
        unsigned long create_flags;/* Flags used by t_create() */
        unsigned long start_mode;   /* Mode used by t_start() */
```

```
        void   (*entry) ();          /* Task entry point */
        } TASK_INFO;
```

name Four-character name of the task passed to **t_create()**.

priority Starting priority of the task passed to **t_create()**.

sstack_sz Size of the supervisor stack (in bytes) passed to **t_create()**.

ustack_sz Size of the user stack (in bytes) passed to **t_create()**.

create_flags Flags passed to **t_create()**.

start_mode Mode passed to **t_start()**.

entry Entry point, if any, for the task passed to **t_start()**.

All elements in the **TASK_INFO** structure are read-only.

The load() Function

load() is defined as follows:

```
#include <loader.h>
unsigned long load (
        unsigned long      fd,
        unsigned long      flags,
        OF_INFO            **of_info
        );
```

load() reads an object file from an open file descriptor **fd** and converts the incoming stream of data into a binary image ready for execution. The information about the object file is read and a pointer to it is returned in the location pointed by **of_info**. The **fd** is either a file descriptor returned by a call to the pHILE+ **open_f()** routine or it is the device number of a pREPC+-compatible device driver. You must set the **LD_DESC_PHILE** or **LD_DESC_DEV** fields in the **flags** argument to specify whether **fd** is a file descriptor returned by **open_f()** or a device number.

The exact behavior of **load()** is controlled by the load type specified by the **flags** argument. You can specify one of three load types (**LD_GET_INFO**, **LD_LOAD_DEF**, **LD_LOAD_MOD**) by bitwise OR-ing one of the three values in the **flags** argument.

If **LD_LOAD_DEF** is specified, **load()** reads the object file and loads the binary image into target memory, using the default load address specified by the object file header. The values used to load the file are stored in an **OF_INFO** structure, and a pointer to this structure is returned through **of_info**.

If **LD_GET_INFO** is specified, **load()** reads the object file header information and returns a pointer to it through **of_info**. No binary image of the object file is loaded in target memory.

You can modify certain values returned in the **OF_INFO** structure and call **load()** to load the binary image by specifying the load type as **LD_LOAD_MOD**.

load() can handle both absolute and relocatable object files. The term *relocatable* also covers the position-independent code generated by MRI compiler tools.

If the object file is absolute, it is always loaded at the address specified by the object file header, and you may not be able to modify these values. Also, it is assumed that it is safe to load an absolute object file at the address specified by the object file header. If the object file is relocatable, then the memory needed to load the object file is automatically allocated by **load()**.

For relocatable object files, you can control the loading of file on a per-section basis by modifying the relevant fields in the **OF_INFO** structure returned by calling **load()** with load type **LD_GET_INFO,** and passing the modified structure to **load()** with load type **LD_LOAD_MOD**.

On success, **load()** returns 0; otherwise, it returns a non-zero error number.

The following errors are returned by **load()**:

ERR_SYNTAX	The loader encountered a syntactic construct in the object file that is not understood by the loader.
ERR_INVAL	An invalid operation was attempted (like trying to call **load()** with flags **LD_LOAD_MOD** without previously calling **load()** with flags **LD_GET_INFO**). Also, this error is returned if the **desc** field of **of_info** is invalid, or an invalid flag is specified.
ERR_NO_OFM	The format of the object file being loaded is not supported by the loader.
ERR_OFM_FULL	An attempt was made to load an object file while the configured maximum number of files has already been loaded and has neither been released nor unloaded.
ERR_UNSUPP	The object file being loaded contains some unsupported feature (like an IEEE-695 relocatable file containing unresolved externals).
ERR_NOT_EXEC	The object file did not compile properly and is not ready for execution.
ERR_INTERNAL	The loader discovers an inconsistency in the internal data structures.
ERR_TOOBIG	One of the sections of the file being loaded is too big to fit in the memory.
ERR_NOSEG	The object file (or a part thereof) cannot be loaded because of a temporary shortage of memory.

Other errors may be returned due to the failure of either a pSOS+ system call or a call made internally by the loader to pHILE+ or a device driver.

CAUTION: When calling **load()** with flags **LD_GET_INFO** or **LD_LOAD_DEF**, you must not allocate memory for the **of_info** structure, as this is done by **load()**. The proper way of calling **load()** is as follows:

```
#include <loader.h>
OF_INFO *my_of_info;
unsigned long fd, flags;
        ...
        ...
load (fd, flags, &my_of_info);
        ...
        ...
```

The unload() Function

unload() is defined as follows:

```
#include <loader.h>
unsigned long unload (
        OF_INFO *of_info
);
```

This function unloads an executable file image from the target memory, where it was loaded previously using **load()**. **of_info** points to the object file information returned by a previous call to **load()**.

If the type of executable being unloaded is **LD_ABSOLUTE**, the **unload()** function does nothing to free the memory associated with the executable -- it is the responsibility of the caller to free up the memory (if any) that it allocated previously.

If the type of executable is **LD_RELOCATABLE**, this function frees up any memory allocated earlier for loading the executable. However, it does not free any memory for sections of executable files that were allocated by the caller. Those must be taken care of by the caller.

unload() frees up any state information associated with **of_info** and preserved internally by the loader. It also frees up the object file information pointed to by **of_info**, and it must not be referred to subsequently by the caller.

The **unload()** function must be called only after the task(s) associated with the loaded executable have been deleted, since all of the memory allocated to load executable code and data is returned to the free storage pool by **unload()** and can be re-used for any purpose at any time.

On success, **unload()** returns 0; otherwise, it returns a non-zero error number.

The following errors are returned by **unload()**:

ERR_INVALID	The **desc** field of **of_info** is invalid. or you tried to unload an executable that has never been loaded.

Other errors may be returned that can be due to the failure of a pSOS+ system call made internally by the loader.

The release() Function

release() is defined as follows:

```
#include <loader.h>
unsigned long release (
            OF_INFO *of_info
);
```

This function frees up the object file information pointed to by **of_info**, and also any state information associated with **of_info** and preserved internally by the loader. It must be called in one of the following situations:

- You have called **load()** with the **LD_GET_INFO** flag but decide not to load the executable image.
- You have loaded the executable with **load()** by specifying either **LD_LOAD_DEF** or **LD_LOAD_MOD** flags and do not intend to ever unload these executables (that is, if the executable corresponds to task(s) that remain memory resident forever).

The object file information pointed to by **of_info** must not be referred to subsequently by the caller.

On success, **release()** returns 0; otherwise, it returns a non-zero error number.

The following errors are returned by **release()**:

ERR_INVALID	The **desc** field of **of_info** is invalid, or you tried to release a stale **of_info**.

Other errors may be returned that can be due to the failure of a pSOS+ system call made internally by the loader.

The ld_prep Utility

The syntax for **ld_prep** is as follows:

ld_prep **{-a|-r}** **[-v]** **[-d** *defaults_file* **]** **[-n** *task_name* **]**
[-p *priority* **]** **[-c** *create_flags* **]** **[-m** *task_mode* **]**

[-e *entry_point* **] [-s** *supv_stack_size* **]**
[-u *user_stack_size* **] [-o** *out_file* **]** *in_file*

ld_prep is a post-processor that must be run on an object file *in_file* before it can be loaded by the pSOSystem loader. The object file can be in either Motorola SREC format or MRI IEEE-695 format. **ld_prep** analyzes the input object file, prepends a header to it, and writes the file to a user-specified output file *out_file* (or to a file **out.ld** by default). The header contains certain information about the object file that is used by the loader.

You must specify whether the input object file has to be loaded at the absolute address specified at link time or whether it can be relocated by the loader to any address of its choosing. You must specify whether the input object file is absolute or relocatable.

Additionally, if the file being loaded corresponds to a task that will be created and started eventually by the user, it is possible to specify all of the task-specific information using **ld_prep**. This information is passed to the loader application via the **TASK_INFO** sub-structure of the **OF_INFO** structure, the pointer to which is returned by **load()**. This information typically consists of the task name, the priority at which it runs, the sizes of the user and supervisor stacks, the task entry point, and various other task attributes that get passed to **t_create()** and **tstart()**.

You must run **ld_prep** on an object file that needs to be loaded by the loader or else an error will be flagged by the loader at runtime.

The following options are provided:

-a	Specifies that the input object file is absolute.
-r	Specifies that the input object file is relocatable.
-v	Specifies the *verbose* option. Some useful information about the file is printed on **stdout**.
-d *defaults_file*	Specifies the name of the file from which the defaults must be picked up for options not specified on the command line. The *defaults_file* must have one or more lines containing the options as they are specified on the command line. A sample *defaults_file* is shown in the examples.
-n *task_name*	Specifies the user-assigned name of the task. If this option is omitted, the task name is set to **LDBL**.
-p *priority*	Specifies the task's initial priority within the range 1 to 239. If this option is omitted, the priority is set to 0.
-c *create_flags*	Specifies the flags that get passed to **t_create()**. The flags can be one or both of **G** and **F**.

The **G** flag specifies that the task is global and addressable by external tasks residing on other nodes. If this flag is omitted, the task is assumed to be local.

The **F** flag specifies that the task uses floating point units. If this flag is omitted, the task is assumed not to use floating point units.

-**m** *task_mode* Specifies the task mode that gets passed to **t_start()**. The mode can be one or more of **A**, **N**, **T**, and **S**.

A specifies that the task's ASRs are disabled. If this flag is omitted, the task's ASRs are assumed to be enabled.

N specifies that the task is non-preemptible. If this flag is omitted, the task is assumed to be preemptible.

T specifies that the task can be timesliced. If this flag is omitted, it is assumed that the task cannot be timesliced.

S specifies that the task runs in supervisor mode. If this flag is omitted, the task is assumed to run in user mode.

-**e** *entry_point* Specifies the address at which the task execution is to begin. If this option is omitted, **ld_prep** first tries to find out the execution start address from the file and stores that in the header. If it cannot be determined, it is set to 0.

-**s** *supv_stack_size* Specifies the size of the task's supervisor stack in bytes, and must be greater than 128. If unspecified, it is set to 0.

-**u** *user_stack_size* Specifies the size of the task's user stack in bytes, and may be 0 if the task executes only in supervisor mode. If unspecified, it is set to 0.

-**o** *out_file* Specifies the name of the output file that will become input to the loader. If this option is omitted, **ld_prep** creates a file **out.ld** in the current directory by default.

in_file Object file.

If you specify a defaults file using the -**d** option, it is parsed first to pick up the defaults. Next, **ld_prep** parses any command line options. Options specified on the command line override values specified in the defaults file

In most cases, when an option is specified neither in the defaults file nor on the command line, the corresponding parameter is set to 0. When detecting the 0 values, the loader application must determine the appropriate values to use. Note that you must specify either the -**a** or -**r** option, either in the defaults file or on the command line; otherwise, an error is flagged by **ld_prep**.

ld_prep exits with status 0 upon successful execution; otherwise, it exits with exit status **1** and an error message is printed to **stderr**. The error messages are self explanatory.

Examples

```
ld_prep -r -o app.ld app.x
```

```
ld_prep -a -v -p 180 -n NApp -cF -mAT -e 0x3c0000 -s 512 -o napp.ld newapp.x
```

is the same as

```
ld_prep -d task.defs -o napp.ld newapp.hex
```

where the file **task.defs** contains the following line:

```
-a -v -p 180 -n NApp -cF -mAT -e 0x3c0000 -s 512 -o app.ld
```

Note that a command line option overrides the options specified in the defaults file (-**o** in the above example).

Warnings

If an option is specified more than once on the command line, the last (rightmost) such definition takes precedence over any previous definition. However, if an option is specified more than once in the defaults file, the behavior of **ld_prep** is undefined.

A warning is issued if the defaults file contains the -**d** option and the option is ignored.

Supported Platforms

The **ld_prep** utility is provided for Sun SPARCstations, Hewlett Packard series 700 workstations, IBM RS6000 workstations, and machines running MS-DOS.

Compiling and Running Applications Using the pSOSystem Loader

The procedure for compiling and running applications using the loader is as follows:

1. Write the loader task, then compile and link it with the loader library and pSOSystem to generate the **ram.hex** file.
2. Next, decide whether to use the SREC format or the IEEE-695 format for the applications that get loaded through the loader task. The SREC format must be chosen if you are generating position-independent code or if the application’s location in target memory can be determined at compile time. The IEEE-695 format must be chosen when you cannot generate position-independent code and it is not possible to determine at compile-time where the application gets loaded in target memory. The IEEE-695 can also be chosen for loading absolute code.

3. Change the **sys_conf.h** configuration files and **Makefiles** provided with the sample loader application for your application and use it to generate **app.hex** (the SREC version) or **app.x** (the IEEE-695 version) files for the application task, which are to be loaded with the loader.
4. Run the **ld_prep** utility with **app.hex** (or **app.x**) as the input file. On the **ld_prep** command line, specify the entry point, the code type (relocatable/absolute), and any other parameters that may be appropriate. A file **out.ld** will be generated, by default, in the current working directory. If you want, you can specify a name of your choice (instead of **out.ld**) using the **-o** command-line option to **ld_prep**.
5. Copy the file produced in Step 4 to the appropriate file system volume and directory from where the loader has been programmed to load this application. For example, when using TFTP pseudo driver to load applications, you may need to copy this file to the **/tftpboot** directory on certain host systems that provide a restricted TFTP facility.
6. Using the bootstrap loader on the target, load the **ram.hex** file generated in Step 1 and restart pSOSystem. If the loader task runs successfully, you should be able to load your application.

Guidelines for Writing Device Drivers

As stated earlier, a device driver that interfaces with the loader must meet the interface requirements set by pREPC+. See the guidelines for writing device drivers in Section 2, "Interfaces and Drivers".” The loader calls only the **de_read()** function internally. It passes an I/O parameter block with the following format:

```
typedef struct {
        unsigned long count;    /* Number of bytes to read */
        void *address;          /* Address of data buffer */
        } iopb;
```

The loader needs the device driver to be capable of skipping data (i.e. seeking in the forward direction). To seek in the forward direction, the loader calls **de_read()** with the **count** field in the **iopb** structure set to the number of bytes to skip, and with the **address** field in the **iopb** structure set to **(void *)NULL**.

The device driver read function, on receipt of an **iopb** structure with **address** field set to **NULL**, reads **count** number of bytes from the device and discards those. Thus, this case is treated the same as any other **read** operation, except that the driver does not copy the data. This is the only additional requirement set by the loader, and it is very easy to implement. For an example, you can refer to the TFTP pseudo device driver sources that are provided with pSOSystem in the **drivers** directory.

NAME

mmulib -- mmu library

DESCRIPTION

mmulib included in the pSOSystem base package provides the following memory management services for 68030, 68040, and 68060 processors:

- Creation of MMU maps (mapping tables for the Memory Management Unit).
- Control of attributes for individual pages in those maps.
- Activation of the maps and enabling of the MMU.

mmulib supports only a logical equal-to-physical mapping. This allows the access characteristics of memory to be managed without introducing the complexities associated with virtual addresses.

mmulib can be used to *disable caching* of certain areas of memory. This is most useful for memory that is used for I/O, DMA (direct memory access), or memory that is shared between multiple CPUs (and hardware snooping is not implemented).

mmulib can also be used to *restrict write access* to certain areas of memory. This allows protection of both the OS and application code. Additionally, access to certain data areas can be limited to a particular task or set of tasks.

A *map* specifies the *access characteristics* of a segment of memory. That segment can be as large as the entire 4 Gigabyte address space. A map divides an address space into *pages*, each containing 4 Kbytes. All memory locations within the same page have the same access characteristics. A particular map is implemented via *mapping tables* stored in memory and is read by the MMU hardware.

Several maps can exist simultaneously; however, only one may be in use by the MMU hardware at any given time. The map that is in use by the MMU hardware is referred to as the *active map.*

In addition to the memory described by the MMU maps, other sections of memory can be described by Transparent Translation Registers**.** For example, you might want to use MMU mapping to control the access characteristics only in physical RAM. A Transparent Translation Register can then control the access characteristics for the memory-mapped I/O area.

A page (and the memory it contains), which is explicitly described by a mapping table entry, is said to be *defined* by the map. All other pages are *undefined*. Any attempt to access a page that is not defined by the active map causes a hardware exception.

mmulib Concepts and Operation

Control of memory defined by the MMU map is done on the scale of a page. Pages defined by a map have associated *attributes* that further control access to memory within that page. Every defined page may have none, some, or all of the following attributes:

Invalid	Accessing the page is disabled. Any attempt to access it causes a hardware exception.
Read-Only	The page is write protected. Any attempt to write to it causes a hardware exception.
Cache-Disabled	The page is not cached.
Copyback	If cache is enabled, use copyback as opposed to write-through mode. (68040 and 68060 only)
Serialized	If cache is disabled, use serialized access. (68040 and 68060 only)
Supervisor	The page can be accessed only in supervisor mode. (68040 and 68060 only)

mmulib lets you provide a default map and additional maps associated with individual tasks. Additional **mmulib** services examine and modify maps, and they integrate **mmulib** with pSOS+ and pROBE+ operations.

Mapping tables are created by the **mmulib** call **map_create()**. The caller provides a description of the desired map via a *map template*. Memory to store the mapping tables can be provided by the caller, or it can be dynamically allocated by the **map_create()** call. **map_create()** sets a *map ID,* which is used in subsequent **mmulib** calls. Numerous mapping tables can be created by calling **map_create()** multiple times.

Once created, any page defined by the map can be examined with the **map_getattr()** call. The attributes associated with any page defined by the map can be altered via the **map_setattr()** call. However, undefined pages can not be added to the map. Thus, the map template should define enough memory to cover anticipated needs, even if this amount is initially invalid.

Following the creation of at least one map, the MMU can be enabled and a map made active. You can control which map is active at any time by defining a *default map,* and you can define other maps associated with one or more tasks. When a task with no associated map is executing, the default map is the active map. Otherwise, the task's associated map is the active map. The user and supervisor mode maps are always the same.

mmulib assumes the MMU is disabled when the ROOT task begins execution. The **map_default()** call is used to define the default map and to enable the MMU with the default map active. Note that because it must alter the MMU registers, **map_default()** must be called from supervisor mode.

If all tasks are to use the default map (for example when the MMU is simply being used to inhibit data caching in certain memory areas), then no further action is required. If some

tasks require a map that is different from the default map, then **map_task()** is used to associate an alternate map with a task.

As can be seen, a single map can be associated with many tasks. The default map can also be associated with one or more tasks through the use of the **map_task()** call (although this is a superfluous operation).

Finally, the **map_getid()** service returns either the ID of the map associated with a task or the ID of the default map.

Page Attributes

When application code exchanges page attributes with **mmulib**, a bit map is used. For each attribute, the bit positions and meanings are defined in **mmulib.h** by the following constants:

MAP_INVBIT When set, the page is invalid.

MAP_WPBIT When set, the page is read only.

MAP_CIBIT When set, the page is cache inhibited.

MAP_CBBIT When set, copyback mode is used. (68040 and 68060 only)

MAP_SERBIT When set, serialized access is used. (68040 and 68060 only)

MAP_SUPBIT When set, the page can be accessed only in supervisor mode. (68040 and 68060 only)

Map Template

You can create mapping tables by using the **map_create()** call, which puts information into an array of structures. Each **map_t** structure describes one contiguous area of physical memory, defined in **mmulib.h** as follows:

```
struct map_t
       {
       void          *addr,
       unsigned long len,
       unsigned long attr
       }
```

where **addr** is the start address of the section, **len** is it length in bytes, and **attr** specifies its initial attributes. **addr** must be on a page boundary and **len** must be a multiple of the page size. **attr** is created by OR-ing together any combination of MAP_INVBIT, MAP_WPBIT, MAP_CIBIT, MAP_CBBIT, MAP_SERBIT, and MAP_SUPBIT.

PAGE_ALIGMENT Mask that can be used to test an address to see if it is aligned to a page boundary.

PAGE_SIZE Define statement that gives the size of a page of memory.

MAP_SIZE(mem_size) Macro that will give the amount of memory needed to hold the map tables for a size of contiguous memory. **mem_size** must be in bytes.

mmulib Functions

mmulib provides two types of functions, *user-callable* and *callout*. User-callable functions are called from a user application. Callout functions are called from pSOS+ and pROBE+ code. **mmulib.h** contains prototypes of all **mmulib** functions.

User-Callable Functions

User-callable **mmulib** functions are as follows:

map_create	Create a memory map.
map_default	Define default map and enable MMU.
map_getattr	Get the attributes of a page.
map_getid	Get the ID of a task's map or of the default map.
map_setattr	Change the attributes of a map.
map_task	Associate a map with a task.

The syntax of the **map_create()** function is as follows:

```
#include <mmulib.h

map_create(   struct map_t *map,
              unsigned long     maplen,
              void              *mapmem,
              unsigned long     mapmemlen,
              unsigned long     *mapid,
              unsigned long     *tablesize
              )
```

map_create() creates mapping tables from a map description provided by the array of **map_t** structures pointed to by **map**. **maplen** specifies the number of array elements in **map**.

mapmem points to the memory area to hold the mapping tables and **mapmemlen** specifies the length, in bytes, of **mapmem**. If **mapmem** is zero, **mmulib** allocates memory for the mapping tables from Region 0. In this case, **mapmemlen** is ignored. If **mapmen** is too small, or Region 0 lacks sufficient space, a fatal error occurs, an error code is returned, and mapping tables are not created.

mmulib calculates the amount of memory required to hold the map tables and returns it in the variable pointed to by **tablesiz**e. If a new map is successfully created, the ID of the

map is returned in the variable pointed to by **mapid**. This ID is then used in subsequent calls to **mmulib**.

map_create() returns 0 or an error code, which can be one of the following:

EMMU_INSUFMEM	Map area was too small or **map_create()** could not allocate enough memory from Region 0.
EMMU_DUP_PAGE_ENTRY	A duplicate page is referenced in the **map_t** array.
EMMU_ADDR_NOT_ON_PAGE	The starting address of a section is not on a page boundary.
EMMU_LEN_NOT_PAGE_MULT	The length of a section is not a multiple of the page size.

The syntax for **map_default()** is as follows:

void map_default (unsigned long mapid)

map_default() makes the map specified by **mapid** the default map and enables the MMU. Unless the calling task has an associated map, upon return, the map specified by **mapid** is active. Since it enables the MMU, **map_default()** must be called from supervisor mode or a privilege violation occurs.

map_default() is normally called just once. However, if it is called multiple times, the most recent call determines the default map.

map_default() has no return value. It does not check the validity of **mapid**. If it is not valid, a hardware exception or other erroneous behavior is likely to result.

The syntax for **map_task()** is as follows:

map_task (unsigned long tid, unsigned long mapid)

map_task() associates the map specified by **mapid** with the task specified by **tid**. If **mapid** is zero, then the task's current association, if any, is removed so that the task subsequently uses the default map. If **tid** is zero, then calling task's map is changed.

The new map does not become active until the task next gains the CPU through a context switch. If a task sets its own map and needs it to be active before proceeding, the task should use **tm_wkafter()** to block for one clock tick.

map_task() returns 0 or the following error code:

EMMU_TID_NOT_VALID	The task ID is not valid.

map_task does not check the validity of **mapid**. If not valid, a hardware exception or other erroneous behavior will probably result.

map_setattr() alters the attributes of a contiguous memory area. **map_setattr()** syntax is as follows:

```
map_setattr(
        unsigned long    mapid,
        void             *addr,
        unsigned long    len,
        unsigned long    mask,
        unsigned long    attr
        )
```

mapid specifies the map to be changed. **addr** specifies the start address of the memory region and **len** specifies its length. If **len** is zero, then the attributes of the single page containing **addr** are changed. Otherwise, **addr** must be on a page boundary, and **len** must be a multiple of the page size.

mask specifies which attributes of the region are to be changed. It is formed by OR-ing together any or all of the page attributes:

MAP_INVBIT, MAP_WPBIT, MAP_CIBIT, MAP_CBBIT, MAP_SERBIT, MAP_SUPBIT

attr specifies the new attributes of the region and is also formed by OR-ing together any or all of the above attributes. If a bit is set in **mask**, then the value in the corresponding bit of **attr** is assigned to the page(s). If a bit is not set in **mask**, the corresponding bit in **attr** is ignored and remains unchanged.

If the map is active, the changes take effect immediately. Otherwise, they take effect the next time the map becomes active.

map_setattr() returns 0 or an error code. Possible error returns by **map_setattr** are:

EMMU_ADDR_NOT_ON_PAGE	The starting address of a section is not on a page boundary.
EMMU_LEN_NOT_PAGE_MULT	The length of a section not a multiple of the page size.
EMMU_PAGE_NOT_DEFINED	A page in the memory area is not defined.

map_getattr() returns the attributes of a given page as defined by the map specified by **mapid**. The syntax for **map_getattr()** is as follows:

map_getattr (unsigned long mapid, void *addr, unsigned long *attr)

addr is any memory location within the page. The pages attributes are returned in the variable pointed to by **attr**. The value returned is formed by OR-ing together any or all of the following page attributes, as appropriate:

MAP_INVBIT, MAP_WPBIT, MAP_CIBIT, MAP_CBBIT, MAP_SERBIT, MAP_SUPBIT

map_getattr() returns 0 or the following error code:

EMMU_PAGE_NOT_DEFINED	A page in the memory area is not defined.

The syntax for **map_getid()** is as follows:

```
unsigned long map_getid (
                unsigned long   tid,
                unsigned long   *defmapid,
                unsigned long   *taskmapid
                )
```

map_getid() returns, in the variables pointed to by **defmapid** and **taskmapid**, respectively, the ID of the default map and the ID of the map associated with the task specified by **tid**. A **tid** of zero refers to the calling task. If the specified task has no associated map, then zero is returned in **taskmapid**.

map_getid() returns 0 or the following error code:

EMMU_TID_NOT_VALID	The task ID is not valid.

Callout Functions

Callout **mmulib** functions are as follows:

map_csco/map_cscoa	Context switch callout procedure.
map_reco	pROBE+ entry callout procedure.
map_rxco	pROBE+ exit callout procedure.

map_cocs() and **map_cocsa()** are special-purpose procedures that change the active map when a context switch occurs. Unless all tasks use the default map, one of these two procedures must be called from the pSOS+ context switch callout. The syntax for these functions is as follows:

void map_csco (unsigned long old_tid, unsigned long new_tid)

void map_cscoa(void)

void map_cscoa() is called from assembly language and expects its input parameters to be in processor registers, as described in the *pSOSystem Programmer's Reference* manual in the Section 4, "Configuration Tables." The address of **map_cscoa()** can be entered directly into the **kc_startco** entry in the pSOS+ Configuration Table.

void map_csco (unsigned long *TCB, unsigned long tid) is called using C language calling conventions. As such, it cannot be entered directly into **kc_startco**. This syntax is used when a C language callout procedure written in C needs to call **map_csco**. **TCB** is the pointer to the task control block of the task that is gaining the CPU**. tid** is the task ID of the task that is gaining the CPU.

map_reco() and **map_rxco()** are special purpose procedures that can be called from the user supplied pROBE+ **ENTRY** and **EXIT** callouts, respectively. They must be used together. The syntax for these functions is as follows:

void map_reco(void)

void map_rxco(void)

map_reco() stores the current MMU state and then disables the MMU. This ensures that pROBE+ has complete access to all physical memory. **map_rxco()** restores the MMU to its state prior to the last **map_reco()** call.

The addresses of **map_reco()** and **map_rxco()** can be entered directly into the **rc_entry** and **rc_exit** entries in the pROBE+ Configuration Table, or they can be invoked from more sophisticated callout procedures.

NAME

NFS Server -- Allow systems to share files in a networked environment

DESCRIPTION

NFS Server contained in the Network Utilities product allows systems to share files in a networked environment. It permits NFS clients to read and write files transparently on pSOSystem disks that pHILE+ manages.

The pSOSystem NFS Server is implemented as two application daemon tasks. The **mntd** task is the mount daemon. It processes requests for mounting and listing exported directories. The **nfsd** task processes all other NFS requests after exported directories have been mounted.

NOTE: The mntd task and nfsd task are provided in the Network Utilities library as position dependent code.

Configuration and Startup

NFS Server requires:

- pSOS+ Real-Time Kernel.
- pHILE+ File System Manager.
- pNA+ TCP/IP Network Manager.
- pRPC+ Remote Procedure Call Library.
- pREPC+ Run-Time C Library.
- Zero Kbytes of user stack and 16 Kbytes of supervisor stack.
- Two UDP datagram sockets used to listen for client requests (port number 2049 is bound to one of these sockets and therefore unavailable) set in pNA+ configuration table.
- Two Kbytes of dynamic storage (which a pREPC+ **malloc()** system call allocates) from region 0.
- A user-supplied configuration table.

The user-supplied NFS Server Configuration Table defines application-specific parameters. A template for this configuration table (shown below) exists in the **include/netutils.h** file.

```
struct nfscfg_t
{
    long task_prio;     /* Priority for nfsd task */
    long unix_auth;     /* UNIX authentication-required flag */
    long error_opt;     /* Error reporting option */
    long vol_blksize;   /* System-wide volume block size */
    char *def_vol_name;/* Name of the default volume */
    nfselist_t *elist;/* Pointer to the list of exported directories */
    long reserved[4];  /* Must be zero */
};
```

Definitions for the NFS Server Configuration Table entries are as follows:

task_prio Defines the initial priority of the daemon tasks **mntd** and **nfsd**.

unix_auth Determines if client authorization is checked. If **unix_auth** equals 1 (TRUE), NFS Server checks a client's UNIX ID for the value 0 (indicating a root client) before mounting. If **unix_auth** equals 0 (FALSE), any client on a trusted machine can mount any of the exported directories.

error_opt Relates to error response. If **error_opt** equals 1 (TRUE), NFS Server returns the appropriate error status on operations that attempt to modify file attributes. If **error_opt** equals 0 (FALSE), NFS Server returns **ok** even if the requested operation did not happen. This allows UNIX utilities that modify file attributes to operate on pHILE+ files even when pHILE+ does not behave exactly the same as UNIX.

vol_blksize Defines the system-wide block size of the volumes that pHILE+ manages. The system-wide block size must match the size defined by the pHILE+ Configuration Table entry **fc_logbsize**. However, the notation for the **vol_blksize** value differs from that of **fc_logbsize**, as follows: **vol_blksize** is specified as the actual block size, and **fc_logbsize** is specified as the exponent of 2 for the block size. For example, if **vol_blksize** is 512 bytes, then **fc_logbsize** is 9 ($2^9 = 512$).

def_vol_name Defines the name of the default volume to use when a client issues a mount request without specifying a volume name.

elist Points to a structure. The structure contains a list of exported directories and trusted clients. If **elist** equals 0, NFS Server looks for the export information in the **/etc/exports** file on the default pHILE+ volume (defined by **def_vol_name**). If no such file exists, NFS Server assumes everything in the system is exportable and accepts all mount requests. When **elist** is specified, its structure must be as follows:

```
struct nfselist_t
{
    char *dir_path;      /* Export list */
    char *hlist;         /* List of trusted clients*/
};
```

The following is an example of an export list with three entries:

```
struct nfselist_t nfselist[] =
{
    {"4.0/", "111.111.11.111", 999.999.99.999"},
    {"5.0/etc", 0},
    {0, 0}
};
```

where the first entry permits the client machines with IP addresses 111.111.11.111 and 999.999.99.999 to mount on the root directory / on volume 4.0, and the second entry allows any client to mount on directory **etc** on volume 5.0. The last entry defines the end of the export list.

reserved Reserved for future use, and each must be 0.

NFS Server comes as one object module and must be linked with a user application. Calling the function **nfsd_start(nfscfg)** any time after pSOSystem initialization (by calling ROOT) starts NFS Server. The parameter **nfscfg** points to the NFS Server Configuration Table. If NFS Server is started successfully, **nfsd_start()** returns 0; otherwise, it returns a non-zero value.

EXAMPLE

The following code fragment shows an example configuration table and the call that starts NFS Server. The complete example code exists in the **apps/netutils/root.c** file.

```
#include <nfsdcfg.h>

start_nfs_server()

{
/* NFS server configuration table */

static nfscfg_t nfscfg =
      {
      250,          /* Task priority for nfsd task */

      1,            /* Requires "root" UNIX client to mount */

      0,            /* No error reporting */

      512,          /* System-wide volume block size */
```

```
        "4.0",        /* Default volume name */
        0,            /* Everything exported */
        0, 0, 0, 0    /* Zeros for all reserved entries */
        };
    /* start the NFS server */
    if (nfsd_start(&nfscfg))
        printf("nfsd_start: failed to start\n");
    }
```

The following features are not supported in the current version:

- Symbolic and hard link files.
- File truncation (for example, **ftruncate()** in SunOS).
- **Lseek** beyond the end of a file.
- Specification of a file's attributes for mode (read/write/execute), ownership (uid/gid), and time (accessed/created/modified).
- File locking.
- Mounting MS-DOS volumes.

On the other hand, the following parameters do apply:

- A file's access, create, and modification times are all updated to the same value whenever a file's content is modified.
- All files are owned by **root** (uid=0 and gid=0).
- All users have read, write, and execute permissions for all regular files.
- All users have read and execute permissions for all directory files.
- All users have read permission for all system files.

NAME

pSH+ -- Interactive command line shell

DESCRIPTION

pSH+ contained in the Network Utilities product provides an interactive, command line shell. pSH+ consists of two parts:

- An application task (**pshc**), which provides the console login shell.
- An application daemon task (**pshd**), which listens for connection requests from the Telnet server daemon and dynamically spawns shell tasks to process Telnet logins.

pSH+ is provided as part of the system utilities object library (**sys/libc/netutils.a**). pSH+ contains a set of built-in commands. Commands or complete applications that will be spawned as separate tasks can be added to pSH+.

Configuration And Startup

pSH+ requires the following components:

- pSOS+ Real-Time Kernel.
- pHILE+ File System Manager.
- pNA+ TCP/IP Network Manager.
- pREPC+ Run-Time C Library.
- Four Kbytes of user stack and two Kbytes of supervisor stack per session.
- One TCP socket, which is used to listen for Telnet server requests, and two additional TCP sockets per shell session.
- Two Kbytes of dynamic storage (which a pREPC+ **malloc()** system call allocates).

The user-supplied pSH+ Configuration Table defines application-specific parameters. The following is a template for this configuration table. The template exists in the file **include/netutils.h**:

```
struct pshcfg_t {
      long flag;               /* Services options */
      long task_prio;          /* Priority for each shell task */
      char *def_vol_name;      /* Default login volume name */
      struct ulist_t *ulist;   /* List of permitted users */
      appdata_t *app;          /* Pointertr to the list of user apps */
      cmddata_t *cmd;          /* Pointer to the list of user cmds */
      unsigned long console_dev;/* The pSH+ console device number */
      unsigned long psedo_dev; /* pSH+ pseudo device number */
```

```
char *cprompt;          /* Console prompt */
char *tprompt;          /* Telnet prompt */
long reserved[2];       /* Must be 0 */
};
```

Definitions of the pSH+ Configuration Table entries are as follows:

flag Specifies which services to provide. If this field is 1, only the part of pSH+ that serves the console login is activated. If this field is 2, only the part of pSH+ that serves Telnet logins is activated. If **flag** is neither 1 nor 2, both parts of pSH+ are activated.

task_prio Defines the priority at which shell tasks start executing.

def_vol_name Names of the default volume to use when a user logs into pSOSystem.

ulist Points to a list of structures. The structures contain login information about permitted users. If this field is 0, any user can log in. The structure format is as follows:

```
struct ulist_t
   {
   char *login_name;           /* User name */
   char *login_passwd;         /* User password */
   long reserved[4];           /* Must be 0 */
   };
```

If **ulist** is provided, the last structure in the array must be all 0's to indicate the end of the list. The following example defines two users:

```
struct ulist_t ulist[] = {
   {"guest", "psos0",    0, 0, 0, 0},
   {"scg",   "andy0",    0, 0, 0, 0},
   {0,       0,          0, 0, 0, 0}
   };
```

app Points to a list of structures. Each of the structures contains information for executing a user application. The **app** entry allows users to add system applications (FTP, Telnet, and so on) and user-defined applications to the shell. (pSH+ comes without built-in user applications. The subsection "Adding Applications to pSH+" on page 2-60 explains how to specify user applications.) The structure format is as follows:

```
struct appdata_t {

   char  *app_name;            /* Application name */
   char  *app_help;            /* Help string */
   void  (*app_entry)();       /* Entry point */
```

```
    char  *app_tname;           /* Task name */
    long  app_tprio;            /* Task priority */
    long  app_sssize;           /* System stack size */
    long  app_ussize;           /* User stack size */
    short app_reentrant_flag;   /* Reentrant flag */
    short app_reentrant_lock;   /* Reentrant lock */
    };
```

The last structure in the array must be all 0's to indicate the end of the list. The following is an example with two entries:

```
struct appdata_t appdata[ ] = {
    {"ftp", "file transfer application", ftp_main,
        "ft00", 250, 2048, 2048,1, 0},
    {"telnet", "telnet application", telnet_main, "tn00",
        250, 2048, 2048, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0}
    };
```

cmd

Points to a list of structures. The structures contain information for executing user commands. pSH+ comes with a number of built-in commands (such as **cd**, **pwd**, **ls**), and users can add commands. The **cmd** entry allows users to add commands to the shell. (The subsection "pSH+ Built-in Commands" on page 2-42 describes built-in commands, and "Adding Commands to pSH+" on page 2-40 explains how to specify user-defined commands to pSH.) The structure format is as follows:

```
struct cmddata_t {
    char  *cmd_name;            /* Command name */
    char  *cmd_help;            /* Help string */
    void  (*cmd_entry)();       /* Entry point */
    short cmd_reentrant_flag;   /* Reentrant flag */
    short cmd_reentrant_lock;   /* Reentrant lock */
    };
```

The last structure in the array must be all 0's to indicate the end of the list. The following is an example with three entries:

```
struct cmddata_t cmddata[] = {
    {"type", "list content of a file", type_main, 1, 0},
    {"volume", "show current working volume",
         volume_main, 1, 0},
    {0, 0, 0, 0, 0}
    };
```

For each built in command, users can add an entry to cmddata[], for example:

```
{"xxx",xxxman,psys_xxx,1,0}
```

The type definitions for xxxman and psys_xxx is in **<netutils.h>.**

NOTE: xxx is any built-in command supported by network utilities library.

console_dev Represents the console device for this application.

psedo_dev Stands for the pseudo device, used for I/O redirection. It is set to DEV_PSEDO, and is defined in **sysconf.h** The pseudo driver is provided as part of the network utilities package. The pseudo driver is used for redirecting socket or file I/O to standard I/O and vice-versa.

cprompt Stands for the pSH+ prompt. If null, the cprompt is set to "<pSH+>".

tprompt Telnet prompt for incoming telnet connections. If null, the tprompt is set to "<pSH+>".

reserved Reserved for future use, and each word must be 0.

Making the **psh_start(pshcfg)** system call from the application starts pSH+. The parameter **pshcfg** is a pointer to the pSH+ Configuration Table. If pSH+ is started successfully, **psh_start()** returns 0; otherwise, it returns a non-zero value.

The following code fragment shows an example configuration table and the call that starts pSH+. The complete example code exists in the **apps/netutils/root.c** file.

```
start_psh()
{
/* user configured command list */
cmddata_t cmds_tab[] = {
                      {"arp", arpman, psys_arp, 1, 0},
                      {"cat", catman, psys_cat, 1, 0},
                       ...
                      {0,0,0,0,0}
                      };
{
/* psh configuration table */
static struct pshcfg_t pshcfg =
       {
       0x03,         /* Services options */
       250,          /* Priority for each shell task */
       "4.0",        /* Default login volume name */
       0,            /* List of permitted users */
       0,            /* Pointer to the list of user apps */
       0,            /* Pointer to the list of user cmds */
```

```
        DEV_CONSOLE, /* Console device */
        DEV_PSEDO,   /* Pseudo device */
        "pSH+>",     /* Pshell prompt */
        "Telnet+>",  /* Incoming telnet connection prompt */
        0, 0,        /* Must be 0 */
        };
/* start the FTP server */
if (psh_start(&pshcfg))
        printf("psh_start: failed to start\n");
}
```

Adding Commands to pSH+

The command set of pSH+ can be extended by specifying the command handlers in a table. The **cmd** entry in the pSH+ Configuration Table must contain the address of this table. The pSH+ task then executes these commands as subroutines. This differs from applications because they execute as separate tasks.

The command table must have the following information about each command:

- The name of the command.
- The starting address of the routine that performs the command.
- An optional **help** string for the command.
- Whether or not the routine that implements the command is reentrant: if a command is not reentrant, pSH+ prevents simultaneous calls to the handler by different shell tasks.

When a shell command is executed, the parameters **argc**, **argv**, and **env** are passed to the subroutine.

The **int argc** parameter is the number of arguments on the command line that were used to invoke the command. The number of arguments includes the command itself.

The **char *argv[]** parameter is an array of pointers to null-terminated character strings, and each string contains one of the arguments to the command as parsed by the shell: **argv[0]** points to the command name as entered on the command line; **argv[1**] points to the first argument (if any); and so on.

The **char *env[]** parameter is an array of pointers to null-terminated character strings. The strings contain the definitions of all of the environment variables. The last element in **env[]** is a null pointer that indicates the end of the environment variables. Two of the environment variables, CVOL and CDIR, define the current working volume and current working directory, respectively.

When a command completes, it exits by executing a **return** to pSH+.

Adding Applications to pSH+

pSH+ applications can be added by placing entries in a table. This table is pointed to by the **app** entry in the pSH+ Configuration Table. Each application described by the table requires:

- The name of the application.
- The starting address of the application.
- An optional **help** string for the application.
- The name and priority to be used for the application task.
- The sizes of the supervisor and user stacks for the application task (in bytes).
- Whether or not the code that implements an application is reentrant. If it is nonreentrant, pSH+ prevents simultaneous instances of the application.

When a shell command is entered with the name of an application, that application is invoked and entered at the specified entry point. The application is also passed four parameters: **argc**, **argv**, **env**, and **exit_param**. The first three parameters are defined in the same way for applications as they are for commands (see the preceding subsection, “Adding Commands to pSH+”). **exit_param** is a parameter used in the **psh_exit** function when an application terminates.

Subroutines

pSH+ provides two subroutines that can be called from the code that implements user-commands and/or applications. The subroutines are **psh_getenv** and **psh_exit**. **psh_getenv** gets the pointer to the value of an environment variable. It has the following syntax:

char *psh_getenv (name, env);

where **name** is the name of an environment variable (for example, CDIR), and **env** is the same parameter passed at the entry point of an application or command.

psh_exit is used when an application that was invoked from pSH+ terminates. It has the following syntax:

void psh_exit (exit_param);

where **exit_param** is the same parameter passed at the entry point of an application.

NOTE: An application cannot interpret the contents of exit_param.

pSH+ Built-In Commands

This subsection contains descriptions of the built-in pSH+ commands. The shell task executes each of these commands as a subroutine. The commands are as follows:

cat	Concatenate and display files.
cd	Change working directory.
clear	Clear the terminal screen.
cmp	Perform a byte-by-byte comparison of two files.
cp	Copy files.
du	Display disk blocks usage.
date	Display or set the date.
echo	Echo arguments to the standard output.
setenv	Set environment variables.
getid	Get NFS user ID and group ID.
getpri	Get task priority.
head	Display the first few lines of the specified files.
help	Display reference manual pages.
kill	Terminate a task.
ls	List the contents of a directory.
mkdir	Create a directory.
mkfs	Construct a pHILE+ file system.
mount	Mount a pHILE+ file system.
mv	Move or rename files.
netstat	Show network status.
nfsmount	Mount a NFS file system.
pcmkfs	Construct an MS-DOS file system.
pcmount	Mount an MS-DOS file system.
ping	Send ICMP ECHO REQUEST packets to network hosts.
popd	Pop the directory stack.
pushd	Push current directory onto the directory stack.
pwd	Display pathname of the current working directory.
resume	Resume a task.
rm	Remove files.
rmdir	Remove directories.
setid	Set NFS user ID and group ID.
setpri	Set task priority.
sleep	Suspend execution for a specified interval.
suspend	Suspend a task.

sync	Force all changed blocks to disk.
tail	Display the last part of a file.
touch	Update the modification time of a file.
umount	Unmount a file system.

The following are descriptions of the pSH+ commands:

cat [-**benstv**] [***filename...***]

Concatenate and display**. cat** sequentially reads each ***filename*** and displays the contents of each named file on the standard output. The following input displays the contents of **goodies** on the standard output:

`psh>` **cat goodies**

Note that **cat** does not redirect the output of a file to the same file. For example, **cat** fails for **filename1** > **filename1** or **filename1** >> **filename1**. You should avoid this type of operation, because it can cause the system to go into an indeterminate state.

cat options are as follows:

b Number the lines, but omit the line numbers from blank lines (similar to -**n**).

e Display non-printing characters, and additionally display a $ character at the end of each line (similar to -**v**).

n Precede each line output with its line number.

s Substitute a single blank line for multiple adjacent blank lines.

t Display non-printing characters (like the -**v** option), and additionally display [TAB] characters as **^I** (a [CTRL]-I).

v Display non-printing characters (with the exception of [TAB] and [NEWLINE] characters), so they are visible. Control characters print like **^X** (for [CTRL]-X); the [DEL] character (octal 0177) prints as ^**?**. Non-ASCII characters (with the high bit set) are displayed as **M-x** where **M**- stands for “meta” and **x** is the character specified by the seven low-order bits.

cd [***directory***]

Change working directory. The argument **directory** becomes the new working directory.

cmp [-**ls**] ***filename1 filename2*** [***skip1***] [***skip2***]

Perform a byte-by-byte comparison of ***filename1*** and ***filename2***. With no arguments, **cmp** makes no comment if the files are the same. If they differ, it reports the byte and line number at which

differences occur, or else it reports that one file is an initial subsequence of the other. Arguments ***skip1*** and ***skip2*** are initial byte offsets into ***filename1*** and ***filename2***, respectively, and can be either octal or decimal (a leading 0 denotes octal).

cmp options are as follows:

l Print the byte number in decimal and the differing bytes in octal for all differences between the two files.

s Silent. Print nothing for differing files.

cp [-**i**] ***filename1 filename2***

cp -**rR** [-**i**] ***directory1 directory2***

cp [-**irR**] ***filename ... directory***

On the first line of the synopsis, the **cp** command copies the contents of ***filename1*** to ***filename2***. If ***filename1*** is either a symbolic link or a duplicate hard link, the **contents** of the file that the link refers to are copied, but the links are not preserved.

On the second line of the synopsis, **cp** recursively copies ***directory1*** along with its contents and subdirectories to ***directory2***. If ***directory2*** does not exist, **cp** creates it and duplicates the files and subdirectories of ***directory1*** within it. If ***directory2*** does exist, **cp** makes a copy of ***directory1*** (as a subdirectory) within ***directory2***, along with its files and subdirectories.

On the third line of the synopsis, each filename is copied to the indicated directory. The basename of the copy corresponds to that of the original. The destination directory must already exist for the copy to succeed.

cp does not copy a file to itself. **cp** options are as follows:

i Interactive: a prompt for confirmation of the copy appears whenever the copy would overwrite an existing file. A **y** answer confirms that the copy should proceed. Any other answer prevents **cp** from overwriting the file.

r See **R**.

R Recursive. If any of the source files are directories, copy the directory along with its files (including any subdirectories and their files). The destination must be a directory.

Example:

In the following example, the first command line entry starts the copy operation. The second command line lists the contents of the directory to verify the results of the copy.

To copy a file:

```
psh> cp goodies goodies.old
psh> ls
goodies   goodies.old
```

To copy a directory, first to a new and then to an existing directory, enter the following:

```
psh> cp -r src bkup
psh> ls -R bkup
x.c  y.c  z.sh
psh> cp -r src bkup
psh> ls -R bkup
src  x.c  y.c  z.sh
src:
x.c  y.c  z.sh
```

date [***yyyymmddhhmm [.ss]***]

Without an input argument, **date** displays the current date and time. Otherwise, **date** sets the current date according to the input argument.

The argument part ***yyyy*** is the four digits of the year; the first ***mm*** is the month number; ***dd*** is the day number in the month; ***hh*** is the hour number (24 hour system); the second ***mm*** is the minute number; and .***ss*** (optional) specifies seconds. If ***yyyy*** is the current year, it can be omitted because the current year value is the default.

Example:

To set the date to Oct 8, 12:45 AM, type

date 10080045

du [-***sa***] [***filename ...***] Display the number of 512-byte disk blocks used per file or directory. This command can display the block count of one or more specified files; all files in either the current or another specified directory; or, recursively, the number of blocks in directories within each specified directory. If no ***filename*** is given,

the current directory (symbolized by a.) is used. Filenames can contain wildcards.

du options are as follows:

s Display only the total for each of the specified filenames.

a Generate an entry for each file.

Entries are generated only for each directory in the absence of options.

Example:

The following is an example of **du** usage in a directory. The example uses the **pwd** command to identify the directory, then uses **du** to show the usage of all the subdirectories in that directory. The total number of blocks in the directory (1211) is the last entry in the display:

```
psh> pwd
/junk

psh> du
5    ./junk1
33   ./xxxxx
44   ./vvvvv/vvvv.junk1
217  ./vvvvv/vvvv.junk2
401  ./vvvvv
144  ./mmmmm
80   ./gggggg
388  ./ffffff
93   ./mine
15   ./yours
1211 .
```

echo [-**n**] [***argument*** **...**]

Echo ***argument***(s) to the standard output. Arguments must be separated by [SPACE] characters or [TAB] characters and terminated by a [NEWLINE].

The -**n** option keeps a [NEWLINE] from being added to the output.

getid Get the user ID and group ID of the shell task.

Example:

```
psh> getid

uid: 23, gid: 140
```

where the second line is output displayed on standard output.

getpri *tname*|-*tid* Return the priority of a task, specified by either the task name (***tname***) or the task identifier (***tid***).

Example:

```
psh> getpri ROOT

ROOT task priority = 250
```

head [-**n**] ***filename...*** Copy the first **n** lines of each filename to the standard output. The default value of **n** is 10.

When more than one file is specified, the start of each file appears as follows:

```
==>filename<==
```

For example, the following line

```
psh> head -4 junk1 junk2
```

produces

```
==> junk1 <==
This is junk file one
==> junk2 <==
This is junk file two
```

help [***command_name***]

Print information about shell commands to the console. If a valid command name is given, **help** prints out information about that command. With no command name for an input, **help** prints out a list of shell commands.

The following example shows the results of **help** without an argument:

```
psh> help

cat cmp echo help mkfs pcmkfs pushd rmdir   sleep
cd cp getid kill mount pcmount pwd setenv
suspendclear date getpri ls mv ping resume setid
sync console du head mkdir  nfsmount popd rm setpri
```

The following example shows the result of **help cat**.

```
psh> help cat

cat - concatenate and display (reentrant, not
locked)
```

kill *tname*|-*tid*

Terminate the task indicted by either the task name (***tname***) or the task identifier (***tid***). The **kill** command does this by calling **t_restart** with a second argument of -**1**. The task must be designed to read this second argument and do its own resource cleanup, then terminate.

Example:

```
psh> kill tftd
```

ls [-***aACdfFgilqrRs1***] ***filename ...***

For each filename that is a directory, **ls** lists the contents of the directory; for each filename that is a file, **ls** repeats its name and any other information requested. By default, the output is sorted alphabetically. With no input arguments, **ls** lists the contents of the current directory.

ls options are as follows:

- **a** List all entries.
- **A** (**ls** only) Same as -**a**, except that the. and the.. are not listed.
- **C** Force multi-column output, with entries sorted down the columns; for **ls**, this is the default when output goes to a terminal.
- **d** If argument is a directory, list only its name (not its contents); often used with -**l** to get the status of a directory.
- **f** Force each argument to be interpreted as a directory and list the name found in each slot. This option turns off -**l**, -**s**, and -**r** and turns on -**a**; the order is the same as the order of the entries appearing in the directory.
- **F** Mark directories with a trailing slash */ and executable files with a trailing asterisk (*).
- **g** For **ls**, show the group ownership of the file in a long output.
- **i** For each file, print the **i**-number in the first column of the report.
- **l** List in long format. Long format shows the mode, the number of links, the owner, the size (in bytes), and the time of each file's last modification. If the last modification occurred more than six months ago, the display format is **month-date-year**; the format for files modified in six or less months is **month-date-time**.

q Display nongraphic characters in filenames as the ? character; for **ls**, this is the default when output goes to a terminal.

r Reverse the order of the sort either to reverse the alphabetic order or list the oldest data first.

R Recursively list subdirectories encountered.

s Give size of each file. Include indirect blocks used to map the file. Display in Kbytes.

l Force single-column output.

mkdir [-**p**] ***dirname...*** Create a directory. The -**p** option allows missing parent directories to be created, as needed.

Example:

```
psh> ls -lR
total 8
-r--r--r--   1 root    512 Mar 31 94 00:00 BITMAP.SYS
-r--r--r--   1 root   2048 Mar 31 94 10:01 FLIST.SYS
drwxrwxrwx   1 root     32 Mar 31 94 13:34 test_dir
./test_dir:

psh> mkdir -p new_dir/next_dir
psh> ls -lR
total 9
-r--r--r--   1 root    512 Mar 31 94 00:00 BITMAP.SYS
-r--r--r--   1 root   2048 Mar 31 94 10:01 FLIST.SYS
drwxrwxrwx   1 root     16 Mar 31 94 13:36 new_dir
drwxrwxrwx   1 root     32 Mar 31 94 13:34 test_dir
./new_dir:
total 0
```

mkfs [-**i**] ***volume_name label size num_of_fds***
Initialize a file system ***volume_name*** and label it with ***label***. The argument ***size*** is the volume size, and ***num_of_fds*** is the number of file descriptors.

The **-i** option initializes a device driver for the device.

Example:

```
psh> mkfs 5.6 HDSK 2096 512
Warning: this operation will destroy all data on the
specified volume.
Do you want to continue (y/n)? y
psh>
```

mount ***volume_name*** [***sync_mode***]

Mount a pHILE+ formatted volume on the file system. (A volume must be mounted before any file operations can be executed on it.)

Permanent (non-removable media) volumes need to be mounted only once. Removable volumes must be mounted and unmounted as required.

The ***sync_mode*** is one of the following:

0 Specifies immediate-write synchronization mode.

1 Specifies control-write synchronization mode.

2 Specifies delayed-write synchronization mode (the default).

Example:

```
psh> mount 5.6/
```

mv [**-if**] ***filename1 filename2***

mv [**-if**] ***directory1 directory2***

mv [**-if**] ***filename... directory***

Move around files and directories in the file system. A side effect of **mv** is that it renames a file or a directory. The three major forms of **mv** appear in the preceding synopses.

The first form of **mv** moves (and changes the name of) ***filename1*** to ***filename2***. If ***filename2*** already exists, it is removed before ***filename1*** is moved.

The second form of **mv** moves (and changes the name of) ***directory1*** to ***directory2*** but only if ***directory2*** does not already exist. If ***directory2*** exists, the third form applies.

The third form of **mv** moves one or more filenames (can also be directories) with their original names into the last directory in the list.

mv does not move either a file to itself or a directory to itself.

mv options are as follows:

i Interactive mode. **mv** displays the name of the file followed by a question mark whenever a move would replace an existing file. If a line starts with **y**, **mv** moves the specified file; otherwise, **mv** does nothing with the file.

f Force. Override any mode restrictions and the **i** option.

netstat [-**airs**]

netstat displays the contents of various network-related data structures in various formats. **netstat** with no option will display all sockets other than the ones related to server tasks.

netstat options are as follows:

a Show the state of all sockets including ones that are listening (server tasks).

i Show the state of all network interfaces.

r Show the routing tables.

s Show per-protocol statistics.

nfsmount ***host_address: host_directory directory***

Mount the remote file system using NFS protocol. The host ***host_address*** should advertise the directory, ***host_directory*** for this command to complete successfully.

pcmkfs [-**i**] ***volume_name size***

Do a **pcinit_vol** of the volume ***volume_name*** for the disk type ***size***, where ***size*** is one of the following:

1 360 Kbyte (5 1/4" double density)

2 1.2 Mbyte (5 1/4" high density)

3 720 Kbyte (3 1/2" double density)

4 1.4 Mbyte (3 1/2" high density)

The -**i** option initializes the device.

Example:

```
psh> pcmkfs 5.3 4
Warning: this operation will destroy all data on the
specified volume.
Do you want to continue (y/n)? y
```

pcmount *volume_name* [*sync_mode*]

Mount an MS-DOS file system ***volume_name***. (A volume must be mounted before any file operations can be executed on it.) The argument ***sync_mode*** can be one of the following:

0 Immediate write synchronization mode.

1 Control write synchronization mode.

2 Delayed write synchronization mode (default).

Example:

```
psh> pcmount 5.3
```

ping [-s] *host_address* [*timeout*]

The **ping** command uses the ICMP protocol's mandatory ECHO_REQUEST datagram to elicit an ICMP ECHO_RESPONSE from the specified host or network gateway. ECHO_REQUEST datagrams (pings) have an IP and ICMP header followed by a **struct timeval** and then an arbitrary number of bytes to pad out the packet. If the host responds, **ping** prints `host is alive` on the standard output and exits. Otherwise, after ***timeout*** seconds, it writes `no answer from host`. The default value of ***timeout*** is 10.

When the **s** option is specified, **ping** sends one datagram per second and prints one line of output for every ECHO_RESPONSE that it receives. No output is produced if no response occurs. The default size for a datagram packet is 64 bytes.

When using **ping** for fault isolation, first **ping** the local host to verify that the local network interface is running.

Example:

```
psh> ping 192.103.54.190
PING (192.103.54.190): 56 data bytes
192.103.54.190 is alive
```

popd — Pop the directory stack and change to the new top directory.

Example:

psh> **pushd test_dir**

psh> **pwd**

5.5/test_dir

psh> **popd**

psh> **pwd**

5.5/

pushd *directory* — Push the current ***directory*** onto the directory stack and change the current working directory to that directory.

Example:

psh> **pwd**

5.5/

psh> **pushd test_dir**

psh> **pwd**

5.5/test_dir

pwd — Display the pathname of the current working directory

Example:

psh> **cd 5.5//usr**

psh> **pwd**

5.5//usr

resume *tname* | -*tid* — Resume a suspended task by the task name (***tname***) or the task identifier ***tid***.

Example:

psh> **resume ROOT**

rm [-fir] *filename...* — Remove (unlink directory entries for) one or more files. If an entry was the last link to the file, the contents of that file are lost.

rm options are as follows:

f Force removal of files without displaying permissions or questions and without reporting errors.

i Prompt whether to delete each file and, under -**r**, whether to examine each directory. (This is sometimes called the interactive option.)

r Recursively delete the contents of a directory, its subdirectories, and the directory itself.

Example:

```
psh> ls -lR
total 9
-r--r--r--   1 root    512 Mar 31 94 00:00 BITMAP.SYS
-r--r--r--   1 root   2048 Mar 31 94 10:01 FLIST.SYS
drwxrwxrwx   1 root     16 Mar 31 94 13:36 new_dir
drwxrwxrwx   1 root     32 Mar 31 94 13:34 test_dir
./new_dir:
total 0
drwxrwxrwx   1 root      0 Mar 31 94 00:00 next_dir
./new_dir/next_dir:
./test_dir:
total 1
-rwxrwxrwx   1 root     33 Mar 31 94 00:00 test_file
psh> rm -rf new_dir
psh> ls -lR
total 8
-r--r--r--   1 root    512 Mar 31 94 00:00 BITMAP.SYS
-r--r--r--   1 root   2048 Mar 31 94 10:01 FLIST.SYS
drwxrwxrwx   1 root     32 Mar 31 94 13:34 test_dir
./test_dir:
total 1
-rwxrwxrwx  1 root    33 Mar 31 94 00:00 test_file
```

rmdir *directory...* Remove each named directory. **rmdir** removes only empty directories.

setenv

setenv *variable_name value*

Change a pSH+ environment ***variable_name*** to a new ***value***. If used without arguments, **setenv** prints a list of pSH+ variables and their values.

Note that the only variable that can be changed is TERM.

Example:

```
psh> setenv
CVOL=5.5
CDIR=/
SOFLIST=5
LOGNAME=guest
IND=0
```

```
OUTD=0
TERM=sun
```

`psh>` **setenv TERM vt100**

`psh>` **setenv**

```
CVOL=5.5
CDIR=/
SOFLIST=5
LOGNAME=guest
IND=0
OUTD=0
TERM=vt100
```

setid *uid gid* Change the ***uid*** and ***gid*** ID of the current pSH+ session.

Example:

`psh>` **getid**

`uid: 23, gid: 140`

`psh>` **setid 2 3**

`psh>` **getid**

`uid: 2, gid: 3`

setpri *tname* | -*tid new_priority*
Set the ***new_priority*** of the task identified by either the task name (***tname***) or task identifier (***tid***).

Example:

`psh>` **getpri ROOT**

`ROOT task priority = 76`

`psh>` **setpri ROOT 252**

`psh>` **getpri ROOT**

`ROOT task priority = 252`

sleep *time* Suspend execution for the number of seconds specified by ***time***.

suspend *tname* | -*tid* Suspend the task identified by either the task name (***tname***) or the task identifier (***tid***).

Example:

`psh>` **suspend tnpd**

sync

Update a mounted volume by writing to the volume all modified file information for open files and cache buffers that contain modified physical blocks.

This call is superfluous under immediate-write synchronization mode and is not allowed on an NFS volume.

Example:

`psh>` **sync**

tail + | -***number*** [**lc**] ***filename***

Copy ***filename*** to the standard output beginning at a designated place.

tail options are typed contiguously and are not separated by dashes (-). The options are as follows:

+***number*** Begin copying at distance ***number*** from the **beginning** of the file. ***number*** is counted in units of lines or characters, according to the appended option **l** or **c**. When no units are specified, counting is by lines. If ***number*** is not specified, the value 10 is used.

-***number*** Begin copying at distance ***number*** from the **end** of the file. The ***number*** argument is counted in units of lines or characters according to the appended option **l** or **c**. When no units are specified, counting is by lines. If ***number*** is not specified, the value 10 is used.

l ***number*** is counted in units of lines.

c ***number*** is counted in units of characters.

touch [-**cf**] ***filename...***

Set the access and modification times of each argument to the current time. A file is created if it does not already exist.

touch options are as follows:

c Do not create file if it does not already exist.

f Attempt to force the touch regardless of read and write permissions on ***filename***.

umount *directory* Unmount a previously mounted file system where ***directory*** is the mount point of the file system. Unmounting a file system causes it to be synchronized (all memory-resident data is flushed to the device).

Example:

```
psh> mount 5.6
psh> cd 5.6/
psh> ls
BITMAP.SYS        FLIST.SYS
psh> cd 5.5/
psh> umount 5.6
psh> cd 5.6/
5.6/: no such file or directory
```

NAME

RARP -- Reverse Address Resolution Protocol

DESCRIPTION

With RARP (Reverse Address Resolution Protocol), you can send a RARP request (for example, from a diskless workstation) and identify a workstation's IP address, or obtain a dynamically assigned IP address from a domain name server (DNS).

The RARP request is a link-layer broadcast with the following syntax:

```
ULONG RarpEth(long (*NiLanPtr)(ULONG fn_code))
```

RarpEth RARP Ethernet broadcast address. The RARP request is a broadcast message to this address.

NiLanPtr Network interface pointer. This parameter is set to the network interface entry procedure (for example, **NiLan**) in the **lan.c** file in the applicable board-support package.

fn_code Pointer to the network interface entry routine.

EXAMPLE of RARP Dialog

The following example shows a typical RARP dialog:

8:0:20:3:f6:24 ff:ff:ff:ff:ff:ff rarp 60

rarp who-is 8:0:20:3:f6:24 tell 8:0:20:3:f6:24

0:0:c0:6f:2d:20 8:0:20:3:f6:24 rarp 24

rarp reply 8:0:20:3:f6:24 at sun

8:0:20:3:f6:24 0:0:c0:6f:2d:20 ip 56:

sun.24999 > bsdi.tftp: 32 RRQ "8CFC0D21.SUN4C"

RARP can be quite useful, but if you need to identify more than an IP address or if you need to query a domain name server (DNS) located across a router, you can use the BOOTP client feature described in "BOOTP Client Code," on page 1-3.

NOTE: If your RARP request fails, it returns a zero (0) for no reply or 0xffffffff for other network errors.

NAME

routed -- routing daemon

DESCRIPTION

The **routed** daemon contained in pSOSystem's Networking Utilities product is an implementation of the Routing Information Protocol, or RIP. **routed** creates two tasks, RTDM and RTDT. The former maintains the daemon's routing tables, exchanges RIP information, and modifies pNA+ routing tables, as appropriate. The latter serves as a timer that wakes up every 30 seconds to remind RTDM to time out some routing information and to broadcast a routing message. The two tasks use a semaphore, RTSM, to achieve mutual exclusion on their shared data.

System/Resource Requirements

To use the **routed** daemon, you must have the following components installed:

- pSOS+ Real-Time Kernel
- pNA+ TCP/IP Network Manager

In addition, the **routed** daemon requires the following system resources:

- Four K bytes of user stack and Four Kbytes of system stack used by RTDM and RTDT.
- Two UDP sockets. One is used to exchange routing information. The other is used to acquire information about the networking interface.
- One pSOS semaphore for mutual exclusion between task RTDM and RTDT.
- The static memory requirement is 2.5 Kbytes. The dynamic memory size is decided by routing entries. For each routing entry, **routed** needs 68 bytes for its routing entry structure and 74 bytes for its interface structure.

The Routing Daemon Configuration Table

routed requires a user-supplied configuration table, defined as follows:

```
struct routedcfg_t  {
   unsigned long priority;
   int   intergtwy;
   int   supplier;
   int   syslog;
   int   maxgates;
   struct gateways *gways;
};
```

```
typedef struct routedcfg_t routedcfg_t;
```

where

priority This defines the priority at which two daemon tasks, RTDM and RTDT, start executing.

intergtwy This flag is set either to 1 or 0. 1 means an inter-network router, which offers a default route. This is typically used on a gateway to the Internet, or on a gateway that uses another routing protocol whose routes are not reported by other local routers.

supplier This flag is set to either 1 or 0. 1 forces **routed** to supply routing information, whether it is acting as an inter-network router or not. This is the default if multiple network interfaces are present, or if a point-to-point link is in use.

syslog This defines the device number of the serial port for the log display. A negative integer means the log display is disabled.

maxgates This defines the number of entries in the **gateways** structure.

gways This is a pointer to the following structure:

```
struct gateways              {
   struct in_addr destination;
   struct in_addr gateway;
   int metric;
   int state;
   int type;
   };
```

The **gateway** structure supplies **routed** with “distant” passive and active gateways that can not be located using only information from the SIOGIFCONF **ioctl()** option. Each parameter is used as follows:

destination Defines destination address in network byte order.

gateway Defines gateway address in network byte order.

metric Defines hop count to the destination.

state Identifies **passive(RT_PASSIVE)**, **active(RT_ACTIVE)** or **external(RT_EXTERNAL)**. A *passive* router does not run **routed** to exchange RIP packets. An *active* router runs **routed** to exchange routing information. The use of an *external* router indicates that another routing daemon will install the route. Active and passive routers are added into the pNA+ routing table. External routers are kept in the **routed** internal routing table. Only active routers are broadcast in RIP packets.

type Indicates whether the destination type is a **host(RT_HOST)** or a **network(RT_NETWORK)**. When the destination is of **host** type, **routed** treats it as a point-to-point link.

Starting the Routing Daemons

In order to use **routed** in an application, you must link the network utilities library. **routed** is started with **routed_start(routedcfg_t *)**. The following code fragment gives an example of a configuration table and shows how to start **routed**:

```
#include <netutils.h>
#include "sys_conf.h"
void start_routed_server() {
  static routedcfg_t rcfg;
  static gateways_t gways[] = {0xC0d8e800, 0xC0d8e702, 2,
                               RT_ACTIVE, RT_NETWORK};
  rcfg.priority = 250;
  rcfg.intergtwy = 0;
  rcfg.supplier = 1;
  rcfg.syslog = DEV_SERIAL+2;
  rcfg.maxgates = 1;
  rcfg.gways = gways;
  if (routed_start(&rcfg))
      printf("routed_start: failed to start\n");
}
```

NAME

Telnet Client -- Supports communication with a remote system running a Telnet Server

DESCRIPTION

The Telnet Client contained in the Network Utilities product supports communication with a remote system that is running a Telnet Server. The Telnet Client runs as an application under pSH+ and is invoked with the following command:

`pSH+ >` **telnet** [**remote_system** [**port**]]

where **remote_system** can be either a system name or an IP address in dot notation. The **port** option specifies the port number on the remote system to establish the connection.

If no arguments are present, Telnet Client enters command mode (indicated by the `telnet>` prompt). In command mode Telnet accepts and executes the commands described under “Telnet Commands” on page 2-58.

Once a connection has been opened, Telnet enters character-at-a-time input mode. Typed text immediately goes to the remote host for processing.

If the **localchars** toggle is TRUE, the user's **quit**, **intr**, and **flush** characters are trapped locally and sent as Telnet protocol sequences to the remote side. Options exist that cause this action to flush subsequent output to the terminal (see **toggle autoflush** and **toggle autosynch** under “Telnet Commands”). The flush proceeds until the remote host acknowledges the Telnet sequence. In the case of **quit** and **intr**, previous terminal input is also flushed.

While a connection to a remote host exists, Telnet command mode can be entered by typing the Telnet *escape character*. Initially, the escape character is **^]** (a [CTRL]-right-bracket). In command mode, the normal terminal editing conventions are available.

Configuration and Startup

A Telnet Client requires:

- pSOS+ or pSOS+m Real-Time Kernel
- pNA+ TCP/IP Network Manager
- pREPC+ Run-Time C Library
- pSH+ interactive shell command
- Four Kbytes of user space
- Four Kbytes of supervisor stack space

pSH+ starts Telnet Client by calling **telnet_main()**. pSOSystem includes a pre-configured version of pSH+ and Telnet Client, but to add Telnet Client to pSH+, an entry for it must be made in the pSH+ list of user applications.

The following shows an example of a user application list that contains Telnet and FTP:

```
struct appdata_t appdata[] = {
   {"telnet", "telnet application", telnet_main, "tn00", 250,
     4096, 4096, 0, 0},
   {"ftp", "file transfer application", ftp_main, "ft00", 250,
     4096, 4096,1, 0},
   {0, 0, 0, 0, 0, 0, 0, 0, 0}
};
```

You can define the other elements in the preceding example ("**tn00**", and so on). The **telnet_main()** function expects four parameters**: argc**, **argv**, **env**, and **exit_param**. Their definitions as follows:

int argc Number of arguments on the command line that were used to invoke the command. The number of arguments includes the command name itself.

char *argv[] Array of pointers to null-terminated character strings, and each string contains one of the arguments to the command as parsed by the shell: **argv[0]** points to the command name as entered on the command line; **argv[1**] points to the first argument (if any); and so on.

char *env[] Array of pointers to null-terminated character strings. The strings contain the definitions of all of the environment variables. The last element in **env[]** is a null pointer that indicates the end of the environment variables. Two of the environment variables, CVOL and CDIR, define the current working volume and current working directory, respectively.

int exit_param Value to use when exiting by way of the **psh_exit()** call.

Telnet Commands

This subsection describes Telnet commands and supported arguments. You need only type enough of each command to uniquely identify it. This also applies to arguments of the **mode**, **set**, **toggle**, and **display** commands.

close Close a Telnet session and return to command mode.

display [***argument***...] Display all or some of the set and toggle values (refer to the **set** and **toggle** descriptions).

? [***command***] Get help. With no arguments, Telnet prints a help summary. If a ***command*** is specified, Telnet prints the help information for that ***command***.

open ***host*** [***port***] Open a connection to the specified host. If no ***port*** number is specified, Telnet attempts to contact a Telnet server at the default port. The host specification must be an Internet address specified in *dot notation*, for example:

`telnet>` **open 999.999.99.999**

quit — Close any open Telnet session and exit Telnet. An EOF (in command mode) also closes a session and exits.

send *arguments* — Send one or more special character sequences to the remote host (more than one argument per command is allowed). The supported ***arguments*** are as follows:

- **escape** — Send the current Telnet escape character. Initially, the escape character is **^]** (input by a [CTRL]-right-bracket).
- **synch** — Send the TELNET SYNCH sequence. This sequence causes the remote system to discard all previously typed--but not yet read--input. This sequence is sent as TCP urgent data (and may not work if the remote system is a 4.2 BSD system: if it does not work, a lowercase r might be echoed on the terminal).
- **brk** — Send the TELNET BRK (Break) sequence, which might be significant to the remote system.
- **ip** — Send the TELNET IP (Interrupt Process) sequence, which should cause the remote system to abort the currently running process.
- **ao** — Send the TELNET AO (Abort Output) sequence, which should cause the remote system to flush all output from the remote system to the user's terminal.
- **ayt** — Sends the TELNET AYT (Are You There) sequence, to which the remote system may or may not respond.
- **ec** — Sends the TELNET EC (Erase Character) sequence, which should cause the remote system to erase the last character entered.
- **el** — Sends the TELNET EL (Erase Line) sequence, which should cause the remote system to erase the line currently being entered.
- **ga** — Sends the TELNET GA (Go Ahead) sequence, which probably has no significance to the remote system.
- **nop** — Sends the TELNET NOP (No Operation) sequence.

? Prints out helpful information for the **send** command.

set *argument value* Set one of the Telnet variables to a specific value. The special value **off** turns off the function associated with the variable. The values of variables can be interrogated with the **display** command. The supported variables follow:

escape This Telnet escape character (initially `**^]**') causes entry into Telnet command mode (when connected to a remote system).

interrupt If Telnet is in **localchars** mode (see **toggle localchars**) and the interrupt character is typed, a TELNET IP sequence is sent to the remote host (see the **send ip** description). The initial value for the interrupt character is taken to be the terminal's **intr** character.

quit If Telnet is in **localchars** mode (see **toggle localchars**) and the **quit** character is typed, a TELNET BRK sequence is sent to the remote host (see also the **send brk** description). The initial **quit** character value becomes the terminal's **quit** character.

flushoutput If the **flushoutput** character is typed and Telnet is in **localchars** mode (see **toggle localchars**), a TELNET AO sequence is sent to the remote host. (See also the **send ao** description.) The initial value for the **flush** character is taken to be the terminal's **flush** character.

erase If Telnet is in **localchars** mode (see **toggle localchars**), a TELNET EC sequence is sent to the remote system when the **erase** character is typed. (See also the **send ec** description) The initial value for the **erase** character becomes the terminal's **erase** character.

kill If Telnet is in **localchars** mode (see **toggle localchars**), a TELNET EL sequence is sent to the remote system when the **kill** character is typed. (See also the **send el** description.) The initial value for the **kill** character becomes the terminal's **kill** character.

status Show the current status of Telnet. The status information also describes the current mode and the peer to which the user is connected.

toggle *argument ...* Toggle various flags (TRUE or FALSE) that control how Telnet responds to events. More than one ***argument*** can be specified. The state of these flags can be checked with the **display** command. The valid arguments are:

localchars If **localchars** is TRUE, the **flush**, **interrupt**, **quit**, **erase**, and **kill** characters are recognized locally (see the **set** description). These five characters are also transformed into appropriate Telnet control sequences (**ao**, **ip**, **brk**, **ec**, and **el**, respectively). Refer also to the **send** description). The initial value for **localchars** is FALSE.

autoflush If **autoflush** and **localchars** are both TRUE, when the **ao**, **intr**, or **quit** characters are recognized and transformed into Telnet sequences, Telnet does not display data on the user-terminal until the remote system acknowledges its Telnet sequence processing by issuing a Telnet Timing Mark. (See also the **set** description.)

autosynch If **autosynch** and **localchars** are both TRUE, when either the **intr** or **quit** characters are typed the resulting Telnet sequence sent is followed by the TELNET SYNCH sequence. (See **set** for descriptions of the **intr** and **quit** characters). This procedure should cause the remote system to begin discarding all previously typed input and continue to do so until both of the Telnet sequences have been read and acted upon. The initial value of this toggle is FALSE.

crmod Toggle RETURN mode. When this mode is enabled, most RETURN characters received from the remote host are mapped into a RETURN followed by a LINEFEED. This mode does not affect the characters typed by the user: it affects only those received from the remote host. This mode is not very useful unless the remote host sends only RETURN (never LINEFEED). The initial value for **crmod** is FALSE.

options Toggle the display of some internal Telnet protocol processing (having to do with Telnet options). The initial value for **options** is FALSE.

netdata	Toggle the display of all network data (in hexadecimal format). The initial value for this toggle is FALSE.
?	Display the legal toggle commands.

Bugs in the Current Version

- No adequate way exists to deal with flow control.
- The normal abort sequence ([CTRL]-C) does not work during a transfer.

NAME

Telnet Server -- Allow remote systems running the Telnet protocol to log into pSH+

DESCRIPTION

Telnet Server contained in the Network Utilities product allows remote systems that are running the Telnet protocol to log into pSH+. It is implemented as a daemon task named **tnpd**. Telnet listens for connection requests from clients and creates server tasks for each Telnet session established by a client.

Configuration and Startup

Telnet Server requires:

- pSOS+ Real-Time Kernel.
- pNA+ TCP/IP Network Manager.
- pREPC+ Run-Time C Library.
- pSH+ interactive shell command.
- Four Kbytes of user stack and four Kbytes of supervisor stack per session.
- One TCP socket, which is used to listen for client session requests, and two additional TCP sockets per session.
- Two Kbytes of dynamic storage (which a pREPC+ **malloc()** system call allocates).
- A user-supplied configuration table.

The user-supplied Telnet Server Configuration Table defines application-specific parameters. The following is a template for this configuration table. The template exists in the **include/netutils.h** file.

```
struct telcfg_t {
       long task_prio;      /* Priority for tnpd task */
       long max_sessions;   /* Maximum number of concurrent sessions */
       char **hlist;        /* List of trusted clients */
       long reserved[2];    /* Must be 0 */
       };
```

Definitions for the Telnet Server Configuration Table entries are as follows:

task_prio	Defines the priority at which the daemon task **tnpd** starts executing.
max_sessions	Defines the maximum number of concurrently open sessions.
hlist	Points to a list of IP addresses of the trusted clients. If this field is 0, Telnet Server accepts connection from any client.
reserved	Reserved for future use, and each must be 0.

Telnet Server comes as one object module and must be linked with a user application. Calling the function **tnpd_start(tnpdcfg)** any time after pSOSystem initialization (when ROOT is called) starts Telnet Server. The parameter **tnpdcfg** is a pointer to the Telnet Server Configuration Table. If Telnet Server is started successfully, **tnpd_start()** returns 0; otherwise, it returns a non-zero value.

EXAMPLE

The following code fragment shows an example configuration table and the call that starts Telnet Server. The complete example code exists in the **apps/netutils/root.c** file.

```
#include <tnpdcfg.h>

start_telnet_server()
{
    /* Telnet Server Configuration Table */
    static telcfg_t telcfg

    {
        250,            /* Priority for tnpd task */
        4,              /* Maximum number of concurrent sessions */
        0,              /* List of trusted clients */
        0, 0            /* Must be 0 */
    };
}

if(tnpd_start(&tnpdcfg))
    printf("tnpd_start failed\n");
```

NAME

TFTP Server -- Allow TFTP clients to read/write files interactively on pHILE+ file systems.

DESCRIPTION

TFTP Server in the Network Utilities component allows TFTP clients to read and write files interactively on the pSOSystem file systems that pHILE+ manages. The transfer modes that are currently supported are **netascii** and **binary**.

TFTP Server is implemented as one application daemon task named **TFD$**. **TFD$** listens for client connection requests on the TFTP PORT. When it detects a connection request, **TFD$** calls on a child to process the request, then it resumes listening.

Two objects are created for communications between a child and the parent tasks. The objects are a semaphore named **TSM4** and an error message queue named **TFEQ**.

Configuration and Startup

TFTP Server requires:

- pSOS+ Real-Time Kernel.
- pHILE+ File System Manager.
- pNA+ TCP/IP Network Manager.
- pREPC+ Run-Time C Library.
- Two Kbytes of supervisor stack for TFD$ and two Kbytes for each session.
- One UDP socket, which is used to listen for client session requests, and one additional UDP socket per session.
- 2656 bytes of dynamic storage per session, which a pREPC+ **malloc()** system call allocates.
- One semaphore.
- One message queue.

The user-supplied TFTP Server Configuration Table defines application-specific parameters. The following is a template for this configuration table. The template exists in the **include/netutils.h** file:

```
struct tftpdcfg_t {
       char *tftpdir;         /* Default directory for files */
       long task_prio;        /* Priority for "TFD$" task */
       long num_servers;      /* Maximum number of concurrent sessions */
       long verbose;          /* 1 - yes; 0 -no */
       long enable_log;       /* Logging 1 = yes, 0 = no */
       long reserved[1];      /* Must be 0 */
       };
```

Definitions for the TFTP Server Configuration Table entries are as follows:

tftpdir Defines the volume and directory that serves as the default TFTP directory for read and write operations. (The runtime path name specified by a client can override **tftpdir**.)

task_prio Defines the priority at which the daemon task **TFD$** starts executing. All child daemon tasks run at level **task_prio** - 1.

num_servers Defines the maximum number of concurrently open sessions.

verbose Determines if log messages are printed by way of a pREPC+ **printf()**. If **verbose** is 1, TFTP Server runs in verbose mode. A 0 disables it.

enable_log Determines the logging code where:

1 = yes
0 = no

reserved Reserved for future use, and each must be 0.

TFTP Server comes as an object module in the networking utilities library. To use it, **sys/libc/netutil.lib** must be linked with the user application. Calling the function **tftpd_start(tftpdcfg)** at any time after pSOSystem initialization (when the ROOT task is called) starts TFTP Server. The parameter **tftpdcfg** points to the TFTP Server Configuration Table. If TFTP Server is started successfully, **tftpd_start()** returns 0; otherwise, it returns a non-zero value.

EXAMPLE

The following code fragment shows an example configuration table and the calls that start and stop TFTP Server. The complete example code exists in the **apps/netutils/root.c** file.

```
#include <netutils.h>
start_tftp_server()
{
    /* TFTP server configuration table */
    static struct tftpdcfg_t tftpdcfg =
    {
        "4.0/tftpboot"   /* Default tftpboot directory */
        250,             /* Priority for tftpd task */
        4,               /* Maximum number of concurrent sessions */
        0,               /* Not verbose */
        0,               /* Not logging */
        0,               /* Must be 0 */
    };
    if (make_dir (tftpdcfg. tftpdir))
        printf("tftpd_start: failed to make directory\n")

    /* start the TFTP server */
    if (tftpd_start(&tftpdcfg))
        printf("tftpd_start: failed to start\n");
    /* do other stuff */
    /* ... */

    /* this usually is not desired */
    if (tftpd_stop())
        printf("tftpd_stop: failed to shut\n);
}
```

The preceding example illustrates the use of **tftpd_stop()**. The **tftpd_stop()** call shuts down TFTP Server gracefully. It frees all the resources that TFTP Server allocated, then returns. A return value of 0 indicates a successful shut down. Otherwise, the return value indicates the error status.

NAME

intro -- Introduction to Section 2: Interfaces and Drivers

DESCRIPTION

The following interfaces and drivers are described in this section:

- Network Interface (NI) (See page 2-3)
- Kernel Interface (KI) (See page 2-21)
- DISI (See page 2-31)
- DISIplus (See page 2-57)
- SCSI (See page 2-93)
- SLIP (See page 2-101)

For more information on driver-related topics, see *pSOSystem System Concepts*.

NAME

NI -- Network Interface

DESCRIPTION

pNA+ accesses a network by calling a user-provided layer of software called the Network Interface (NI). The interface between pNA+ and the NI is standard and independent of the network's physical media or topology; it isolates pNA+ from the network's physical characteristics.

The NI is essentially a device driver that provides access to a transmission medium. The terms *network interface*, *NI*, and *network driver* are all used interchangeably in this manual.

There must be one NI for each network connected to a pNA+ node. In the simplest case, a node is connected to just one network and has just one NI. However, a node can be connected to several networks simultaneously and can therefore have several network interfaces. Each NI must be assigned a unique IP address.

Your Network Interface to pSOSystem should include the following services called by pNA+:

Service	Function Code	Description
NI_BROADCAST	5	Broadcast an NI packet.
NI_GETPKB	2	Get an NI packet buffer.
NI_INIT	1	Initialize the NI.
NI_IOCTL	7	Perform I/O control operations.
NI_POLL	6	Poll for pROBE+ packets.
NI_RETPKB	3	Return an NI packet buffer.
NI_SEND	4	Send an NI packet.

These services are defined by **#define** in the file **include/pna.h**. In addition, the NI can include an interrupt service routine (ISR) to handle packet interrupts.

Packets and Packet Buffers

The fundamental unit of communication in pNA+ is a *packet*. To transmit data, pNA+ prepares a packet and then passes it to the NI for transmission. It is the responsibility of the NI to deliver the packet to the specified destination.

pNA+ supports two types of packet interfaces:

1. pNA+-Independent Packet Interface
2. pNA+-Dependent Packet Interface.

pNA+ determines which type of packet interface the supporting device driver uses, by the setting of the **flag** element in the **ni_init** structure of the interface table entry for each driver. See the file **include/pna.h** for a description of the **ni_init** structure. If the pNA+

Independent Packet Interface is used the IFF_RAWMEM bit is not set. If the pNA+ Dependent Packet Interface is used, the IFF_RAWMEM bit is set.

pNA+-Independent Interface

This interface supports packets that are contained in contiguous blocks of memory called *packet buffers*. When pNA+ calls the NI to send a packet, it passes a pointer to a packet buffer containing the packet. Similarly, when a packet is received, the NI passes the packet to pNA+ by returning a pointer to the packet buffer used to hold the packet.

The NI is responsible for maintaining a pool of packet buffers and allocating them to pNA+. This approach enables the NI to have its own memory management. First, the NI can create packet buffers within an area of memory best suited for direct retrieval and transmission. Second, for purposes required by the communications protocol, the NI often needs an *envelope* for the packet. This is certainly a common requirement for all network connections. In such cases, the NI can easily maintain a pool of envelopes. When pNA+ requests a packet buffer, the NI allocates an envelope, and returns to pNA+ a pointer to the packet that is contained inside the envelope. pNA+ does not need to know about the envelope.

pNA+ uses the **NI_GETPKB** and **NI_RETPKB** services, respectively, to allocate and return packet buffers. The number of packet buffers necessary is dependent on the implementation and hardware requirements of the NI.

pNA+-Independent Packet Transmission

To prepare and send a packet, pNA+

1. Uses **NI_GETPKB** to obtain a packet buffer.
2. Stores data in the packet buffer.
3. Calls **NI_SEND** or **NI_BROADCAST** to send the packet to the destination; these services have the responsibility of returning the packet buffer to the NI packet pool.

pNA+-Independent Packet Reception

On most systems, the arrival of a packet triggers an interrupt. In this case, the following actions occur on the receiving system:

1. The interrupt transfers control to a Packet ISR, which should be part of the NI, and which receives the packet into a packet buffer.
2. For each pending packet (several may arrive nearly simultaneously), the ISR calls the pNA+ **Announce_Packet** entry (see “The pNA+ Announce_Packet Entry’’ on page 2-6) to transfer the packet to pNA+. pNA+ enqueues the packet and returns to the ISR.

3. After all packets have been transferred to pNA+, the ISR exits using the pSOS+ **i_return** system call (see "pROBE+ Debug Support" on page 2-9 for one exception to this rule).
4. pNA+ processes the packet(s) that were just received.
5. pNA+ calls **NI_RETPKB** to return each packet buffer to the NI.

It is also possible to implement a system in which incoming packets are detected via polling by setting an NI's **POLL** flag. If this flag is set, **NI_POLL** is called every 100 milliseconds.

pNA+-Dependent Interface

Internally, pNA+ uses optimized memory management to transfer packets between various protocol layers. Each packet is represented by a linked list of data structure triplets: Message Block, Data Block and Data Buffer.

This interface supports packet transfer using message block linked lists. When pNA+ sends a packet, it passes the NI a pointer to a message block. Similarly, when the driver receives a packet, it attaches a message block to the data buffer and passes pNA+ a pointer to the message block via the **Announce_Packet** entry.

This facility offers maximum performance by eliminating the need for copying between the NI and pNA+. Also, the driver requires less memory to operate, since the need for transmit buffers is eliminated.

A pointer to the memory management routines **pna_allocb**, **pna_esballoc**, **pna_freeb**, and **pna_freemsg** is passed to the NI during **NI_INIT** calls. The NI should use these routines to allocate and deallocate message block triplets.

A pointer to an interface callback function is passed to the NI during **ni-init**. The callback function may be used by the NI to inform pNA+ of changes in the status of the interface.

pNA+-Dependent Packet Transmission

pNA+ calls **NI_SEND** or **NI_BROADCAST** to prepare and send a packet to a destination. pNA+ passes a pointer to a message block list to be transmitted. The services are responsible for freeing the message block linked list.

pNA+-Dependent Packet Reception

Upon receipt of a packet, typically via an Interrupt Service Routine (ISR), the driver performs the following actions:

1. The interrupt transfers control to a Packet ISR, which should be part of the NI, and which receives the packet into a packet buffer.

2. The driver attaches the packet buffer to a message block using the **pna_esballoc** service call. The driver then calls the **Announce_Packet** entry and passes the message block pointer to pNA+. pNA+ enqueues the packet and returns to the ISR.
3. The driver repeats Step 2 for each pending packet. The ISR then exits using the pSOS **i_return** system call. (If you are using pROBE+, there is an exception to this rule. See “pROBE+ Debug Support,” on page 2-9.)
4. pNA+ processes the packet and then calls the free routine (passed via **pna_esballoc**) to free the buffer.

The pNA+ Announce_Packet Entry

When a packet arrives, the NI driver must inform pNA+ by calling the special pNA+ **Announce_Packet** entry. The address of this entry point is passed to the NI by pNA+ as input when it calls **NI_INIT**.

To call **Announce_Packet**, the NI driver pushes six input parameters onto the stack and then uses a JSR instruction to pass control to the **Announce_Packet** entry. pNA+ processes the packet and returns to the NI using an RTS instruction. There are no output parameters.

The following is a list of the input parameters. Each parameter is 32 bits long.

Parameter	Explanation
type	Type of packet. It must be one of the following: 0x00000800 = IP packet 0x00000806 = ARP packet Packets with headers other than IP or ARP are not passed to pNA+; they are discarded by the NI.
buff_addr	Pointer to the packet buffer containing the packet. When **IFF_RAWMEM** is set, **buff_addr** contains a pointer to the message block list containing the packet.
count	Size, in bytes, of the packet.
if_num	Number of the NI that received the packet. Network interface numbers are assigned to each NI by pNA+ during initialization and are returned to the NI by the **NI_INIT** call.
src_addr	Pointer to the source hardware address of the packet.
dest_addr	Pointer to the destination hardware address of the packet.

The C syntax for the announce_packet function is:

```
*announce_packet {
                    unsigned long    type,
                    char             *buff_addr,
                    unsigned long    count,
                    unsigned long    IF_NUM,
                    unsigned long    src_addr,
                    unsigned long    dest_addr,
                    }
```

In the above syntax, **announce_packet** is the function pointer handed down to the NI driver from pNA+ in the NI_INIT service call.

To summarize, upon receiving control via the **Announce_Packet** entry, pNA+ expects the stack to look like the following:

Stack ptr	+ 0	Return addr
	+ 4	type
	+ 8	buff_addr
	+ 12	count
	+ 16	if_num
	+ 20	src_addr
	+ 24	dest_addr

Announce_Packet preserves all registers except D0. D0 is used to return a pSOS+ *status flag*, which is used by the ISR if pNA+ is providing communication facilities for pROBE+ (see "pROBE+ Debug Support" on page 2-9).

pNA+ Interface Callback

pNA+ provides the NI with an interface callback function.The callback function is passed to the NI by pNA+ during NI_INIT. The callback function may be used by the NI to inform pNA+ of changes in the status of the interface. The calling format resembles pNA+ **ioctl()** call.

The NI may set parameters such as the IP Address, IP Mask, IP Destination address for point-to-point links, the MTU, IP broadcast address or the flags of the interface. This callback is only meant to be used for setting and not for retrieving interface parameters. For

instance PPP may use this callback to notify pNA about the new IP address after a negotiation is complete and that the interface is now UP.

To call the interface callback, the NI pushes 4 input parameters on the stack and then uses a JSR instruction to pass control to the **Interface_Callback** entry. pNA+ process the interface request and returns to the NI using the RTS instruction. There are no output parameters.

The following is a list of input parameters. Each parameter is 32 bits long.

Parameter	Explanation
cmd	is the request code. This must be one of: SIOCSIFADDR — Set the interface address. SIOCSIFBRDADDR — Set the IP broadcast address of the NI. SIOCSIFDSTADDR — Set point-to-point address for the interface. SIOCSIFNETMASK — Set the network mask. SIOCSIFMTU — Set the maximum transmission unit of the NI. SIOCSIFFLAGS — Set interface flags field. If the interface is marked down, any processes currently routing packets through the interface are notified. IFF_POLL, IFF_EXTLOOPBACK and IFF_UP flags can be set by using this call.
argbuf	Pointer to the command data. This must be filled with the **ifreq** structure in **pna.h**.
size	Size of the argbuf - must be sizeof(struct **ifreq**)
if_num	Number of the NI making the request. It is the number returned by the NI_INIT call.

To summarize, upon receiving control via the interface callback entry, pNA+ expects the stack to look like the following:

	Offset	
Stack ptr	+ 0	Return addr
	+ 4	cmd
	+ 8	argbug
	+ 12	size
	+ 16	if_num

pROBE+ Debug Support

If pNA+ is used by pROBE+ to communicate with XRAY+, then two additional requirements must supported by the NI.

The first requirement arises because pNA+ (and the NI) may be operating before pSOS+ is initialized. Normally, every ISR should exit by calling the pSOS+ **i_return** system call. Obviously, this is not possible if pSOS+ is not running; therefore, the NI ISR must be coded so that it will exit using an RTE instruction before pSOS+ is initialized and using an **i_return** after pSOS+ is initialized. The following code fragment can be inserted at the end of your NI ISR to implement this feature:

```
            TST.B     PSOS_FLAG     ;IS PSOS INITIALIZED?
            BEQ.S     DO_RTE        ;NO, EXIT WITH AN RTE
            TRAP      #13           ;YES, EXIT VIA I_RETURN
DO_RTE:
            RTE
```

The variable **PSOS_FLAG** must be zero prior to pSOS+ initialization, and non-zero afterwards. There are a number of ways the ISR can detect pSOS+ initialization. However, to simplify the process, **Announce_Packet** returns a pSOS+ *status flag* in register D0. This flag indicates the status of pSOS+ as follows:

0x00000000 = pSOS+ not initialized

0x00000001 = pSOS+ initialized.

By storing the low byte of D0 into **PSOS_FLAG** after each **Announce_Packet** call, the above code fragment operates correctly.

The second requirement is a result of the fact that pROBE+ sometimes polls for incoming packets. The NI must provide an additional NI service called **NI_POLL**, which is described under “NI Services,” later in this section.

If your NI driver ISR is written in C, you may use the **NI_WRAPPER** code provided with pSOSystem instead of the assembly code above. The interrupt vector can be set to the **NI_WRAPPER** code which will in turn call your ISR. When your ISR exits, it should return the value that was returned from the **announce_packet** call. To use the **NI_WRAPPER** code, you should use the SysSetVector utility function that comes with pSOSystem. For example,

SysSetVector (V_ENET, (void(*)())ni_isr, NI_WRAPPER);

V_ENET is the vector number for the interrupt of the Ethernet chip, and **ni_isr** is the interrupt function. **NI_WRAPPER** is a **#define** statement located in the file **include/bspfuncs.h**. It is a flag SysSetVector uses so that it knows to use the **NI_WRAPPER**.

NI Calling Conventions

pNA+ calls the NI services as subroutines with a JSR instruction. The address used is contained in the network interface table entry **nit_entry**. The NI should perform the requested service and return to pNA+ using an RTS instruction. A return value is returned in register D0. Before calling the NI, pNA+ pushes two 32-bit input parameters onto the stack.

The input parameters are as follows:

fn_code An integer indicating the service requested

pblock A pointer to a parameter block which contains input parameters specific to the requested service

In other words, on entry to the NI, the stack looks like the following:

Stack ptr + 0	Return addr
+ 4	fn_code
+ 8	pblock

The following conventions must be observed by the NI:

1. All registers except D0.L, D1.L, A0.L and A1.L must be preserved.
2. The NI must return a value in D0. Return values are documented in individual service descriptions.
3. The NI may use pSOS+ services only if pSOS+ is running. Recall that pNA+ may be started by pROBE+ before pSOS+ has been initialized. In this case, the NI must not call pSOS+. In any case, the NI must never block.

NI Services

The NI services explained in this section must be provided by the specific NI driver. For each service, the structure of the arguments passed to the driver, and the arguments themselves will be explained. The driver may be written in C code. The pSOSystem provides a union that can be used to facilitate the argument passing in C code. This union is called “nientry”. The nientry union is located in the file **include/pna.h**. The syntax to the entry point of the NI driver is as follows:

long NiLan (unsigned long function, union nientry *args)

function Code of the function to execute. Function codes may be one of the following:

Function Code	Description
NI_INIT	NI init call.

NI_GETPKB	NI buffer call.
NI_RETPKB	NI return buffer call.
NI_SEND	NI send packet call.
NI_BROADCAST	NI broadcast call.
NI_POLL	NI poll call.
NI_IOCTL	NI ioctl call.

These codes are **#defines** located in the **include/pna.h** file.

args Pointer to the argument structure for a particular function code. The individual structures, their names, and the specific return value for the function code will be explained in the specific section that covers the function code.

NI_BROADCAST

NI_BROADCAST is called by pNA+ to transmit a packet to all nodes in the network. The parameter block for this service is as follows:

Offset	Field
pblock + 0	buff_addr
+ 4	count
+ 8	type
+ 12	if_num

buff_addr Address of the buffer containing the packet. When **RAWMEM** is set, **buff_addr** contains a pointer to the message block list.

count Size of the packet in bytes.

type Packet type. Its use depends on the data link protocol implemented (Ethernet, token ring, and so on).

if_num Network interface number assigned to this NI.

The C structure in the nientry union for the NI_BROADCAST function is as follows:

```
struct nibrdcast
    {
    char *buff_addr;    /* Address of the packet buffer */
    long count;         /* Size of the packet */
    long type           /* Type of the packet ARP/IP */
    long if_num         /* NI interface number */
    } nibrdcast;
```

An example of addressing the count field of structure is as follows:

args->nibrdcast.count

NI_BROADCAST returns 0 if the packet is successfully broadcast. Otherwise, it returns an error.

Note the following:

1. **NI_BROADCAST** is responsible for returning the packet buffer whether or not the packet is successfully broadcast.
2. This service is similar to **NI_SEND** except that **NI_BROADCAST** transmits the packet to all other nodes in the network. If the medium (Ethernet) permits, this can be accomplished by a single transmission. Otherwise, the packet must be individually sent to each node.
3. If the application does not use ARP or does no IP broadcasts, this service is unnecessary, and pNA+ never calls it.

NI_GETPKB

NI_GETPKB is called by pNA+ to allocate a packet buffer. This call is not necessary for the drivers that support the pNA+ Dependent Packet interface, that is if:

IFF_RAWMEM == TRUE

The parameter block for this service is as follows:

Offset	Field
pblock + 0	count
+ 4	hwa_ptr
+ 8	if_num

count Specifies the size of the requested packet buffer. NI can allocate a larger buffer but not a smaller one. Normally, this parameter is not used (see Notes).

hwa_ptr Points to the destination hardware address. Normally, this parameter is not used (see Notes).

if_num Network interface number assigned to this NI.

The C structure in the nientry union for the NI_GETPKB function is as follows:

```
struct nigetpkb
        {
        long count;          /* Size of the packet */
        char *hwa_ptr;       /* Pointer to dest hardware address */
```

```
long if_num        /* NI interface number */
} nigetpkb;
```

An example of addressing the count field of structure is as follows:

args->nigetpkb.count

NI_GETPKB should return either the address of the allocated buffer, or a -1 if no buffers are available.

Note that in most cases, the NI allocates all buffers from a pool of fixed-size buffers. The input parameters passed by pNA+ can, however, be used to select different sized buffers, based on the size of the requested buffer and the packet's destination.

NI_INIT

NI_INIT is called by pNA+ to initialize the NI. It is called during pNA+ initialization if the NI is defined in the Initial NI Table. Otherwise, it is called when **add_ni()** is used to install the NI.

NI_INIT should initialize the network hardware; create a pool of packet buffers; and initialize all other NI data structures. In addition, it should save the pNA+ **Announce_Packet** Entry address and the network interface number, both of which are passed to **NI_INIT** by pNA+ in the parameter block, as follows:

Offset	Field
pblock + 0	ap_addr
+ 4	if_num
+ 8	ip_addr
+ 12	ni_funcs

ap_addr Address of the **Announce_Packet** entry point and is returned by pNA+. The NI must save this address.

if_num Network interface number assigned to this interface and is returned by pNA+. The NI must save this address.

ip_addr Internet address of the network interface. This is the address provided by the user in the network interface table and is passed by pNA+. Normally it can be ignored.

ni_funcs Pointer passed to memory management routines (**pna_allocb**, **pna_esballoc**, **pna_freeb**, and **pna_freemsg**), and the interface callback routine (**pna_intf_cb**); it points to the **ni_funcs** structure defined in <**pna.h**>.

The C structure in the nientry union for the NI_INIT function is as follows:

```
struct niinit
        {
        long (*ap_addr) ();         /* pNA entry to receive packet */
        long if_num;                /* NI interface number */
        long ip_addr;               /* NI interface IP address */
        struct ni_funcs *funcs;     /* pNA functions (memory) */
        } niinit;
```

An example of addressing the if_num field of structure is as follows:

```
args->niinit.if_num
```

The NI must return a pointer to the hardware address of the network interface. A return value of -1 is interpreted to mean the network interface is not functional.

Note the following:

1. If the interface is not using ARP, the hardware address returned by **NI_INIT** is not used.
2. **NI_INIT** may raise the processor interrupt level. It should never lower the interrupt level. On exit, it must restore the level to its value upon entry to **NI_INIT**. If **NI_INIT** is called from pNA+ initialization, the interrupt level is always 7. If **NI_INIT** is called as a result of an **add_ni()**, the interrupt level is the same as that of the calling task.
3. If called during pNA+ initialization (that is, the NI is in the Initial NI table), then **NI_INIT** must not make pSOS+ system calls. If NI initialization requires pSOS+ services, **NI_INIT** can set a flag that is detected during the next NI call. If **NI_INIT** is called as a result of an **add_ni()** call, pSOS+ services can be used.

NI_IOCTL

NI_IOCTL is called by pNA+ to perform various I/O control operations on the Network Interface. The requested operation is indicated by the value of the **command** element in the parameter block passed to the function.

The **command** element contains a constant, which is defined in the include files **pna.h** and **pna_mib.h**, and can be one of the following:

SIOCSIFADDR:	Inform NI of setting of NI's IP address.
SIOCSIFDSTADDR:	Inform a Pnt-to-Pnt NI of the destination IP address.
SIOCPSOSINIT:	Inform NI that pSOS is initialized.

Multicasting Related Operations

SIOCADDMCAST	Add multicast hardware address for packet reception.

SIOCDELMCAST	Delete multicast hardware address for packet reception.
SIOCMAPMCAST	Map a protocol multicast address to a hardware address. arg is a pointer to ni_map_mcast structure in **pna.h**. The type field in the structure defines the type of protocol. For IP a value of 0x0800 is set. The hardware multicast address must be returned in the field hdwraddr.

MIB-II Related Operations:

SIOCSGIFDESCR:	Get the NI descriptor.
SIOCGIFTYPE:	Get NI type.
SIOCGIFMTUNIT:	Get NI maximum transmission unit.
SIOCGIFSPEED:	Get NI interface speed.
SIOCGIFPHYSADDRESS:	Get NI physical address.
SIOCGIFADMINSTATUS:	Get NI administrative status.
SIOCGIFOPERSTATUS:	Get NI operational status.
SIOCGIFLASTCHANGE:	Get NI last change of status.
SIOCGIFINOCTETS:	Get number of octets received by the NI.
SIOCGIFINUCASTPKTS:	Get number of unicast packets received by the NI.
SIOCGIFINNUCASTPKTS:	Get number of multicast/broadcast packets received by the NI.
SIOCGIFINDISCARDS:	Get number of packets discarded by the NI.
SIOCGIFINERRORS:	Get number of error packets received by the NI.
SIOCGIFINUNKNOWNPROTOS:	Get number of packets with unknown higher layer protocols.
SIOCGIFOUTOCTETS:	Get number of octets sent by the NI.
SIOCGIFOUTUCASTPKTS:	Get number of unicast packets sent by the NI.
SIOCGIFOUTNNUCASTPKTS:	Get number of multicast/broadcast packets sent by the NI.
SIOCGIFOUTDISCARDS:	Get number of outbound packets discarded by the NI due to resource problems.
SIOCGIFOUTERRORS:	Get number of outbound packets discarded due to errors.
SIOCGIFOUTQLEN:	Get length of outbound queue of the NI.
SIOCGIFSPECIFIC:	Get NI specific object.
SIOCSIFADMINSTATUS:	Set NI administrative status.

The parameter block for the **NI_IOCTL** service is as follows:

pblock + 0	command
+ 4	arg
+ 8	if_num

command Operation to be performed by the NI. The operations that can be called by pNA+ are defined in **pna.h**.

arg Argument for the operation indicated by **command**. Unless specified, **arg** is a pointer to data type structure **ifreq** (defined in **pna.h**). For MIB-II related operations, **arg** is a pointer to the data type struct **mib_ifreq**, defined in the C header file **pna_mib.h**.

if_num Network interface number to which the call is made.

The C structure in the nientry union for the NI_IOCTL function is as follows:

```
struct niioctl
    {
    long cmd;           /* ioctl command */
    long *arg;          /* Pointer to ioctl argument */
    long if_num;        /* NI interface IP address */
    } niioctl;
```

An example of addressing the if_num field of structure is as follows:

```
args->niioctl.if_num
```

The NI returns a 0 if successful, or an error value if an error condition exists.

Note the following:

1. MIB-II operations might not be implemented if the application does not require MIB-II support. A call to the NI to retrieve/set the MIB object is made when the application makes an **ioctl()** call on the NI MIB object.
2. The operations SIOCSIFADDR and SIOCSIFDSTADDR are called by pNA+ when an application changes the NI's IP address or the destination's IP address (Point-to-Point links) by using the **ioctl()** function call.
3. NI can implement a private operation, and the call can be made available to the **ioctl()** call.

4. The operation SIOCPSOSINIT is called when pNA+ is initialized by pSOS+. This call is useful when pNA+ is used by pROBE+. Since pNA+'s memory is re-initialized during the pSOS+ initialization, the driver should remove all references to pNA+ data structures. The driver typically has references to message block pointers.

NI_POLL

NI_POLL is called by pNA+ on behalf of pROBE+ to poll for incoming packets. It is only called when pNA+ is being used by pROBE+ to support network debugging.

If a packet has been received, **NI_POLL** must pass the packet to pNA+ using the **Announce_Packet** entry point as described in the *pSOSystem System Concepts* manual.

The parameter block is as follows:

pblock + 0	if_num

if_num Network interface number assigned to this NI by pNA+.

The C structure in the nientry union for the NI_POLL function is as follows:

```
struct nipoll
        {
        long if_num;            /* NI interface number */
        } nipoll;
```

An example of addressing the if_num field of structure is as follows:

args->nipoll.if_num

NI_POLL should always return 0.

Only one packet can be passed to pNA+ with each **Announce_Packet** call. **Announce_Packet** should continue to be called until all packets have been transferred to pNA+.

NI_RETPKB

NI_RETPKB is called by pNA+ to return a packet buffer to the NI. This call is not necessary for drivers that support the pNA+ Dependent Packet interface, that is if:

IFF_RAWMEM == TRUE

The parameter block is as follows:

pblock + 0	buff_addr
+ 4	if_num

buff_addr Address of the packet buffer being returned.

if_num Network interface number assigned to this NI by pNA+.

The C structure in the nientry union for the NI_RETPKB function is as follows:

```
struct niretpkb
        {
        char *buff_addr;     /* Address of the buffer */
        long if_num          /* NI interface number */
        } niretpkb;
```

An example of addressing the if_num field of structure is as follows:

```
args->niretpkb.if_num
```

This service should always return 0.

NI_SEND

NI_SEND is called by pNA+ to send a packet. The parameter block is as follows:

pblock + 0	hwa_ptr
+ 4	buff_addr
+ 8	count
+ 12	type
+ 16	if_num

hwa_ptr Pointer to the hardware address of the destination.

buff_addr Address of the packet buffer containing the packet. When IFF_RAWMEM is set it contains the pointer to the message block list.

count Size of the packet in bytes.

type Packet type. Its use depends on the data link protocol implemented (Ethernet, token ring, and so on).

if_num Network interface number assigned to this NI.

The C structure in the nientry union for the NI_SEND function is as follows:

```
struct nisend
        {
        char *hwa_ptr;        /* Pointer to dest hardware address */
        char *buff_addr;      /* Address of the packet buffer */
        long count;           /* Size of the packet */
        long type;            /* Type of the packet IP/ARP */
        long if_num;          /* NI interface number */
        } nisend;
```

An example of addressing the if_num field of structure is as follows:

```
args->nisend.if_num
```

This service returns 0 if the packet is successfully sent. Otherwise, it returns an error code.

NI_SEND is responsible for returning the packet buffer whether or not the packet was successfully sent. When the RAWMEM flag is set the system call **pna_freemsg** is used to free the message block linked list.

NAME

KI -- Kernel Interface

DESCRIPTION

On every node in a multiprocessor system, user-supplied Kernel Interface (KI) software must be present. Its purpose is to provide a set of standard services that pSOS+m uses to transmit and receive packets.

The pSOSystem supplies a shared memory Kernel Interface for supported boards that can used a shared memory via a VME bus. The pSOSystem contains a generic driver for this purpose. Refer to the chapter on “Understanding and Developing Board Support Packages” in *pSOSystem Getting Started* for more information on the generic driver.

The KI is dependent on the medium and logical protocol chosen for node-to-node communication. For example, the connection may use a memory bus, Ethernet, MAP, point-to-point link, or a mixture of the above. However, the KI interface to pSOS+m is fixed, as are certain restrictions on its implementation and behavior.

A node’s KI must provide the following services called by pSOS+m:

ki_getpkb Get a packet buffer from the KI.

ki_init Initialize the node’s KI.

ki_receive Get a received (incoming) packet.

ki_retpkb Return a packet buffer to the KI.

ki_roster Provide roster information to the KI.

ki_send Send a packet to another node.

ki_time Allow the KI to perform its own timing; for example, to time transmission retries.

pSOS+m calls the above KI operations as simple subroutines. Input parameters are passed in registers. The KI performs the requested service and simply returns to pSOS+m using a subroutine return. Output parameters, if any, are also passed in registers. The operations and their calling interfaces are described in detail later in this section.

Packets And Packet Buffers

The fundamental unit of communication between nodes is a packet. Whenever pSOS+m needs to communicate with its counterpart on another node, it prepares a packet and then passes it to the KI for transmission. It is the responsibility of the KI to reliably deliver the packet to the destination node.

Packets are physically contained within packet buffers. When pSOS+m calls the KI to send a packet, it passes a pointer to a packet buffer containing the packet. Similarly, when a packet is received, the KI passes the packet to pSOS+m by returning a pointer to the packet buffer used to hold the packet.

The KI is responsible for maintaining a pool of packet buffers and allocating them to pSOS+m. This approach results in optimum efficiency, notably by eliminating any need for the KI to copy the packet. First, the KI can create the packet buffers within a memory area best suited for direct retrieval and transmission. Second, for purposes required by the communication protocol, the KI often needs an envelope for the packet. This is certainly a common requirement for all network connections. In such cases, the KI can easily maintain a pool of envelopes. When pSOS+m requests a packet buffer, the KI allocates an envelope, and returns to pSOS+m a pointer to the packet that is contained inside the envelope. pSOS+m does not need to know about the envelope.

pSOS+m uses the **ki_getpkb** and **ki_retpkb** services to allocate and return packet buffers, respectively. The number of such packet buffers necessary is dependent on the implementation and hardware requirements of the KI.

Packet Buffer Sizes

When requesting a packet buffer, pSOS+m passes the KI the length of the packet to be sent so that the KI can allocate a packet buffer of the appropriate size. With two exceptions, all packets sent through the KI take no more than 100 bytes. These exceptions are as follows:

1. Systems with **mc_nnode** > 576 nodes. In such systems, whenever a node joins, the master node will request a packet buffer of size

   28 + ceil(mc_nnode / 32) * 4

   where **ceil** is the ceiling function.

   For example, if the system has 900 nodes, then a packet buffer containing 144 bytes will be required whenever a node joins.

2. Systems that transmit variable length messages larger than 28 bytes. Whenever such a message is sent or requested, pSOS+m will request a packet buffer of size

   72 + message size

   For example, if a **q_vreceive** call is made with **buf_len** equal to 128, then pSOS+m will request a 200-byte packet buffer from the KI.

If neither of the above exceptions applies to a system, then the KI can ignore the packet size parameter and simply provide fixed-size 100-byte packet buffers. This is the simplest implementation. However, if the characteristics of a system require that the KI provide packet buffers of widely varying sizes, then a more sophisticated KI implementation may be required.

For example, if it is known that a node send/receives messages of lengths 256 and 512, then the KI could create three pools of buffers having sizes 100, 328 and 584. When **ki_getpkb** is called, the KI can allocate the buffer from the appropriate buffer pool based on the required size.

The Multiprocessor Configuration Table entry **mc_kimaxbuf** specifies the maximum buffer size that the KI is capable of allocating. It must be the same on every node, and a slave node will not be allowed to join if its **mc_kimaxbuf** is different from that of the master node. **mc_kimaxbuf** is used by pSOS+m in two ways:

1. During startup, pSOS+m verifies the **mc_kimaxbuf** is at least 100 and also large enough based on the value **mc_nnode**. If not, a fatal startup error occurs.
2. Any attempt to create a *global* variable length message queue will fail if **mc_kimaxbuf** is too small to accommodate the largest message that might be sent to the queue.

pSOS+m also provides the KI with the packet size when calling **ki_send**. However, do not confuse packet size with packet buffer size. For example, pSOS+m may request a packet buffer for a packet of size 80. The KI may allocate a packet buffer of size 256. Subsequently, pSOS+m calls **ki_send** to send the packet. At this time, pSOS+m will pass a packet size of 80, not 256. If the KI has multiple packet buffer pools, then certain KI services, most notably **ki_retpkb** will need to know the packet buffer size of a packet provided by pSOS+m. This is best accomplished by embedding the packet buffer size in the packet envelope.

Packet Transmission

pSOS+m calls the KI to send a packet as a result of numerous system activities. To prepare and send a packet, pSOS+m does the following:

- It uses **ki_getpkb** to obtain a packet buffer.
- It constructs and stores the packet in the packet buffer.
- It calls **ki_send** to send the packet to the destination node. This call has the responsibility of returning the packet buffer to the KI.

The KI on the source node must deliver the packet to the KI on the target node. The target node's KI is then responsible for delivering the packet to pSOS+m on that node.

Packet Reception

On most systems, the arrival of a packet at a node triggers an interrupt. In this case, the following actions occur on the receiving node:

1. The interrupt vectors control to a Packet ISR, which should be part of the KI, and the Packer ISR receives the packet into a packet buffer.
2. The ISR calls the pSOS+m **Announce_Packet** entry (see page 2-24) to inform pSOS+m that one or more packets are pending in the KI.
3. The ISR exits using the pSOS+m **i_return** system call.

4. At the next dispatch (normally part of **i_return**), pSOS+m calls **ki_receive** to obtain the received packet.
5. pSOS+m processes the packet.
6. What happens from this point is dependent on the packet.

Since several packets may arrive nearly simultaneously at a single node, the KI may have to maintain an inbound packet queue. If implemented, this queue must preserve the order of the packets received. Since several packets may be in the queue after Step 6 above, pSOS+m actually returns to Step 4. If the queue is empty, **ki_receive** returns a NULL pointer and pSOS+m terminates packet processing.

If hardware or other limitations make it impossible or impractical for an incoming packet to generate an interrupt, then a node must periodically poll for packets that have arrived. This is normally accomplished from the real-time clock/timer ISR. The ISR simply calls a routine (which is normally part of the KI) to check for arrived packets and processes it as described in Steps 1 and 2 above.

Note that while a polled KI does not affect the features available with pSOS+m, it does significantly affect the transmission time, since in the worst case, an entire clock tick may elapse before the packet is delivered to pSOS+m.

The pSOS+m Announce_Packet Entry

When a packet arrives at a node, the KI must inform pSOS+m by calling the special pSOS+m **Announce_Packet** entry. The address of this entry point is passed by pSOS+m as input when it calls **ki_init**.

The KI must call **Announce_Packet** with a JSR instruction from supervisor mode. This pSOS+m subroutine neither accepts nor returns any parameters, and it preserves all the caller's registers.

NOTE: Announce_Packet must be called only from an ISR. If an inbound packet causes an interrupt at the node, it is natural to call it from the packet ISR. On the other hand, if a node must poll for incoming packets, then this polling should be done, and Announce_Packet called, from the node's real-time clock/timer ISR.

Transmission Requirements

pSOS+m assumes the KI implementation supports:

- **RELIABLE TRANSMISSION** -- The KI must be responsible for delivery of packets. Failure detection, retransmission (if necessary) and reporting must be done in the KI. Rule No.1: Packets must be delivered correctly to the destination node or an error code must be returned to pSOS+m.
- **ORDER PRESERVATION** -- Between any two nodes, packets must be received in exactly the order in which they are sent. However, packets destined for different

nodes may be sent or received out of temporal order. Rule No. 2: Between any node pair, packet order must be strictly preserved.

- **NO DUPLICATION** -- pSOS+m cannot handle duplicates of the same packet. Rule No. 3: Packets must be delivered without duplicates.

Aside from the above requirements, pSOS+m does not impose any restrictions regarding routing, protocol, or any other implementation dependent KI behavior.

KI Error Conditions

Every KI service call must return an error code to pSOS+m. A value of 0 indicates the call completed successfully. Any other value indicates an error occurred. No specific KI error codes are defined since they are highly implementation dependent. However, pSOS+ reserves error codes 0x10000 and above for user-generated errors, including KI errors. No ISI product generates a code in this range.

Although supported by pSOS+m, most multiprocessor applications do not anticipate and hence will not tolerate node failure. In these cases, the best KI implementation is to always return 0. In the event of any error condition, the KI should simply call **k_fatal()** and trigger a system abort. This simple implementation has the advantage that the application does not need to manage errors resulting from low-level KI failures, which will be, at best, difficult to recover.

Systems that tolerate node failure will need to use KI error codes, since the KI services **ki_getpkb** and **ki_send** may fail if the destination node has failed. In these cases, the KI may first take corrective action such as aborting either the source or destination node via **k_fatal()** or **k_terminate()**, and then, if **k_fatal()** was not called, return an error code to pSOS+m. pSOS+m then takes further actions based on the identity of the source and destination nodes and type of packet that it was trying to deliver.

The following rules summarize the behavior of pSOS+m when **ki_getpkb** or **ki_send** return an error. There are three cases to consider:

Slave to Master — If a slave node cannot send a packet to the master node, then pSOS+m on the slave node shuts down operation of the slave node. The ability to communicate with the master node is essential to slave node operation.

Master to Slave — If the master node cannot send a packet to a slave node, pSOS+m on the master node internally invokes **k_terminate()** to terminate the slave node. Again, communication between the master and slave is essential to proper slave node operation. In addition, if the packet was an RSC, then pSOS+m on the master node will return the KI error code to the calling task.

Slave to Slave — The only packets that are passed between two slave nodes are RSC, RSC reply and asynchronous RSC error notification packets (see

pSOSystem System Concepts). If an RSC packet cannot be delivered, the RSC call is aborted and the error code returned by the KI is passed back to the calling task. If an RSC reply or asynchronous RSC error notification packet cannot be delivered, then pSOS+m on the source node internally invokes **k_terminate()** to abort the destination node.

If **ki_init**, **ki_retpkb**, **ki_receive**, **ki_time**, or **ki_roster** return an error, pSOS+m will simply shut down the node for lack of anything better to do. Even in systems that tolerate node failure, these KI calls should never return an error.

KI Conventions and Restrictions

pSOS+m calls KI services as subroutines. Parameters are passed in registers. The KI should perform the requested service and return to pSOS+m by a subroutine return, passing any output parameters in their assigned registers. The following calling conventions apply:

Entry: pSOS+m calls the KI with a JSR instruction. The address used will be that contained in the Multiprocessor Configuration Table entry **mc_kicode**. pSOS+m passes a function code in D0.L to select the specific KI service. Additional input parameters are passed in assigned registers.

Return: The KI should use an RTS to return to pSOS+. An error code must always be returned in D0.L. Zero indicates success. All other error codes must be > 0x10000. All registers except D0.L and those used to return output parameters must be restored. The KI is called by pSOS+m from the supervisor state. The interrupt level may be raised, but must not be lowered, provided that it is restored prior to returning to pSOS+.

The KI is logically an extension to pSOS+. It is not, and must not be confused with, a pSOS+m I/O driver. As such, there are critical restrictions regarding the pSOS+m system calls that can be made from KI. In general, KI may use any of the system calls allowed from an ISR. In addition, the KI can make a system call if the following are true:

1. That call does not generate a recursive request to the KI (e.g. an RSC). This is normally not a problem, since the pSOS+m system calls needed by the KI are unlikely to require remote service.

2. That call does not attempt to block. Recall that the KI executes as an extension to the kernel, not in the context of any particular task. Therefore, blocking is not possible. This is also not a serious limitation, since most KI implementations should have no need to block. If blocking is needed, then the KI should defer some of its operations to a server task, which of course can block.

Note carefully the following consequence of the first limitation. The KI can use a pSOS+m local-only (i.e. un-exported) partition to create its packet buffer pool and to allocate and deallocate packet buffers. This is sufficient for a network-connected system. Now consider a memory-bus-connected system. Whereas it may appear convenient and natural, to create a pSOS+m global partition and use it as the KI packet buffer pool, in practice this is difficult. The reason is that the **pt_create()** system call, if called from **ki_init** to create and export this partition, will recursively call the KI to deliver the partition information to the master node.

KI Services

ki_getpkb

ki_getpkb obtains a packet buffer from the KI. Its syntax is as follows:

INPUT:	D0 = 2
	D1 = Packet size in bytes
	D2 = Destination node number. 0 means packet will be broadcast.
OUTPUT:	D0 = 0, or KI-specific error code.
	A0 = Pointer to packet buffer

The packet size and destination node number is provided for KI implementations that need to allocate the buffer from different pools, based on either the node to which the packet will be sent or the size of the packet, or both. This might be the case, for example, in a shared memory implementation that writes the packet directly into the visible memory on the target node. Most KI implementations can likely ignore one or both parameters.

ki_init

ki_init initializes the node's KI. Its syntax is as follows:

INPUT:	D0 = 1
	A0 = Address of pSOS+ **Announce_Packet** Entry
OUTPUT:	D0 = 0, or KI-specific error code

ki_init is called during pSOS+ startup to initialize the KI. **ki_init** is only called once for each system startup. This service should do the following:

1. Initialize the communication hardware.
2. Initialize all KI data structures.

3. Create a pool of packet buffers. If enough buffers are not created, a system failure can result.
4. Save the pSOS+ **Announce_Packet** Entry address.

ki_init is called after all local pSOS+m facilities (including creation of the ROOT and IDLE tasks) have been initialized, and are thus usable. **ki_init** is subject to the same restrictions as all other KI services (see “KI Error Conditions” on page 2-25), except that

- It is always called with the interrupt mask at Level 7, and
- **ki_init** can drop the interrupt mask, provided that the necessary steps have been taken (for example, setting up ISRs) to handle any possible interrupt sources, and the mask is restored to level 7 before returning to pSOS+.

ki_receive

ki_receive obtains a received packet. Its syntax is as follows:

INPUT: D0 = 6

OUTPUT: D0 = 0, or KI specific error code.

A0 = Pointer to packet buffer. 0 means none.

pSOS+ calls **ki_receive** only after a call has been made by the KI to the special pSOS+ **Announce_Packet** Entry.

ki_retpkb

ki_retpkb returns a packet buffer to the KI. Its syntax is as follows:

INPUT: D0 = 3

A0 = Pointer to packet buffer.

OUTPUT: D0 = 0, or KI specific error code.

NOTE: pSOS+m does not provide the size of the packet buffer. If the KI needs this information, it should embed the size in the packet buffer envelope.

ki_roster

ki_roster provides roster information to the KI. Its syntax is as follows:

INPUT: D0 = 9

D1 = Type of roster change. (see below)

OUTPUT: D0 = 0, or KI specific error code.

D1 specifies the type of change as follows:

D1	Type of Change
0	Initial roster. A0 points to the internal pSOS+m roster.
1	A node has joined. D2 and D3 contain, respectively, the node number and sequence number of the new node.
2	A node has failed. D2, D3, and D4 contain, respectively, the node number of the failed node, the failure code, and node number of the node that initiated removal of the node from the system (which may be the failed node itself).

ki_send

ki_send sends a packet to another node. Its syntax is as follows:

INPUT: D0 = 4

D1 = Packet size. (in bytes)

D2 = Destination node number.

A0 = Pointer to the packet buffer.

OUTPUT: D0 = 0, or KI-specific error code.

ki_send must deliver the packet to the destination node. The packet size, specified in D1, is provided for systems which must transmit the packet over a relatively slow medium. In such cases, the KI can transmit only the packet, if it is much smaller than 100 bytes. Most kernel interfaces can likely ignore this parameter.

ki_send is responsible for returning the packet buffer after a successful transmission, or whenever it is no longer needed. Note, pSOS+m does not provide the size of the packet buffer. If the KI needs this information, it should embed the size in the packet buffer envelope.

ki_time

ki_time allows the KI to implement its own timing and timing-dependent operations, if necessary. Its syntax is as follows:

INPUT: D0 = 7

OUTPUT: D0 = 0, or KI-specific error code.

ki_time is called by pSOS+m at each clock tick to allow, if necessary, the KI to implement its own timing and timing-dependent operations, such as transmission retries.

If the KI does not need any timing operations, then **ki_time** should simply return.

NAME

DISI -- Device Independent Serial Interface

DESCRIPTION

The Device Independent Serial Interface (DISI) is the interface between the device-dependent and the device-independent parts of a serial driver. The DISI interface is used by pSOSystem Terminal, SLIP, PPP, and pROBE+ upper level drivers to interface with the chip dependent lower level driver.

The DISI separates the hardware dependent driver and the independent serial protocols. The DISI is the standard interface between the upper level hardware independent drivers to a low-level hardware-dependent driver. You would use this interface specification if you needed to write a serial driver for a serial chip that was not supported by pSOSystem. This specification will tell you what lower-level chip-dependent functions you need to write and the functionality they need. There is a template of a lower-level serial driver that you can start from. This template contains skeleton functions and some common code that can help you organize the chip dependent part of your driver. This template is called **disi.c** and is located in **drivers/serial**. There is an include file in the include directory called **disi.h** that contains definitions of the **#defines** and structures discussed in this specification.

You can also use this specification if you have a new protocol or custom serial needs that you wanted to add on top of a lower-level serial chip driver that is supported by pSOSystem. This specification will tell you what services are provided by those drivers. Figure 1 on page 2-32 illustrates the DISI interface.

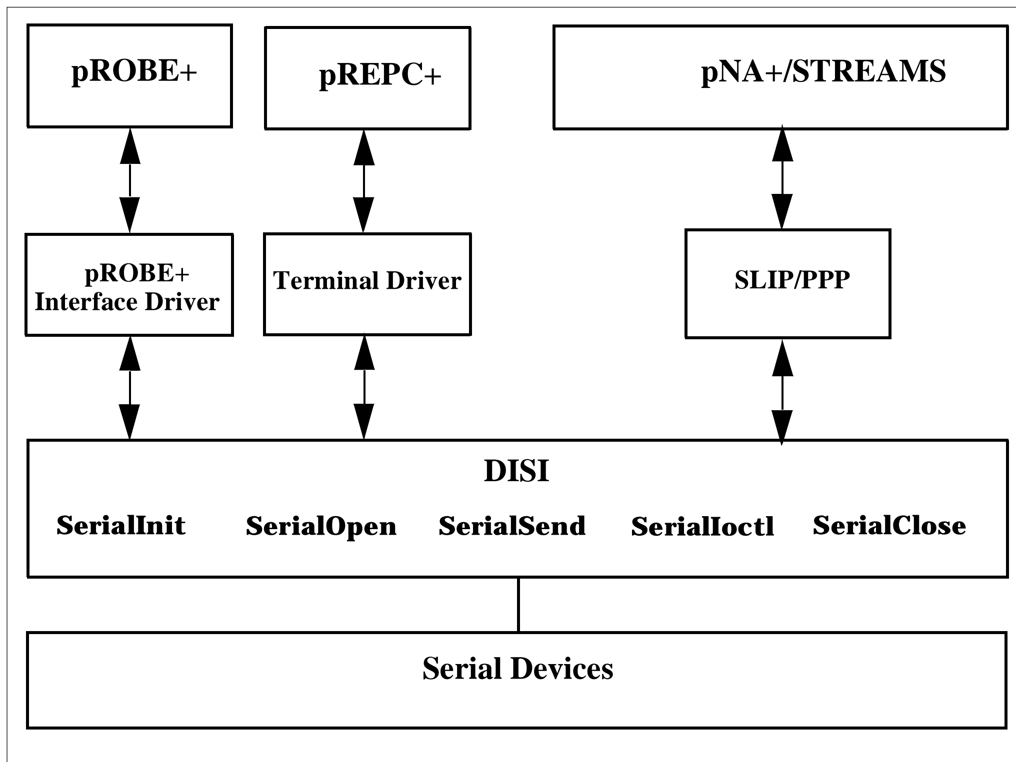

Figure 1 DISI Interface

The DISI interface consists of two parts:

1. Functions that must be provided by the lower-level hardware dependent device driver.
2. Callback functions that must be provided by the upper level hardware independent device driver.

Function Calls

The DISI function calls are called from the upper-level serial driver to:

- Initialize the interface.
- Initialize and open a serial channel.

- Send data.
- Issue control operation.
- Close down a serial channel.

The five functions that must be implemented in the device-dependent lower-level serial code are:

SerialInit	Initialize the driver.
SerialOpen	Open a channel.
SerialSend	Send data on the channel.
SerialIoctl	Perform a control operation on the channel.
SerialClose	Close the channel.

NOTE: All of these functions must be non-blocking asynchronous functions.

Callback Functions

The callback functions are supplied by one of the upper level drivers such as the pROBE+ interface driver, SLIP, PPP, and Terminal driver. The callback functions are called from the device-dependent lower-level serial driver to:

- Indicate data reception.
- Indicate exception condition.
- Confirm data sent.
- Confirm a control operation.
- Access memory services.

The seven callback functions that must be supported by the upper-level serial driver are:

UDataInd	Indicate reception of data.
UExpInd	Indicate an exception condition.
UDataCnf	Indicate completion of a **SerialSend** operation.
UCtlCnf	Indicate completion of a **SerialIoctl** operation.
UEsballoc	Attach external buffer to message block.
UAllocb	Allocate a message block triplet.
UFreemsg	Free a message block triplet list.

The addresses to these callback functions are passed to the lower-level serial code when the **SerialOpen** function is called.

Figure 2 below illustrates function calls and callbacks in the serial interface:

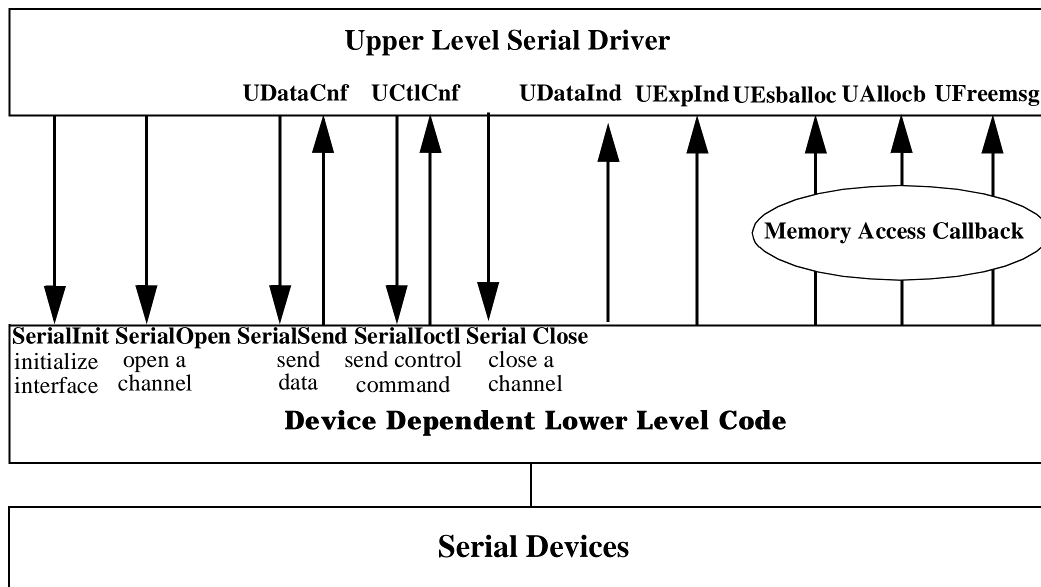

Figure 2 Function calls and callbacks in the Serial Interface

Data is transferred between the upper-level drivers and the DISI using the **SerialSend** call to send data out a channel and **UDataInd** call to receive data from a channel. Data is transferred using the Streams message block structure.

The DISI implements various features such as:

- Character mode asynchronous
- Block mode Asynchronous
- Flow control, special character detection

If a feature is not supported by a chip set, it should be emulated by software in the device-dependent lower-level code. For example, if software flow control is not a function of the chip set, then the lower-level code should emulate it.

DISI Functions

The following sections explain the functions that must be implemented in the device-dependent layer of the DISI.

SerialInit Function

The **SerialInit** function initializes the device-dependent lower-level code.

```
void SerialInit (void);
```

SerialInit is called before any components are initialized. It sets the driver to a default state with all channels closed, interrupts off, and all buffer pools empty.

SerialOpen Function

The **SerialOpen** function opens a channel for a particular mode of operation.

```
long SerialOpen(
    unsigned long channel,          input
    ChannelCfg *cfg,                input
    Lid *lid,                       output
    unsigned long *hdwflags)        output
    );
```

channel Indicates the serial channel to be opened.

cfg Points to the configuration table that defines various configuration parameters such as baud rate, various line parameters, and the addresses of the callback functions. See Data Structures for more details on the configuration table.

lid Set by the lower-level driver and is the lower level's reference ID for this channel. All calls to the DISI by the upper layer pass *lid* except for the **SerialInit** command.

hdwflags Not used for DISI.

EXAMPLE

The following example shows the use of a **SerialOpen** function call to open a channel.

```
/**********************************************************/
/* The Open function is an example of the use of the*/
/* SerialOpen function */
/*  */
/* It takes one argument the channel number to open.*/
/**********************************************************/
```

```
/*********************************************************/
/* The global array called lids will be used to store*/
/* the lower IDs*/
/*********************************************************/
unsigned long lids[NUMBER_OF_CHANNELS];

unsigned long Open(int channel)
{
ChannelCfg channelcfg;

/*********************************************************/
/* Set up configuration structure that will be passed*/
/* to DISI interface.*/
/*********************************************************/
/* Clear the ChannelCfg structure */
/*********************************************************/
bzero(&channelcfg, sizeof(ChannelCfg));

/*********************************************************/
/* Set Mode to UART mode*/
/*********************************************************/
channelcfg.Mode = SIOCASYNC;

/*********************************************************/
/* Set character size to 8 bits */
/*********************************************************/
channelcfg.Cfg.Uart.CharSize = SCS8;

/*********************************************************/
/* Set Flags for software flow control and to cause an*/
/* interrupt when a break is received.*/
/*********************************************************/
channelcfg.Cfg.Uart.Flags = SBRKINT | SWFC;
/*********************************************************/
/* Set Xon and Xoff characters to be used for software*/
/* flow control */
/*********************************************************/
channelcfg.Cfg.Uart.XOnCharacter = XON;
channelcfg.Cfg.Uart.XOffCharacter = XOFF;

/*********************************************************/
/* Set the len of transmit request to 4 so there can*/
/* be only 4 requests outstanding at one time*/
/*********************************************************/
```

```
channelcfg.OutQLen = 4;

/*******************************************************/
/* Set the channels baudrate.NOTE SysBaud is a global*/
/* variable defined by pSOSystem to the default baud rate*/
/*******************************************************/
channelcfg.Baud = SysBaud;

/*******************************************************/
/* Set the line mode to full duplex*/
/*******************************************************/
channelcfg.LineMode = FULLD;
/*******************************************************/
/* Set the pointers to the call back functions */
/*******************************************************/
channelcfg.dataind = term_dataind;
channelcfg.expind = term_expind;
channelcfg.datacnf = term_datacnf;
channelcfg.ctlcnf = term_ctlcnf;
channelcfg.allocb = gs_allocb;
channelcfg.freemsg = gs_freemsg;
channelcfg.esballoc = gs_esballoc;

/*******************************************************/
/* Set the ID to be used by the lower driver when*/
/* referencing this channel.       */
/*******************************************************/
channelcfg.uid = channel;

/*******************************************************/
/* Call the DISI interface open*/
/*******************************************************/
if(error = SerialOpen(channel, (ChannelCfg *)&channelcfg,
           (Lid )&lids[channel],
           (unsigned long *)&DChanCfg[minor].hdwflags))
    {
    /***************************************************/
    /* Return error code.*/
    /***************************************************/
    switch (error)
        {
        case SIOCAOPEN:
             /*******************************************/
             /* The Channel has already been opened by */
             /* another driver*/
             /*******************************************/
```

```
            return(1);

        case SIOCBADCHANNELNUM
            /********************************************/
            /* Channel is not a valid channel for this*/
            /* hardware*/
            /********************************************/
            return(2);

        case SIOCCFGNOTSUPPORTED
            /********************************************/
            /* Hardware cannot be configured by the*/
            /* DISI as given*/
            /********************************************/
            return(3);

        case SIOCBADBAUD:
            /********************************************/
            /* Baud rate not supported by hardware.*/
            /********************************************/
            return(4);

        case SIOCNOTINIT:
            /********************************************/
            /* This error shows that the lower driver*/
            /* thinks it has not been initialized.*/
            /********************************************/
            return(6);
       }
```

SerialSend Function

The **SerialSend** function is used by the upper level serial driver to transfer data to the lower-level driver.

```
long SerialSend(
   Lid lid,          input
   mblk_t* mbp       input
   );
```

lid The lower-level ID that was acquired during **SerialOpen** operation for the channel to which this is directed.

mbp A pointer to the message block that contains the data to be transmitted.

A 0 return code indicates that the message block has been queued to send. The **UDataCnf** callback will be used by the lower-level driver when the data in the message block has actually been sent.

> NOTE: If a SIOCOQFULL error is received, no data was sent because the transmit queue is full. SerialSend continues to return SIOCOQFULL until the next UDataCnf callback happens. Since UDataCnf is the confirmation of a message being sent, the transmit queue will no longer be full.

EXAMPLE

The following example shows the use of a **SerialSend** call to send data to the lower serial driver.

```
/*-----------------------------------------------------------*/
/* This is an example of a function that will get a mblock from*/
/* the mblock pool, fill the mblock's data buffer with some    */
/* information and send it to the lower serial driver.         */
/*-----------------------------------------------------------*/
#include <gsblk.h>
#include <disi.h>

static char test_string[] = "This is a Test Buffer";
/************************************************************/
/* SendData:  Gets a mblock, puts some data into it and sends  */
/*            it to the lower driver.                         */
/*                                                            */
/*              (Lid)lid lower level id gotten when the       */
/*               SerialOpen call was made.                    */
/*                                                            */
/*     RETURNS: 0 on success                                  */
/*              1 gs_allocb failure                           */
/*              2 SerialSend failure                          */
/*     NOTE(S):                                               */
/*                                                            */
/************************************************************/
int SendData((Lid)lid)
{
int i;

/************************************************************/
/* The typedefs frtn_t and mblk_t are found in pna.h.         */
/************************************************************/
mblk_t *m;

/************************************************************/
```

```
/* Call gs_allocb to get a buffer attached to a mblock         */
/* structure.                                                  */
/*                                                             */
/* gs_allocb is a function supplied by pSOSystem in the file   */
/* drivers/gsblk.c. It is compiled into bsp.lib.               */
/* gs_allocb takes two arguments                               */
/*          size: size of message block to be allocated        */
/*          pri: allocation priority (LO, MED, HI)             */
/*                                                             */
/* gs_allocb is a utility that allocates a message block of    */
/* type M_DATA and a buffer of a size greater than or equal to */
/* specified size. pri indicates the priority of the allocation*/
/* request. Currently pri is not used and should be set to 0   */
/* On success, gs_allocb returns a pointer to the allocated    */
/* message block. gs_allocb returns a NULL pointer if it could */
/* not fill the request                                        */
/*                                                             */
/* mblk_t *gs_allocb( int size, int pri)                       */
/*                                                             */
/* A mblk_t structure looks like this:                         */
/*                                                             */
/*   struct msgb                                               */
/*       {                                                     */
/*       struct msgb    *b_next;   next msg on queue           */
/*       struct msgb    *b_prev;   previous msg on queue       */
/*       struct msgb    *b_cont;   next msg block of msg       */
/*       unsigned char  *b_rptr;   first unread data byte in   */
/*                                 buffer                      */
/*       unsigned char  *b_wptr;   first unwritten data byte   */
/*                                 in buffer                   */
/*       struct datab   *b_datap;  data block                  */
/*      }                                                      */
/***************************************************************/
if(m = gs_allocb(sizeof(test_string), 0) == 0)
    return(1);

/***************************************************************/
/* Copy data to buffer                                         */
/***************************************************************/
for (i = 0; i < sizeof(test_string); i++, m->b_wptr++)
    *(m->b_wptr) = test_string[i];

/***************************************************************/
/* Send mblock to lower driver                                 */
/***************************************************************/
if(SerialSend(lid, m) != 0)
```

```
    return(2);
else
    return(0);
}
```

SerialIoctl Function

The **SerialIoctl** function specifies various control operations that modify the behavior of the DISI.

```
long SerialIoctl(
      Lid lid,            input
      unsigned long cmd,  input
      void *arg           input
      )
```

lid The lower-level ID that is acquired during a **SerialOpen** operation.

cmd The type of control operation.

arg Specific information for the operation.

In some cases, a **SerialIoctl** operation may not complete immediately. In those cases, the **UCtlCnf** function is called when the operation has completed with the final status of the command.

SerialIoctl Commands

The SerialIoctl commands are:

SIOCPOLL — Polls the serial device for asynchronous events such as data indication and exception indication. It provides an ability to perform as a pseudo ISR and call the callback functions when the channel is in SIOCPOLL mode or when interrupts are disabled. For example, when pROBE+ is in control, the processor operates with interrupts turned off. This command checks for data received, data transmitted, or exceptions and then triggers the callback function for these conditions, as needed.

SIOCGETA — Gets the channel configuration and stores this information into a **ChannelCfg** structure pointed to by the **arg** parameter. This command is immediate, so no callback is made.

SIOCPUTA — Sets the channel configuration using the information stored in a **ChannelCfg** structure pointed to by the

	arg parameter. The effect is immediate, so no callback is made.
SIOCSACTIVATE	Activates the channel. This enables the receiver and transmitter of the channel and waits until the channel becomes active. In dial-in connections, the SIOCSACTIVATE command puts the hardware in a mode capable of handling an incoming call. The **UCtlCnf** callback is made when the call arrives. When using HDLC (even when no dial-up connection is involved), the **UCtlCnf** callback is made when the link is active, i.e., it starts receiving flags.
SIOCBREAKCHK	This command will check to see if a break character has been sent. This command is used by pROBE+ to see if the user wants to enter pROBE+. The **arg** parameter is set to SIOCBREAKR if there has been a break sent to the channel.
SIOCPROBEENTRY	This command tells the driver that pROBE+ is being entered. The driver should now switch to the debugger callouts, **uid** and switch from interrupt mode to polled mode.
SIOCPROBEEXIT	This commands tell the driver the pROBE+ is being exited and the driver should now switch from the debugger callouts to the normal callouts, normal **uid** and allow interrupts. Normal callouts and **uid** are the ones from a **SerialOpen** call. If pROBE+ is the only user of the channel then the normal callouts and **uid** and the debugger callouts and **uid** will be the same.
SIOCMQRY	Gets information about which modem controls are supported by the channel and stores this information into the **long int** pointed to by the **arg** parameter. A set bit indicates that the particular control line is supported by the channel. This command is immediate, so no callback is made.

The modem control lines are:

SIOCMDTR	Data terminal ready.
SIOCMRTS	Request to send.
SIOCMCTS	Clear to send.
SIOCMDCD	Data carrier detect.
SIOCMRI	Ring indicator.

	SIOCMDSR Data set ready.
	SIOCMCLK Clock (sync support).
	Since the interface is a DTE, DTR and RTS are outputs and CTS, RI, DSR, and DCD are inputs.
SIOCMGET	Gets the current state of the modem control lines and stores this information into the **long int** pointed to by the **arg** parameter. SIOCMGET uses the same encoding as SIOCMQRY. Bits pertaining to control lines not supported by the channel and the SIOCMCLK bit are cleared. This command is immediate, so no callback is made.
SIOCMPUT	Sets the modem controls of the channel. The **arg** parameter is a pointer to a **long int** containing a new set of modem control lines. The modem control lines are turned on or off, depending on whether their respective bits are set or clear. SIOCMPUT uses the same encoding as SIOCMQRY. Bits pertaining to control lines not supported by the channel and the SIOCMCLK bit have no effect. The effect is immediate, so no callback is made.
SIOCRXSTOP	Stops the flow of receive characters. This is used when the upper level serial driver needs to stop the flow of characters it is receiving. The lower-level serial code takes the correct action such as sending an **XOFF** character if software flow control is being used or changing the hardware lines if hardware flow control is being used. The effect is immediate so no call back is made.
SIOCRXSTART	Indicates that the upper level serial driver wants to continue to receive characters. The lower-level serial code will take the correct action such as sending an **XON** character if software flow control is being used or changing the hardware lines if hardware flow control is being used. The effect is immediate so no call back is made.
SIOCNUMBER	Gets the total number of serial channels and stores this information into the **long int** pointed to by the **arg** parameter. This command is immediate, so no call back is made

EXAMPLE

The following example shows the use of a **SerialIoctl** function call to get the baud rate of the channel.

```
/*********************************************************/
/* This get_number_of _ports function is an example of a */
/* SerialIoctl function call.                            */
/*********************************************************/
int get_number_of_ports(unsigned long number)
{
/*********************************************************/
/* Assume the lower level ID is stored by the SerialOpen */
/* call in a global array called lids. Use the           */
/* SIOCNUMBER I/O control command to get the total       */
/* number of serial channels and number as a place to    */
/* store that number.                                    */
/*********************************************************/
if(SerialIoctl(lids[channel], SIOCNUMBER, (void *)&number)
   return(-1);
else
   return(number);
}
```

SerialClose Function

The **SerialClose** function terminates a connection on a serial channel and returns the channel to its default state.

```
long SerialClose(
        Lid lid    input
        )
```

lid The lower-level ID that was acquired during **SerialOpen** operation for the channel that is to be closed.

If the channel is not open, SIOCNTOPEN is returned.

EXAMPLE

The following example shows a **SerialClose** function call to close the channel.

```
/***************************************************************/
/* This function TermClose is an example of a SerialClose call */
/* SerialClose will close the channel. This will flush all     */
/* transmit buffers, discard all pending receive buffers and   */
/* disable the receiver and transmitter of the channel. All    */
/* rbuffers associated with the channel will be released       */
/* (freed) and the device will hang up the line                */
```

```
/*                                                              */
/**************************************************************/

void TermClose (channel)
{

SerialClose((Lid)lids[channel]);

/*All semaphores and queues for the channel should be deleted here.*/
}
```

User Callback Functions

This section contains the templates of the callback functions that must be provided by the upper-level driver. Pointers to these functions are passed in the **ChannelCfg** structure during the **SerialOpen** of the channel to the device-dependent lower-level code. These pointers can be changed via the **SerialIoctl** command **SIOPUTA**.

NOTE: These calls must be callable from an interrupt. Consequently, it is important that they do not block within the call and only call OS functions that are callable from an ISR.

UDataInd Callback Function

The **UdataInd** callback function will be called during an interrupt by the device-dependent lower-level code to indicate reception of data to the upper level serial driver.

```
static void UDataInd(
               Uid uid,            input
               mblk_t * mbp,       input
               unsigned long b_flagsinput
               );
```

uid The upper-level serial driver's ID for the associated channel. The ID is passed to the lower-lever serial driver during the **SerialOpen** of the channel on which the data is arriving.

mbp A pointer to message block that contains the data received by the channel.

b_flags The status flags associated with this message block. The flags can be:

SIOCOKX	Received with out error.
SIOCMARK	Idle Line Condition.
SIOCBREAKR	Break Received.
SIOCPARITY	Parity Error.

SIOCOVERRUN Overrun of buffers.

SIOCCDLOST Carrier Detect Lost.

UDataInd must unblock any task that is waiting for data from this channel.

NOTE: If the SerialOpen call returned hdwflags that had the SIOCHDWRXPOOL bit set, then the lower-level code has a receive buffer pool. This pool will need replenishing through the use of a call to SerialIoctl with the command SIOCREPLENISH.

The user supplied functions in the upper layer serial driver must use **SerialIoctl** to replenish the buffers. The upper level serial driver must free the message block (pointed to by **mbp**) when it is emptied by calling **UFreemsg**.

EXAMPLE

The following example shows a **UDataInd** function call to send data and status to a task.

```
/****************************************************************/
/* This function term_dataind is an example of a UDataInd      */
/* function. It will get as input:                              */
/*                                                              */
/*              Uid uid pointer to channels configuration      */
/*              mblk_t mblk message block containing data      */
/*              unsigned long b_flags condition code for block */
/*                                                              */
/* term_dataind will use a message queue to send the mblock    */
/* and status on to a task that is waiting for data.           */
/*                                                              */
/* Assume receive_ques is an array of message queue IDs.       */
/****************************************************************/
static void term_dataind(Uid uid, mblk_t *mblk, unsigned long b_flags)
{
/****************************************************************/
/* Set up the message buffer with the pointer to the mblock    */
/* and status                                                   */
/****************************************************************/
msg_buf[0] = (unsigned long)mblk;
msg_buf[1] = b_flags;
/****************************************************************/
/* Send message to channels message queue.                     */
/****************************************************************/
q_send(receive_ques[(unsigned long)*uid], msg_buf);
}
```

UExpInd Callback Function

The **UExpInd** callback function is called by the device-dependent lower-level code to indicate an exception condition.

```
static void UExpInd(
   Uid uid,                 input
   unsigned long exp        input
   );
```

uid The upper-level serial driver's ID for the associated channel which is passed to the lower-lever serial driver during the **SerialOpen** of the channel on which the exception has occurred.

exp Type of exception.

Exceptions can be one of the following:

SIOCMARK	Idle Line Condition.
SIOCBREAKR	Break Received.
SIOCFRAMING	Framing Error.
SIOCPARITY	Parity Error.
SIOCOVERRUN	Overrun of buffers.
SIOCCDLOST	Carrier Detect Lost.
SIOCCTSLOST	Clear To Send has been lost.
SIOCCTS	Clear To Send found.
SIOCCD	Carrier Detect detected.
SIOCFLAGS	Non Idle Line Condition.

UDataCnf Callback Function

The **UDataCnf** callback function is called by the device-dependent lower-level code to confirm that the data sent using **SerialSend** call has been transmitted.

```
static void UDataCnf(
   Uid uid,                 input
   mblk_t * mbp,            input
   unsigned long b_flags    input
   );
```

uid The upper-level serial driver's ID for the associated channel which is passed to the lower-lever serial driver during the **SerialOpen** of the channel on which the data was sent.

mbp Points to the message block sent using **SerialSend** call.

b_flags Status flags associated with the message block. The b_flags must be one of the following:

SIOCOK	Completed without error
SIOCUNDERR	Tx underrun (HDLC)
SIOCABORT	Tx aborted

The **UDataCnf** function must unblock any task that was waiting for data to be sent. The task is responsible for any maintenance necessary to the message block such as freeing it or reusing it.

EXAMPLE

The following example shows a **UDataCnf** function call to confirm that data has been sent.

```
/***********************************************************/
/* This function term_datacnf is an example of a UDataCnf      */
/* function. It takes as inputs:                                */
/*                                                              */
/*             Uid uid pointer to channels number               */
/*             mblk_t mblk message block containing data        */
/*             unsigned long b_flags condition code for block */
/*                                                              */
/* This code assumes that the driver is not waiting for        */
/* completion of a transmission.                                */
/***********************************************************/
static void term_datacnf(Uid uid, mblk_t *mblk, unsigned long b_flags)
{
gs_freemsg(mblk);
}
```

UCtlCnf Callback Function

The **UCtlCnf** callback function is used to confirm the completion of a **SerialIoctl** control command.

```
static void UCtlCnf(
              Uid uid,            input
              unsigned long cmd   input
              );
```

uid The upper-level serial driver's ID for the associated channel which is passed to the lower-lever serial driver during the **SerialOpen** of the channel on which the I/O control call was made.

cmd The command being confirmed.

EXAMPLE

The following example shows a **UCtlCnf** function call to confirm the completion of a **SerialIoctl** control command.

```
/*********************************************************/
/* static void term_ctlcnf                               */
/*                                                       */
/* This function term_ctlcnf is an example of a UCtlCnf  */
/* function. It takes as inputs:                         */
/*                                                       */
/*             Uid uid pointer to a configuration        */
/*             unsigned long cmd I/O control cmd that    */
/*             is being confirmed.                       */
/*                                                       */
/* term_ctlcnf assumes that a task is waiting for a      */
/* semaphore.                                            */
/* semaphore_ctl_ids is an array that stores the ID for  */
/* each channel                                          */
/*********************************************************/
void term_ctlcnf(Uid uid, unsigned long cmd)
{

/*-------------------------------------------------------*/
/* Release the channels I/O Control semaphore            */
/*-------------------------------------------------------*/
sm_v(semaphore_ctl_ids[(unsigned long)*uid]);
}
```

Access Memory Services

The following callback functions are used to manage message blocks and a buffer pool. The message blocks are similar to those used by Streams I/O. See the **pna.h** file in the **include** directory of the pSOSystem release for a definition of the message block structures used here. All of these functions are provided with the pSOSystem software. They are found in the file **drivers/gsblk.c**.

UEsballoc Callback Function

The **UEsballoc** callback function returns a message block triplet by attaching the user supplied buffer as a data block to a message block structure. See the **SendFrame** example under the **SerialSend** function for an example of this call.

```
 static mblk_t * UEsballoc(
                  char *bp,         input
                  long len,         input
                  long pri,         input
```

```
    frtn_t *frtn      input
    );
```

bp Points to the use-supplied buffer.

len Specifies the number of bytes in the buffer.

pri Specifies the priority for message block allocation.

frtn Pointer to the free structure of type **frtn_t.** This structure is as follows:

```
typedef struct
    {
    void (*free_func)();
    void *free_arg;
    } frtn_t
```

free_func **UFreemsg** calls the function pointed to by **free_func** when the caller-supplied buffer needs to be freed. The caller must supply the function pointed to by **free_func**.

free_arg A pointer to the user supplied buffer.

frtn_t The pointer to **frtn_t** must be stored by the **UEsballoc** call. This makes it available to the **UFreemsg** call when **UFreemsg** is used to free the message block.

The **UEsballoc** call may be used by the upper or the lower levels of the interface. In either case the "user" is who ever is making the call. One use of **UEsballoc** is a case where there is a special ram area to be used by the serial chip.

NOTE: This function corresponds to the gs_esballoc function supplied by pSOSystem in the file drivers/gsblk.c. It is compiled into bsp.lib. You may use a pointer to gs_esballoc for the UEsballoc callback function.

UAllocb Callback Function

The **UAllocb** callback function returns a message block triplet or a NULL if no buffer or message block could be found. See the *SendData* example under the **SerialSend** function for an example of this call.

```
static mblk_t * UAllocb(
        long size,    input
        long pri      input
        );
```

size Specifies the size of the buffer.

pri Specifies the priority for message block.

NOTE: This function corresponds to the gs_allocb function supplied by pSOSystem in the file drivers/gsblk.c. It is compiled into bsp.lib. You may use a pointer to gs_allocb for the UAllocb callback function.

UFreemsg Callback Function

The **UFreemsg** callback function is used to free a message block. See the *term_ctlcnf* example under the **UDataCnf** function for an example of this call.

```
static void UFreemsg(
        mblk_t *mbp,        input.
        );
```

mbp Points to the message block triplet for this specific message block pool. If the message block was formed using the **UEsballoc** call, **UFreemsg** calls the function pointed by **free_func** with a pointer to **free_arg** as its argument.

NOTE: This function corresponds to the gs_freemsg function supplied by pSOSystem in the file drivers/gsblk.c. It is compiled into bsp.lib. You may use a pointer to gs_freemsg for the UFreemsg callback function.

Data Structures

Following are templates of data structures. They can be found in **include/disi.h.**

CCfg

```
typedef struct ccfg {
   unsigned long  Mode;
   Modecfg        Cfg;
   unsigned long  NRBuffs;
   unsigned long  RBuffSize;
   unsigned long  OutQLen;
   unsigned long  Baud;
   unsigned long  LineMode;
   void           (*dataind)(uid,mblk_t, unsigned long);
   void           (*expind)(uid, unsigned long);
   void           (*datacnf)(uid,mblk_t,unsigned long);
   void           (*ctlcnf)(uid, unsigned long);
   mblk_t         * (*allocb)(long, long);
   void           (*freemsg)(mblk_t);
   mblk_t         * (*esballoc)(char,long, long,frtn_t);
   Uid            uid;
   unsigned long  Reserve[4];
   } ChannelCfg;
```

Mode **Mode** can be:

SIOCASYNC	Asynchronous mode MUST BE SET
SIOCPOLLED	Poll mode - interrupt if not set
SIOCLOOPBACK	Local loop back mode
SIOCPROBEMODE	pROBE+ mode

SIOCPROBEMODE is used to tell the lower driver that it should save the call back function pointers and the uid to be used for the I/O control SIOCPROBEENTRY.

NRBuffs The number of receive buffers to allocate for the receive queue.

RBuffSize NOT USED

OutQLen The maximum number of message buffers waiting to be transmitted. If the maximum number is exceeded, **SerialSend** fails with an **SIOCOQFULL** error.

Baud Set to the actual desired baud rate. If the selected baud rate is not supported by the lower-level device-dependent code, **SerialOpen** or **SerialIoctl** fails, an error is returned.

LineMode **Line mode** can be:

HALFD	Half-Duplex
FULLD	Full-Duplex

dataind Pointer to a data indication routine. See **UDataInd** for additional information.

expind Pointer to an exception indication routine. See **UExpInd** for additional information.

datacnf Pointer to a data confirmation routine. See **UDataCnf** for additional information.

ctlcnf Pointer to a control confirmation routine. See **UCtlCnf** for additional information.

allocb Pointer to an allocate message block routine. See **UAllocb** for additional information.

freemsg Pointer to a free message list routine. See **UFreemsg** for additional information.

esballoc Pointer to an attach message block routine. See **UEsballoc** for additional information.

UartCfg

```
struct UartCfg{
   unsigned long  CharSize;
   unsigned long  Flags;
   LineD          Lined[2];
   unsigned char  XOnCharacter;
   unsigned char  XOffCharacter;
   unsigned short MinChar;
   unsigned long  MaxTime;
   unsigned long  ParityErrs;
   unsigned long  FramingErrs;
   unsigned long  OverrunErrs;
   unsigned long  Reserve[4];
   }
```

CharSize

CharSize can be:

CS5	5 bits per character
CS6	6 bits per character
CS7	7 bits per character
CS8	8 bits per character

Flags

Flags can be:

C2STOPB	Send two stop bits, else one
PARENB	Parity enable
PARODD	Odd parity, else even
HWFC	Hardware flow control on
SWFC	Software flow control on
SWDCD	Software data carrier detect
LECHO	Enable local echo
BRKINT	Interrupt on reception of Break
DCDINT	Interrupt on loss of DCD

When PARENB is set, parity generation and detection is enabled and a parity bit is added to each character. When parity is enabled, odd parity is used if the PARODD flag is set, otherwise even parity is used.

When HWFC is set, the channel uses CTS/RTS flow control. If the channel does not support hardware flow control, this bit is ignored.

When SWFC bit is set, **XON/XOFF** flow control is enabled.

When SWDCD is set, the channel responds as if the hardware data carrier detect (DCD) signal is always asserted. If SWDCD is not set, the channel is enabled and disabled by DCD.

When BRKINT is set, the channel issues an **UExpInd** exception callback function if a break character is received.

When DCDINT is set, the channel issues an **UExpInd** exception callback function upon loss of the DCD signal.

LineD Not used for DISI.

XOnCharacter Software flow control character used to resume data transfer.

XOffCharacter Software flow control character used to temporarily terminate data transfer.

ParityErrs Keeps track of the parity errors that happen on the channel. This information is used by MIB.

FramingErrs Keeps track of the framing errors that happen on the channel. This information is used by MIB.

OverrunErrs Keeps track of the overrun errors that happen on the channel. This information is used by MIB.

Error Codes

The following error codes can be returned:

SIOCAOPEN	Channel already open.
SIOCBADCHANNELNUM	Channel does not exist.
SIOCCFGNOTSUPPORTED	Configuration not supported.
SIOCNOTOPEN	Channel not open.
SIOCINVALID	Command not valid.
SIOCBADARG	Argument not valid.
SIOCOPERATIONNOTSUP	Operation not supported.
SIOCOQFULL	Output queue full, send failed.
SIOCBADBAUD	Baud rate not supported.
SIOCWAITING	Waiting for previous command to complete.
SIOCNOTINIT	Driver not initialized.

Multiplex Driver Mapping

This describes how the lower chip level serial drivers for different chip types are to be multiplexed under the DISI so they can be used by one upper level driver. This is used when there is more then one type of serial chip. The method described here uses a private table under the **bsp**. The mapping tells the DISI what lower chip specific driver to call and the port number in that driver to use. This table would be set up in **board.c**.

SDRVCNFG Structure

There is one SDRVCNFG for each serial channel in the system. This is an array of structures that maps a channel number to a serial driver and physical port number in that driver. The structure is defined in **include/disi.h** as:

```
typedef struct
{
unsigned long dnum;       /* Driver number */
unsigned long pnum;       /* Physical port number */
Lid   lid;                /* Lower driver port ID */
}SDRVCNFG
```

SERIALFUNCS Structure

In order to multiplex more then one driver the names of the function call entry points for the lower-level driver code need to be different for each driver. For example, instead of **SerialOpen** function name for the port open function you might use ser_360_Open in a driver for the MC68360 chip. Each lower-level serial driver will have an entry in an array of structures called **SerialFuncs**. The **SerialFuncs** is an array of SERIALFUNCS structures. The SERIALFUNCS structure maps the DISI function calls with corresponding driver functions. The structure is defined in **include/disi.h** as follows:

```
typedef struct
{
long (*Init)(void);
long (*Open)(unsigned long, ChannelCfg *, Lid *, unsigned long) *;
long (*Send)(Lid, mblk_t *);
long (*Ioctl)(Lid, unsigned long, void *);
long (*Close)(Lid);
}SERIALFUNCS;
```

There is one SERIALFUNCS for each chip level driver.

The file **drivers/ser_mplx.c** contains multiplexed DISI driver entry functions. To used these functions requires a **#define int bsp.h** called BSP_NUM_SER_DRVRS. This **#define** should be set to the number of serial drivers that will be multiplexed. Also required to use these functions are two structures that should be declared in **board.c SerialFuncs** and **SDrvCnfg**. The following is an example of these two structures. These examples assumes there are two drivers to multiplex, a MC6836 serial driver called ser_360 and a Zilog 8530 driver that will be called Z8530:

```
/*              Driver            Driver #*/
        #define ser_360           0
        #define Z8530             1

SERIAL_FUNCS SerialFuncs[] =
/* INIT       OPEN             SEND           IOCTL        CLOSE       */
ser_360_Init,ser_360_Open,     ser_360_Send,ser_360_Ioctl,ser_360_Close,
Z8530_Init,  Z8530_Open,       Z8530_Send,  Z8530_Ioctl,  Z8530_Close;

SDRVCFNG SDrvCnfg[BSP_SERIAL + 1] =
/* Driver        Port              Lid  Channel */
0,                0,               0    /* Channel 0 - not used */
ser_360,          1,               0    /* Channel 1 */
ser_360,          2,               0    /* Channel 2 */
ser_360,          3,               0    /* Channel 3 */
Ser_360,          4,               0    /* Channel 4 */
Z8530,            1,               0    /* Channel 5 */
Z8530,            2;               0    /* Channel 6 */
```

Index zero of the array is not used because pSOSystem uses 0 for the default system console. The system console is mapped to an actual channel before being looked up in the **SDrvCnfg** array.

The **Lid** element of the **SDrvCnfg** will be set by the call to **SerialOpen** for each port.

In the above example, channel 4 corresponds to index 4 which is the 68360 driver port 4 and channel 6 corresponds to index 6 the Z8530 driver port 2.

NAME

DISIplus -- Device Independent Serial Interface

DESCRIPTION

DISIplus is an enhancement of the DISI interface. In addition to the features provided for by the DISI specification DISIplus adds several new I/O control calls and specifications for the use of HDLC. DISIplus is a super-set of DISI.

The Device Independent Serial Interface (DISI) is the interface between the device-dependent and the device-independent parts of a serial driver. The DISI interface is used by pSOSystem Terminal, SLIP, PPP and pROBE+ upper level drivers to interface with the chip dependent lower level driver. DISIplus adds the use of X.25 and synchronous PPP to the list of protocols used by the DISI specification.

The DISI separates the hardware dependent driver and the independent serial protocols. The DISI is the standard interface between the upper level hardware independent drivers to a low-level hardware-dependent driver. You would use this interface specification if you needed to write a serial driver for a serial chip that was not supported by pSOSystem. This specification will tell you what lower-level chip-dependent functions you need to write and the functionality they need. There is a template of a lower-level serial driver that you can start from. This template contains skeleton functions and some common code that can help you organize the chip dependent part of your driver. This template is called **disi.c** and is located in **drivers/serial**. There is an **include** file in the **include** directory called **disi.h** that contains definitions of the **#defines** and structures discussed in this specification.

You can also use this specification if you have a new protocol or custom serial needs that you wanted to add on top of a lower-level serial chip driver that is supported by pSOSystem. This specification will tell you what services are provided by those drivers. Figure 3 on page 2-58 illustrates the interface.

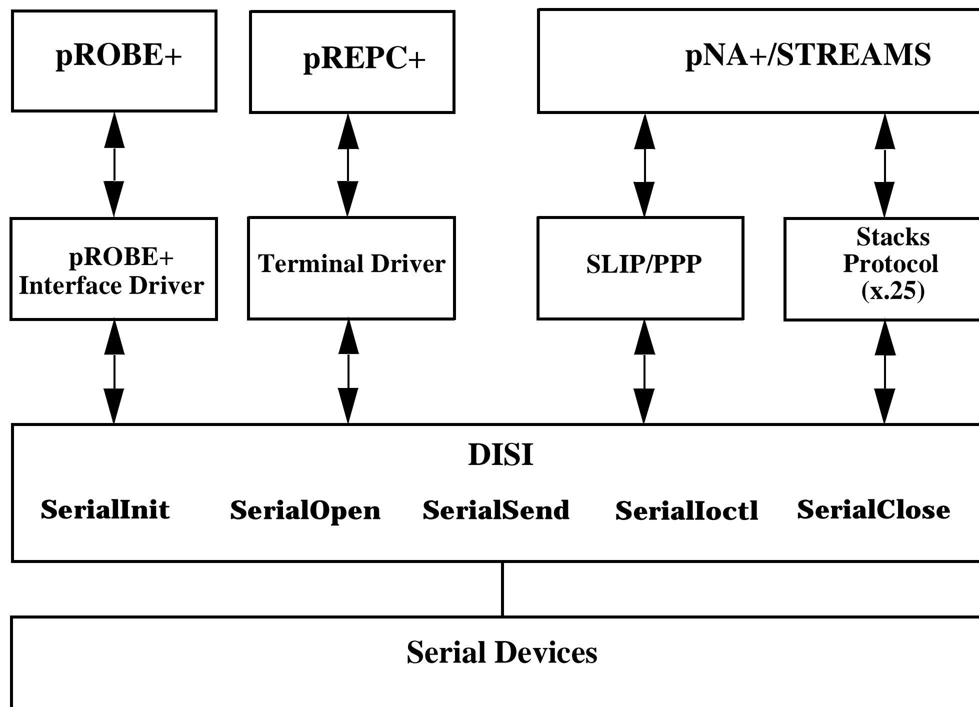

Figure 3 DISIplus Interface

The DISI interface consists of two parts:

1. Functions that must be provided by the lower-level hardware dependent device driver.
2. Callback functions that must be provided by the upper level hardware independent device driver.

Function Calls

The DISI function calls are called from the upper-level serial driver to:

- Initialize the interface.
- Initialize and open a serial channel.

- Send data.
- Issue control operation.
- Close down a serial channel.

The five functions that must be implemented in the device-dependent lower-level serial code are:

SerialInit	Initialize the driver.
SerialOpen	Open a channel.
SerialSend	Send data on the channel.
SerialIoctl	Perform a control operation on the channel.
SerialClose	Close the channel.

NOTE: All of these functions must be non-blocking asynchronous functions.

Callback Functions

The callback functions are supplied by one of the upper level drivers such as the pROBE+ interface driver, SLIP, PPP, and Terminal driver. The callback functions are called from the device-dependent lower-level serial driver to:

- Indicate data reception.
- Indicate exception condition.
- Confirm data sent.
- Confirm a control operation.
- Access memory services.

The seven callback functions that must be supported by the upper-level serial driver are:

UDataInd	Indicate reception of data.
UExpInd	Indicate an exception condition.
UDataCnf	Indicate completion of a **SerialSend** operation.
UCtlCnf	Indicate completion of a **SerialIoctl** operation.
UEsballoc	Attach external buffer to message block.
UAllocb	Allocate a message block triplet.
UFreemsg	Free a message block triplet list.

The addresses to these callback functions are passed to the lower-level serial code when the **SerialOpen** function is called.

Figure 2 on page 2-34 illustrates function calls and callbacks in the serial interface:

Data is transferred between the upper-level drivers and the DISI using the **SerialSend** call to send data out a channel and **UDataInd** call to receive data from a channel. Data is transferred using the Streams message block structure.

The DISI implements various features such as:

- Character mode asynchronous.
- Block mode Asynchronous and Block mode synchronous.
- Flow control, special character detection and protocol control.

If a feature is not supported by a chip set, it should be emulated by software in the device-dependent lower-level code. For example, if software flow control is not a function of the chip set, then the lower-level code should emulate it.

DISI Functions

The following sections explain the functions that must be implemented in the device-dependent layer of the DISI.

SerialInit Function

The **SerialInit** function initializes the device-dependent lower-level code.

```
void SerialInit (void);
```

SerialInit is called before any components are initialized. It sets the driver to a default state with all channels closed, interrupts off, and all buffer pools empty.

SerialOpen Function

The **SerialOpen** function opens a channel for a particular mode of operation.

```
long SerialOpen(
    unsigned long channel,   input
    ChannelCfg *cfg,         input
    Lid *lid,                output
    unsigned long *hdwflags) output
    );
```

channel Indicates the serial channel to be opened.

cfg Points to the configuration table that defines various configuration parameters such as baud rate, various line parameters, and the

addresses of the callback functions. See Data Structures for more details on the configuration table.

lid Set by the lower-level driver and is the lower level's reference ID for this channel. All calls to the DISI by the upper layer pass **lid** except for the **SerialInit** command.

hdwflags Returned by the DISI to indicate the capabilities of the lower-level serial code. The **hdwflags** flags can be:

SIOCHDWHDL	HDLC supported
SIOCHDWRXPOOL	Has receive buffer pool
SIOCHDMAXTIM	Can do intercharacter timing
SIOCAUTOBAUD	Can do autobaud (sync only)

If SIOCHDWRXPOOL is set, the lower level contains a buffer pool to receive characters and, as they are sent up through the DISI, these buffers need to be replenished. (See the **SerialIoctl** command SIOCREPLENISH and **UDataInd** call for more information.)

The following example shows the use of a **SerialOpen** function call to open a channel:

EXAMPLE

```
/*********************************************************/
/* The Open function is an example of the use of the*/
/* SerialOpen function */
/*  */
/* It takes one argument the channel number to open.*/
/*********************************************************/

/*********************************************************/
/* The global array call lids will be used to store*/
/* the lower IDs*/
/*********************************************************/
unsigned long lids[NUMBER_OF_CHANNELS];

unsigned long Open(int channel)
{
ChannelCfg channelcfg;

/*********************************************************/
/* Set up configuration structure that will be passed*/
/* to DISI interface.*/
/*********************************************************/
/* Clear the ChannelCfg structure */
/*********************************************************/
```

```
bzero(&channelcfg, sizeof(ChannelCfg));

/******************************************************/
/* Set Mode to UART mode*/
/******************************************************/
channelcfg.Mode = SIOCASYNC;

/******************************************************/
/* Set character size to 8 bits */
/******************************************************/
channelcfg.Cfg.Uart.CharSize = SCS8;

/******************************************************/
/* Set Flags for software flow control and to cause an*/
/* interrupt when a break is received.*/
/******************************************************/
channelcfg.Cfg.Uart.Flags = SBRKINT | SWFC;
/******************************************************/
/* Set the channels baudrate.NOTE SysBaud is a global*/
/* variable defined by pSOSystem to the default baud rate*/
/******************************************************/

channelcfg.Cfg.Uart.LineD[0].LChar = NL;
channelcfg.Cfg.Uart.LineD[0].LFlags = 0;
channelcfg.Cfg.Uart.LineD[1].LChar = EOT;
channelcfg.Cfg.Uart.LineD[1].LFlags = ENDOFTABLE;

/******************************************************/
/* Set Xon and Xoff characters to be used for software*/
/* flow control */
/******************************************************/
channelcfg.Cfg.Uart.XOnCharacter = XON;
channelcfg.Cfg.Uart.XOffCharacter = XOFF;

/******************************************************/
/* Set MinChar and MaxTime so at least one character will*/
/* be received and at most four characters. If three*/
/* tens of a second pass between characters, a read*/
/* request will be considered filled and the UDataInd*/
/* function will be called*/
/******************************************************/
channelcfg.Cfg.Uart.MinChar = 4;
channelcfg.Cfg.Uart.MaxTime = 3;

/******************************************************/
```

```
/* Set the receive buffer size to 4 characters   */
/*******************************************************/
channelcfg.RBuffSize = 4;

/*******************************************************/
/* Set the len of transmit request to 4 so there can*/
/* be only 4 requests outstanding at one time*/
/*******************************************************/
channelcfg.OutQLen = 4;

/*******************************************************/
/* Set the channels baudrate.*/
/*******************************************************/
channelcfg.Baud = SysBaud;

/*******************************************************/
/* Set the line mode to full duplex*/
/*******************************************************/
channelcfg.LineMode = FULLD;
/*******************************************************/
/* Set the pointers to the call back functions */
/*******************************************************/
channelcfg.dataind = term_dataind;
channelcfg.expind = term_expind;
channelcfg.datacnf = term_datacnf;
channelcfg.ctlcnf = term_ctlcnf;
channelcfg.allocb = gs_allocb;
channelcfg.freemsg = gs_freemsg;
channelcfg.esballoc = gs_esballoc;

/*******************************************************/
/* Set the ID to be used by the lower driver when*/
/* referencing this channel.       */
/*******************************************************/
channelcfg.uid = channel;

/*******************************************************/
/* Call the DISI interface open*/
/*******************************************************/
if(error = SerialOpen(channel, (ChannelCfg *)&channelcfg,
           (Lid )&lids[channel],
           (unsigned long *)&DChanCfg[minor].hdwflags))
    {
    /***************************************************/
    /* Return error code.*/
    /***************************************************/
```

```
switch (error)
    {
    case SIOCAOPEN:
         /********************************************/
         /* The Channel has already been opened by */
         /* another driver*/
         /********************************************/
        return(1);

    case SIOCBADCHANNELNUM
        /********************************************/
        /* Channel is not a valid channel for this*/
        /* hardware*/
        /********************************************/
        return(2);
    case SIOCCFGNOTSUPPORTED
        /********************************************/
        /* Hardware cannot be configured by the*/
        /* DISI as given*/
        /********************************************/
        return(3);

    case SIOCBADBAUD:
        /********************************************/
        /* Baud rate not supported by hardware.*/
        /********************************************/
        return(4);

    case SIOCBADMINCHAR:
        /********************************************/
        /* MinChar is greater then receive buffer*/
        /* size.*/
        /********************************************/
        return(5);

    case SIOCNOTINIT:
        /********************************************/
        /* This error shows that the lower driver*/
        /* thinks it has not been initialized.*/
        /********************************************/
        return(6);
    }
```

SerialSend Function

The **SerialSend** function is used by the upper level serial driver to transfer data to the lower-level driver.

```
long SerialSend(
            Lid lid,          input
            mblk_t* mbp       input
            );
```

lid The lower-level ID that was acquired during **SerialOpen** operation for the channel to which this is directed.

mbp A pointer to the message block that contains the data to be transmitted.

A 0 return code indicates that the message block has been queued to send. The **UDataCnf** callback will be used by the lower-level driver when the data in the message block has actually been sent.

NOTE: If a SIOCOQFULL error is received, no data was sent because the transmit queue is full. SerialSend continues to return SIOCOQFULL until the next UDataCnf callback happens. Since UDataCnf is the confirmation of a message being sent, the transmit queue will no longer be full.

EXAMPLE

The following example shows the use of a **SerialSend** call to send data to the lower serial driver.

```
/*-------------------------------------------------------------*/
/* This is an example of a function that will get a mblock from*/
/* the mblock pool, fill the mblock's data buffer with some    */
/* information and send it to the lower serial driver.         */
/*-------------------------------------------------------------*/
#include <gsblk.h>
#include <disi.h>

static char test_string[] = "This is a Test Buffer";
/*************************************************************/
/* SendData:  Gets a mblock, puts some data into it and sends  */
/*            it to the lower driver.                          */
/*                                                             */
/*              (Lid)lid lower level id gotten when the        */
/*               SerialOpen call was made.                     */
/*                                                             */
/*     RETURNS: 0 on success                                   */
/*              1 gs_allocb failure                            */
/*              2 SerialSend failure                           */
```

```
/*     NOTE(S):                                                 */
/*                                                              */
/****************************************************************/
int SendData((Lid)lid)
{
int i;

/****************************************************************/
/* The typedefs frtn_t and mblk_t are found in pna.h.          */
/****************************************************************/
mblk_t *m;

/****************************************************************/
/* Call gs_allocb to get a buffer attached to a mblock         */
/* structure.                                                   */
/*                                                              */
/* gs_allocb is a function supplied by pSOSystem in the file   */
/* drivers/gsblk.c. It is compiled into bsp.lib.               */
/*                                                              */
/* gs_allocb takes two arguments                                */
/*          size: size of message block to be allocated        */
/*          pri: allocation priority (LO, MED, HI)             */
/*                                                              */
/* gs_allocb is a utility that allocates a message block of    */
/* type M_DATA and a buffer of a size greater than or equal to */
/* specified size. pri indicates the priority of the allocation*/
/* request. Currently pri is not used and should be set to 0   */
/* On success, gs_allocb returns a pointer to the allocated    */
/* message block. gs_allocb returns a NULL pointer if it could */
/* not fill the request                                        */
/*                                                              */
/* mblk_t *gs_allocb( int size, int pri)                       */
/*                                                              */
/* A mblk_t structure looks like this:                         */
/*                                                              */
/*   struct msgb                                               */
/*       {                                                      */
/*       struct msgb    *b_next;   next msg on queue          */
/*       struct msgb    *b_prev;   previous msg on queue      */
/*       struct msgb    *b_cont;   next msg block of msg      */
/*       unsigned char  *b_rptr;   first unread data byte in  */
/*                                 buffer                     */
/*       unsigned char  *b_wptr;   first unwritten data byte  */
/*                                 in buffer                  */
/*       struct datab   *b_datap;  data block                 */
/*       }                                                      */
```

```
/*                                                            */
/**************************************************************/
if(m = gs_allocb(sizeof(test_string), 0) == 0)
    return(1);

/**************************************************************/
/* Copy data to buffer                                        */
/**************************************************************/
for(i = 0; i < sizeof(test_string); i++, m->b_wptr++)
    *(m->b_wptr) = test_string[i];

/**************************************************************/
/* Send mblock to lower driver                                */
/**************************************************************/
if(SerialSend(lid, m) != 0)
    return(2);
else
    return(0);
}
```

The next example shows the use of the **SerialSend** function to take a list of data buffers and attach them to an mblock and chain the mblocks together so they will all be part of one HDLC frame.

```
#include <gsblk.h>
#include <disi.h>
/***************************************************************/
/* In this sample we will use LEN as the length of the buffers */
/* that are being sent. However the length of a buffer could   */
/* vary in the code. You just need a way to compute each       */
/* buffers length.                                             */
/***************************************************************/
#define LEN 512

/***************************************************************/
/* SendFrame: Attaches buffers of data to mblocks so that the  */
/*            buffers will be sent in a single HDLC frame. This */
/*            is also known as scatter-gather.                  */
/*                                                             */
/*      INPUTS: char **buffs - array of buffer pointers        */
/*              terminated by a null pointer.                  */
/*                                                             */
/*              (Lid)lid lower level id gotten when the        */
/*               SerialOpen call was made.                     */
/*                                                             */
```

```
/*     RETURNS: 0 on success                                   */
/*              1 gs_esballoc failure                          */
/*              2 SerialSend failure                           */
/*     NOTE(S):                                                */
/*                                                             */
/***************************************************************/
int SendFrame(char **buffs, (Lid)lid)
{

/***************************************************************/
/* The typedefs frtn_t and mblk_t are found in pna.h.          */
/***************************************************************/
frtn_t frtn;
mblk_t *m, *mfirst, *mprevious = (mblk_t *)0;

while(*buffs)
    {

    /***********************************************************/
    /* Set up the frtn structure so the retbuff function will   */
    /* be called with an argument that contains the pointer     */
    /* to the buffer that can be reclaimed.                     */
    /*                                                          */
    /* NOTE: retbuff is a function that needs to be supplied    */
    /*       by the user as part of the upper layer code.       */
    /***********************************************************/
    frtn.free_func = (void (*)())retbuff;
    frtn.free_arg = (char *) *buffs;

    /***********************************************************/
    /* Call gs_esballoc to attach buffer to a mblock structure. */
    /*                                                          */
    /* gs_esballoc is a function supplied by pSOSystem in the   */
    /* file drivers/gsblk.c. It is compiled into bsp.lib.       */
    /*                                                          */
    /* gs_esballoc takes four arguments:                        */
    /*                                                          */
    /* unsigned char *base      Base pointer of user buffer     */
    /* int size                 Size of user buffer             */
    /* int pri                  Not Used                        */
    /* frtn_t *frtn             Free function and argument for  */
    /*                          user buffer.                    */
    /***********************************************************/
    if(m = gs_esballoc((unsigned char *)*buffs, LEN, 0, &frtn)) == 0)
    {
```

```
    /*************************************************************/
    /* Free any mblocks used so far.                             */
    /*************************************************************/
    while  (Mfirst)
        {
         m = mfirst;

         while (m->b_cont != (mblk_t *) 0)
             m = m->b_cont;

         if (m == mfirst)
             mfirst = (mblk_t *) 0;
         gs_freemgs(m);
         }

     return(1);
     }

    /*************************************************************/
    /* Increment the mblock's write pointer so it points to     */
    /* the first unwritten character in the buffer.             */
    /*************************************************************/
    m->b_wptr = (m->b_rptr + LEN);

    /*************************************************************/
    /* If this is not the first mblock, then chain this mblock  */
    /* into the mblock chain by setting b_cont of the previous  */
    /* mblock to point the current mblock.                      */
    /*                                                           */
    /* If this is the first mblock then save a pointer to it    */
    /* in mfirst. mfirst will be used in the SerialSend call.   */
    /*************************************************************/
    if(mprevious != (mblk_t *)0)
        mprevious->b_cont = m;
    else
        mfirst = m;
    mprevious = m;

    ++buffs;
    }

if(SerialSend(lid, mfirst) != 0)
    return(2);
else
    return(0);
```

SerialIoctl Function

The **SerialIoctl** function specifies various control operations that modify the behavior of the DISI.

```
long SerialIoctl(
      Lid lid,                input
      unsigned long cmd,      input
      void *arg               input
      )
```

lid The lower-level ID that is acquired during a **SerialOpen** operation.

cmd The type of control operation.

arg Specific information for the operation.

Not all operations listed below need be supported by the lower layer chip set code. Any non-supported operation returns with the error code SIOCOPERATIONNOTSUP.

In some cases, a **SerialIoctl** operation may not complete immediately. In those cases, the **UCtlCnf** function is called when the operation has completed with the final status of the command.

SerialIoctl Commands

The SerialIoctl commands are:

SIOCPOLL Polls the serial device for asynchronous events such as data indication and exception indication. It provides an ability to perform as a pseudo ISR and call the callback functions when the channel is in SIOCPOLL mode or when interrupts are disabled. For example, when pROBE+ is in control, the processor operates with interrupts turned off. This command checks for data received, data transmitted, or exceptions and then triggers the callback function for these conditions, as needed.

SIOCGETA Gets the channel configuration and stores this information into a **ChannelCfg** structure pointed to by the **arg** parameter. This command is immediate, so no callback is made.

SIOCPUTA Sets the channel configuration using the information stored in a **ChannelCfg** structure pointed to by the **arg** parameter. The effect is immediate, so no callback is made.

SIOCBREAKCHK This command will check to see if a break character has been sent. This command is used by pROBE+ to see if the user wants to enter pROBE+. The **arg** parameter is set to SIOCBREAKR if there has been a break sent to the channel.

SIOCPROBEENTRY	This command tells the driver that pROBE+ is being entered. The driver should now switch to the debugger callouts, **uid** and switch from **interrupt** mode to **polled** mode.
SIOCPROBEEXIT	This commands tell the driver the pROBE+ is being exited and the driver should now switch from the debugger callouts to the normal callouts, normal **uid** and allow interrupts. Normal callouts and **uid** are the ones from a **SerialOpen** call. If pROBE+ is the only user of the channel then the normal callouts and **uid** and the debugger callouts and **uid** will be the same.
SIOCBREAK	Sends a break character out the channel. Any argument passed is ignored. This command is immediate, no callback is made.
SIOCMQRY	Gets information about which modem controls are supported by the channel and stores this information into the **long int** pointed to by the **arg** parameter. A set bit indicates that the particular control line is supported by the channel. This command is immediate, so no callback is made.

The modem control lines are:

SIOCMDTR	Data terminal ready
SIOCMRTS	Request to send
SIOCMCTS	Clear to send
SIOCMDCD	Data carrier detect
SIOCMRI	Ring indicator
SIOCMDSR	Data set ready
SIOCMCLK	Clock (sync support)

Since the interface is a DTE, DTR and RTS are outputs and CTS, RI, DSR, and DCD are inputs.

SIOCMGET	Gets the current state of the modem control lines and stores this information into the **long int** pointed to by the **arg** parameter. SIOCMGET uses the same encoding as SIOCMQRY. Bits pertaining to control lines not supported by the channel and the SIOCMCLK bit are cleared. This command is immediate, so no callback is made.
SIOCMPUT	Sets the modem controls of the channel. The **arg** parameter is a pointer to a **long int** containing a new set of modem control lines. The modem control lines are turned on or off, depending on whether their respective bits are set or clear. SIOCMPUT uses the same encoding as SIOCMQRY. Bits pertaining to control lines not supported by the channel and the SIOCMCLK bit have no effect. The effect is immediate, so no callback is made.

SOCFLGET — Gets the current state of the flags (defined by the **UartCfg** structure) and stores this information into an **unsigned long int** pointed to by the **arg** parameter. This call is ignored when the channel is being used in synchronous mode (HDLC). This command is immediate, so no call back is made.

SIOFLPUT — Sets the flags for the channel. The **arg** parameter is a pointer to a **long int** containing a new set of flags defined by the flag element in the **UartCfg** structure. This call is ignored when the channel is being used in synchronous mode (HDLC). The effect is immediate, so no call back is made.

SIOCXFGET — Gets the current **XOFF** character and stores this information into the **long int** pointed to by the **arg** parameter. This command is immediate, so no call back is made.

SIOCXFPUT — Sets the new **XOFF** using the **long int** pointed to by the **arg** parameter. The effect is immediate, so no call back is made.

SIOCXNGET — Gets the current **XON** character and stores this information the **long int** pointed to by the **arg** parameter. This command is immediate, so no call back is made.

SIOCXNPUT — Sets the new **XON** using the **long int** pointed to by the **arg** parameter. The effect is immediate, so no call back is made.

SIOCREPLENISH — Causes the receive buffer pool (if any) to be replenished with new buffers. In some cases, the lower drivers uses a ring of buffers to receive data. As a buffer in the ring is used, it is attached to a **mblk** and sent to the upper driver via a **UDataInd** call. For more efficient operation and to keep the interrupt latency down, the upper driver must use the SIOCREPLENISH command so the lower driver replenishes those buffers. The upper driver configures the size of the buffers in the ring in the **SerialOpen** call by the setting of **RBuffSize** in the **ChannelCfg** structure. The number of buffers in the ring is also set in the **SerialOpen** call by setting **NRBuffs**. The upper driver code should keep track of the number of buffers used (one used each time **UDataInd** is called) and use the SIOCREPLENISH command when it determines more should be added to the receive buffer pool. This level should be a factor of the amount of data being received and the baud rate. It should then be set so the lower driver does not run out of buffers. Of course, the upper driver can also use the SIOCREPLENISH command every time **UDataInd** is called. Since **UDataInd** is called as part of the interrupt routine, using the SIOCREPLENISH command causes the interrupt to take longer.

SIOCREPLENISH is necessary only if the **hdwflags** passed in the **SerialOpen** call had the SIOCHDWRXPOOL bit set. If **SIOCHDWRXPOOL** is not set, the lower driver maintains its own buffer pool and the command is ignored. This command is immediate, so no call back is made.

SIOCGBAUD
: Gets the baud rate of the channel and stores this information into the **long int** pointed to by the **arg** parameter. This command is immediate, so no call back is made.

SIOCSBAUD
: Sets the new baud rate for the channel using the information stored in the **long int** pointed to by the **arg** parameter. The effect is immediate so no call back is made.

SIOCGCSIZE
: Gets the character size (in bits) and stores this information into the **long int** pointed to by the **arg** parameter. This command is immediate, no call back is made.

SIOCSCSIZE
: Sets the new character size (in bits) using the information stored in the **long int** pointed to by the **arg** parameter. The effect is immediate so no call back is made.

SIOCSACTIVATE
: Activates the channel. This enables the receiver and transmitter of the channel and waits until the channel becomes active. In dial-in connections, the SIOCSACTIVATE command puts the hardware in a mode capable of handling an incoming call. The **UCtlCnf** callback is made when the call arrives. When using HDLC (even when no dial-up connection is involved), the **UCtlCnf** callback is made when the link is active, i.e., it starts receiving flags.

SIOCSDEACTIVATE
: Deactivates the channel. This disables the receiver and transmitter of the channel. The SIOCSDEACTIVATE command drops the connection (DTR) and invalidates the transmitter and the receiver. The effect is immediate so no call back is made.

SIOCTXFLUSH
: Discards all characters in the transmit queue for the channel. The **UDataCnf** callback is made for each message that was discarded with **b_flags** set to SIOCABORT. A **UCtlCnf** callback is made when the transmit queue is empty.

SIOCRXFLUSH
: Closes the current receive buffer. This causes **UDataInd** to be called for the current mblock. Serial interrupts must be blocked before making this call. A **UCtlCnf** callback is made when the command is completed. Serial_interrupts should be enabled when the **UCtlCnf** callback is received for this command.

SIOCRXSTOP	Stops the flow of receive characters. This is used when the upper level serial driver needs to stop the flow of characters it is receiving. The lower-level serial code takes the correct action such as sending an **XOFF** character if software flow control is being used or changing the hardware lines if hardware flow control is being used. The effect is immediate so no call back is made.
SIOCRXSTART	Indicates that the upper level serial driver wants to continue to receive characters. The lower-level serial code will take the correct action such as sending an **XON** character if software flow control is being used or changing the hardware lines if hardware flow control is being used. The effect is immediate so no call back is made.
SIOCNUMBER	Gets the total number of serial channels and stores this information into the **long int** pointed to by the **arg** parameter. This command is immediate, so no call back is made.
SIOCAUTOBAUD	Allows the channel to automatically set the baud instead of using the given baud rate, parity, and character size.

EXAMPLE

The following example shows the use of a **SerialIoctl** function call to get the baud rate of the channel.

```
/*******************************************************/
/* This get_baud_rate function is an example of a        */
/* SerialIoctl function call.                            */
/*******************************************************/
int get_baud_rate(unsigned long channel)
{
int baud;
/*******************************************************/
/* Assume the lower level ID is stored by the SerialOpen */
/* call in a global array called lids. Use the         */
/* SIOCGBAUD to get the baud rate and baud as a place to */
/* store the baud rate.                                  */
/*******************************************************/
if(SerialIoctl(lids[channel], SIOCGBAUD, (void *)&baud)
   return(-1);
else
   return(baud);
}
```

SerialClose Function

The **SerialClose** function terminates a connection on a serial channel and returns the channel to its default state.

```
long SerialClose(
            Lid lid        input
            )
```

lid The lower-level ID that was acquired during **SerialOpen** operation for the channel that is to be closed.

If the channel is not open, SIOCNTOPEN is returned.

EXAMPLE

The following example shows a **SerialClose** function call to close the channel.

```
/****************************************************************/
/* This function TermClose is an example of a SerialClose call */
/* SerialClose will close the channel. This will flush all     */
/* transmit buffers, discard all pending receive buffers and   */
/* disable the receiver and transmitter of the channel. All    */
/* rbuffers associated with the channel will be released       */
/* (freed) and the device will hang up the line                */
/*                                                             */
/****************************************************************/

void TermClose (channel)
{

SerialClose((Lid)lids[channel]);

/*All semaphores and queues for the channel should be deleted here.*/
}
```

User Callback Functions

This section contains the templates of the callback functions that must be provided by the upper-level driver. Pointers to these functions are passed in the **ChannelCfg** structure during the **SerialOpen** of the channel to the device-dependent lower-level code. These pointers can be changed via the **SerialIoctl** command SIOPUTA.

NOTE: These calls must be callable from an interrupt. Consequently, it is important that they do not block within the call and only call OS functions that are callable from an ISR.

UDataInd Callback Function

The **UdataInd** callback function will be called during an interrupt by the device-dependent lower-level code to indicate reception of data to the upper level serial driver.

```
static void UDataInd(
              Uid uid,             input
              mblk_t * mbp,        input
              unsigned long b_flagsinput
              );
```

uid The upper-level serial driver's ID for the associated channel. The ID is passed to the lower-lever serial driver during the **SerialOpen** of the channel on which the data is arriving.

mbp A pointer to message block that contains the data received by the channel.

b_flags The status flags associated with this message block. The flags can be:

SIOCOK	Rx received with out error
SIOCLGFRAME	Frame with exceeding length
SIOCCONTROL	Control Character Received
SIOCMARK	Idle Line Condition
SIOCBREAKR	Break Received
SIOCFRAMING	Framing Error
SIOCPARITY	Parity Error
SIOCOVERRUN	Overrun of buffers
SIOCCDLOST	Carrier Detect Lost

UDataInd must unblock any task that is waiting for data from this channel.

NOTE: If the SerialOpen call returned hdwflags that had the SIOCHDWRXPOOL bit set, then the lower-level code has a receive buffer pool. This pool will need replenishing through the use of a call to SerialIoctl with the command SIOCREPLENISH.

The user supplied functions in the upper layer serial driver must use **SerialIoctl** to replenish the buffers. The upper level serial driver must free the message block (pointed to by **mbp**) when it is emptied by calling **UFreemsg**.

In the case of SIOCCONTROL (Control Character Received) the control character will be the last character in the receive buffer if REJECTCHAR was not set for the **LineD** entry of that character. If REJECTCHAR was set, the control character will not be part of the buffer. In this case **UDataInd** will be call when the control character is received with the current

receive buffer. The last character in the buffer will be the character received just before the control character was received.

EXAMPLE

The following example shows a **UDataInd** function call to send data and status to a task.

```
/**************************************************************/
/* This function term_dataind is an example of a UDataInd      */
/* function. It will get as input:                             */
/*                                                             */
/*              Uid uid pointer to channels configuration      */
/*              mblk_t mblk message block containing data      */
/*              unsigned long b_flags condition code for block */
/*                                                             */
/* term_dataind will use a message queue to send the mblock    */
/* and status on to a task that is waiting for data.           */
/*                                                             */
/* Assume receive_ques is an array of message queue IDs.       */
/**************************************************************/
static void term_dataind(Uid uid, mblk_t *mblk, unsigned long b_flags)
{

/**************************************************************/
/* Set up the message buffer with the pointer to the mblock    */
/* and status                                                  */
/**************************************************************/
msg_buf[0] = (unsigned long)mblk;
msg_buf[1] = b_flags;

/**************************************************************/
/* Send message to channels message queue.                     */
/**************************************************************/
q_send(receive_ques[(unsigned long)*uid], msg_buf);
}
```

UExpInd Callback Function

The **UExpInd** callback function is called by the device-dependent lower-level code to indicate an exception condition.

```
static void UExpInd(
            Uid uid,            input
            unsigned long exp   input
            );
```

uid The upper-level serial driver's ID for the associated channel which is passed to the lower-lever serial driver during the **SerialOpen** of the channel on which the exception has occurred.

exp Type of exception.

Exceptions can be one of the following:

SIOCMARK	Idle Line Condition
SIOCBREAKR	Break Received
SIOCFRAMING	Framing Error
SIOCPARITY	Parity Error
SIOCOVERRUN	Overrun of buffers
SIOCCDLOST	Carrier Detect Lost
SIOCCTSLOST	Clear To Send has been lost
SIOCNAFRAME	Frame not divisible by 8
SIOCABFRAME	Frame aborted
SIOCCRCERR	CRC error
SIOCCTS	Clear To Send found
SIOCCD	Carrier Detect detected
SIOCFLAGS	Non Idle Line Condition

UDataCnf Callback Function

The **UDataCnf** callback function is called by the device-dependent lower-level code to confirm that the data sent using **SerialSend** call has been transmitted.

```
static void UDataCnf(
              Uid uid,            input
              mblk_t * mbp,       input
              unsigned long b_flagsinput
              );
```

uid The upper-level serial driver's ID for the associated channel which is passed to the lower-lever serial driver during the **SerialOpen** of the channel on which the data was sent.

mbp Points to the message block sent using **SerialSend** call.

b_flags Status flags associated with the message block. The **b_flags** must be one of the following:

SIOCOK	Completed without error
SIOCUNDERR	Tx underrun (HDLC)
SIOCABORT	Tx aborted

The **UDataCnf** function must unblock any task that was waiting for data to be sent. The task is responsible for any maintenance necessary to the message block such as freeing it or reusing it.

EXAMPLE

The following example shows a **UDataCnf** function call to confirm that data has been sent.

```
/***************************************************************/
/* This function term_datacnf is an example of a UDataCnf      */
/* function. It takes as inputs:                               */
/*                                                             */
/*              Uid uid pointer to channels number             */
/*              mblk_t mblk message block containing data      */
/*              unsigned long b_flags condition code for block */
/*                                                             */
/* This code assumes that the driver is not waiting for        */
/* completion of a transmission.                               */
/***************************************************************/
static void term_datacnf(Uid uid, mblk_t *mblk, unsigned long b_flags)
{
gs_freemsg(mblk);
}
```

UCtlCnf Callback Function

The **UCtlCnf** callback function is used to confirm the completion of a **SerialIoctl** control command.

```
  static void UCtlCnf(
           Uid uid,                input
           unsigned long cmd       input
           );
```

uid The upper-level serial driver's ID for the associated channel which is passed to the lower-lever serial driver during the **SerialOpen** of the channel on which the I/O control call was made.

cmd The command being confirmed.

EXAMPLE

The following example shows a **UCtlCnf** function call to confirm the completion of a **SerialIoctl** control command.

```
/*******************************************************/
/* static void term_ctlcnf                             */
/*                                                     */
/* This function term_ctlcnf is an example of a UCtlCnf  */
/* function. It takes as inputs:                       */
/*                                                     */
/*             Uid uid pointer to a configuration      */
/*             unsigned long cmd I/O control cmd that  */
/*             is being confirmed.                     */
/*                                                     */
/* term_ctlcnf assumes that a task is waiting for a    */
/* semaphore.                                          */
/* semaphore_ctl_ids is an array that stores the ID for  */
/* each channel                                        */
/*******************************************************/
void term_ctlcnf(Uid uid, unsigned long cmd)
{

/*-----------------------------------------------------*/
/* Release the channels I/O Control semaphore          */
/*-----------------------------------------------------*/
sm_v(semaphore_ctl_ids[(unsigned long)*uid]);
}
```

Access Memory Services

The following callback functions are used to manage message blocks and a buffer pool. The message blocks are similar to those used by Streams I/O. See the **pna.h** file in the **include** directory of the pSOSystem release for a definition of the message block structures used here. All of these functions are provided with the pSOSystem operating system. They are found in the file **drivers/gsblk.c**.

UEsballoc Callback Function

The **UEsballoc** callback function returns a message block triplet by attaching the user supplied buffer as a data block to a message block structure. See the **SendFrame** example under the **SerialSend** function for an example of this call.

```
static mblk_t * UEsballoc(
             char *bp,           input
             long len,           input
             long pri,           input
             frtn_t *frtn        input
             );
```

bp Points to the use-supplied buffer.

len Specifies the number of bytes in the buffer.

pri Specifies the priority for message block allocation.

frnt Pointer to the free structure of type **frtn_t.** This structure is as follows:

```
typedef struct
    {
    void (*free_func)();
    void *free_arg;
    } frtn_t
```

free_func **UFreemsg** calls the function pointed to by **free_func** when the caller-supplied buffer needs to be freed. The caller must supply the function pointed to by **free_func**.

free_arg A pointer to the user supplied buffer.

frtn_t The pointer to **frtn_t** must be stored by the **UEsballoc** call. This makes it available to the **UFreemsg** call when **UFreemsg** is used to free the the message block.

The **UEsballoc** call may be used by the upper or the lower levels of the interface. In either case the "user" is who ever is making the call. One use of **UEsballoc** is a case where there is a special ram area to be used by the serial chip.

NOTE: This function corresponds to the gs_esballoc function supplied by pSOSystem in the file drivers/gsblk.c. It is compiled into bsp.lib. You may use a pointer to gs_esballoc for the UEsballoc callback function.

UAllocb Callback Function

The **UAllocb** callback function returns a message block triplet or a NULL if no buffer or message block could be found. See the **SendData** example under the **SerialSend** function for an example of this call.

```
static mblk_t * UAllocb(
        long size,       input
        long pri         input
        );
```

size Specifies the size of the buffer.

pri Specifies the priority for message block.

NOTE: This function corresponds to the gs_allocb function supplied by pSOSystem in the file drivers/gsblk.c. It is compiled into bsp.lib. You may use a pointer to gs_allocb for the UAllocb callback function.

UFreemsg Callback Function

The **UFreemsg** callback function is used to free a message block. See the **term_ctlcnf** example under the **UDataCnf** function for an example of this call.

```
static void UFreemsg(
        mblk_t *mbp,   input.
        );
```

mbp Points to the message block triplet for this specific message block pool. If the message block was formed using the **UEsballoc** call, **UFreemsg** calls the function pointed by **free_func** with a pointer to **free_arg** as its argument.

NOTE: This function corresponds to the gs_freemsg function supplied by pSOSystem in the file drivers/gsblk.c. It is compiled into bsp.lib. You may use a pointer to gs_freemsg for the UFreemsg callback function.

Data Structures

Following are templates of data structures. They can be found in **include/disi.h.**

CCfg

```
typedef struct ccfg {
   unsigned long  Mode;
   Modecfg        Cfg;
   unsigned long  RBuffSize;
   unsigned long  NRBuffs;
   unsigned long  OutQLen;
   unsigned long  Baud;
   unsigned long  LineMode;
   void           (*dataind)(uid,mblk_t, unsigned long);
   void           (*expind)(uid, unsigned long);
   void           (*datacnf)(uid,mblk_t,unsigned long);
   void           (*ctlcnf)(uid, unsigned long);
   mblk_t         * (*allocb)(long, long);
   void           (*freemsg)(mblk_t);
   mblk_t         * (*esballoc)(char,long, long,frtn_t);
   Uid            uid;
   unsigned long  Reserve[4];
   } ChannelCfg;
```

Mode — **Mode** can be:

SIOCSYNC	Sync mode - async if not set
SIOCPOLLED	Poll mode - interrupt if not set
SIOCLOOPBACK	Local loop back mode

The **Mode** decides which structure to use; if SIOCSYNC is set,**HdlcCfg** is used, otherwise **UartCfg** is used.

```
typedef union {
     struct HdlcCfg Hdlc;
     struct UartCfg Uart;
     } ModeCfg;
```

RBuffSize — The size of the receive buffers. **RBuffSize** can only be set during the **SerialOpen** call. It cannot be changed by a **SerialIoctl** call.

NRBuffs — The number of receive buffers to allocate for the receive queue.

OutQLen — The maximum number of message buffers waiting to be transmitted. If the maximum number is exceeded, **SerialSend** fails with an SIOCOQFULL error**.**

Baud — Set to the actual desired baud rate. If the selected baud rate is not supported by the lower-level device-dependent code, **SerialOpen** or **SerialIoctl** fails, an error is returned.

LineMode — **LineMode** can be:

HALFD	Half-Duplex
FULLD	Full-Duplex
MULTIDROP	Multi-Drop lines

dataind — Pointer to a data indication routine. See **UDataInd** for additional information.

expind — Pointer to an exception indication routine. See **UExpInd** for additional information.

datacnf — Pointer to a data confirmation routine. See **UDataCnf** for additional information.

ctlcnf — Pointer to a control confirmation routine. See **UCtlCnf** for additional information.

allocb — Pointer to an allocate message block routine. See **UAllocb** for additional information.

freemsg Pointer to a free message list routine. See **UFreemsg** for additional information.

esballoc Pointer to an attach message block routine. See **UEsballoc** for additional information.

HdlcCfg

```
struct HdlcCfg{
   unsigned char  TxClock;
   unsigned char  RxClock;
   unsigned char  Modulation;
   unsigned char  Flags;
   unsigned short Crc32Bits;
   unsigned short MaxFrameSize;
   unsigned short Address[4];
   unsigned short AddressMask;
   unsigned long  FrameCheckErrs;
   unsigned long  TransmitUnderrunErrs;
   unsigned long  ReceiveOverrunErrs;
   unsigned long  InterruptedFrames;
   unsigned long  AbortedFrames;
   unsigned long  Reserve[4];
   };
```

TxClock and **RxClock**

Can be:

CLK_INTERNAL	Internal clock (xmit only)
CLK_EXTERNAL	External supplied clock
CLK_DPLL	Digital Phase Lock Loop
CLK_INVERT	Transmit DPLL invert data

Modulation Can be:

MOD_NRZ

MOD_NRZI_MARK

MOD_NRZI_SPACE

MOD_FM0

MOD_FM1

MOD_MANCHESTER

MOD_DMANCHESTER

Flags Number of interframe flags

Crc32Bits Can be CRC32. If not set, 16 bit CRC is assumed.

MaxFrameSize Used to discard any frame that is greater than the value of **MaxFrameSize**.

Address The addresses to be recognized. There must be 4 values in the **Address** fields (duplicates are allowed) because, in HDLC, no single character can serve as an end-of-list indicator.

AddressMask Determines which of the possible 16 bits (of each **Address**[i]) are used to filter the addresses of the received frames: 0 means no filtering, 0xFF means 8-bit addresses and 0xFFFF means 16-bit addresses. Other masks are allowed to filter on fewer than 8 bits: for example the mask 0x00F0 with **Address**[i] set to 0x00C0 will cause the driver to receive only frames that have their first byte starting with 0xC0 to 0xCF.

FrameCheckErrs
The total number of frames with an invalid frame check sequence input from the channel since the system re-initialized and while the channel was active. This data is collected for the MIB.

TransmitUnderrunErrs
The total number of frames that failed to be transmitted on the channel since the system was re-initialized and while the channel was active. **TransmitUnderrunErrs** can occur because data was not available to the transmitter in time. This data is collected for the MIB.

ReceiveOverrunErrs
The total number of frames that failed to be received on the channel since the system was re-initialized and while the channel was active. **ReceiveOverrunErrs** can occur because the receiver did not accept the data in time. This data is collected for the MIB.

InterruptedFrames
The total number of frames that failed to be received or transmitted on the channel since the system was re-initialized and while the channel was active. **InterruptedFrames** can occur because of loss of modem signals. This data is collected for the MIB.

AbortedFrames
The number of frames aborted on the channel since the system was re-initialized and while the channel was active. **AbortedFrames** can occur due to receiving an abort sequence. This data is collected for the MIB.

Reserved Reserved field.

UartCfg

```
struct UartCfg{
   unsigned long  CharSize;
   unsigned long  Flags;
   LineD          Lined[2];
   unsigned char  XOnCharacter;
   unsigned char  XOffCharacter;
   unsigned short MinChar;
   unsigned long  MaxTime;
   unsigned long  ParityErrs;
   unsigned long  FramingErrs;
   unsigned long  OverrunErrs;
   unsigned long  Reserve[4];
   }
```

CharSize — **CharSize** can be:

CS5	5 bits per character
CS6	6 bits per character
CS7	7 bits per character
CS8	8 bits per character

Flags — **Flags** can be:

CANON	Canonical mode
C2STOPB	Send two stop bits, else one
PARENB	Parity enable
PARODD	Odd parity, else even
HWFC	Hardware flow control on
SWFC	Software flow control on
SWDCD	Software data carrier detect
LECHO	Enable local echo
BRKINT	Interrupt on reception of Break
DCDINT	Interrupt on loss of DCD
AUTOBAUDENB	Enable autobaud

When CANON is set, the input is processed and assembled in blocks of data with the use of the line-delimiters in **LineD**. When the block is assembled, the **UDataInd** callback function is called. The **MinChar** and **MaxTime** arguments are ignored when CANON is set. If CANON is not set, the delimiters in **LineD** are ignored and the values of

MinChar and **MaxTime** are used to determine when to call the **UDataInd** callback function.

When PARENB is set, parity generation and detection is enabled and a parity bit is added to each character. When parity is enabled, **odd** parity is used if the PARODD flag is set, otherwise **even** parity is used.

When **HWFC** is set, the channel uses CTS/RTS flow control. If the channel does not support hardware flow control, this bit is ignored.

When SWFC bit is set, **XON/XOFF** flow control is enabled.

When SWDCD is set, the channel responds as if the hardware data carrier detect (DCD) signal is always asserted. If SWDCD is not set, the channel is enabled and disabled by DCD.

When BRKINT is set, the channel issues an **UExpInd** exception callback function if a break character is received.

When DCDINT is set, the channel issues an **UExpInd** exception callback function upon loss of the DCD signal.

When AUTOBAUDENB is set, the channel may use the auto baud feature if it is supported by the lower-level driver.

LineD An array of structures defined as follows:

```
typedef struct
{
unsigned char LChar;
unsigned char LFlags;
} LineD;
```

LChar Any 8 bit value that the user wants to use as a character that, when received, causes an interrupt that will cause the **UDataInd** function to be called.

LFlags A bit field that controls the characters use as follows:

ENDOFTABLE - non valid (last entry in table)

If table has 2 entries none will have this bit set

REJECTCHA - character is rejected

If REJECTCHAR is set, the character does not become part of the buffer and an interrupt is generated but the buffer is not closed (characters will still be received). If REJECTCHAR is not set, an interrupt is generated and the character is the last character in the buffer. The buffer is closed and another buffer is used to receive data.

If this function is not supported by the chip set it must be emulated by the lower-level device-dependent code.

XOnCharacter Software flow control character used to resume data transfer.

XOffCharacter Software flow control character used to temporarily terminate data transfer.

MinChar/Maxtime

Used in non-canonical mode processing (CANON bit not set in flags). In non-canonical mode input processing, input characters are not assembled into lines. The **MinChar** and **MaxTime** values are used to determine how to process the characters received.

MinChar represents the number of characters that are received before **UDataInd callback function** is called**. MinChar** cannot be larger than **RBuffSize.**

MaxTime is a timer of 0.10 second granularity that is used to override the **MinChar** value so the driver does not wait forever for that amount of characters. The four possible values for **MinChar** and **MaxTime** and their interactions are described below.

1. If **MinChar** > 0 and **MaxTime** > 0 then **MaxTime** serves as an intercharacter timer and is activated after the first character is received. Since it is an intercharacter timer, it will be reset after a character is received. The interaction between **MinChar** and **MaxTime** is as follows: as soon as one character is received, the intercharacter timer is started. If **MinChar** characters are received before the intercharacter timer expires, **UDataInd** is called which sends the receive buffer up to the next level. If the timer expires before **MinChar** characters are received, **UDataInd** is called with the characters received to that point.

2. If **MinChar** > 0 and **MaxTime** = 0 then, since the value of **MaxTime** is zero, only **MinChar** is significant. **UDataInd** is not called until **MinChar** characters are received.

3. If **MinChar** = 0 and **MaxTime** > 0 then, since **MinChar** = 0, **MaxTime** no longer represents an intercharacter timer but serves as a read timer. It is activated as soon as a **read()** is started. **UDataInd** is called as soon as a single character is received or the timer expires.

4. If **MinChar** = 0 and **MaxTime** = 0 then no action is required by the lower-level code. The lower-level code uses **RBuffSize** as the number of characters to receive before calling **UDataInd**.

ParityErrs Keeps track of the parity errors that happen on the channel. This information is used by MIB.

FramingErrs Keeps track of the framing errors that happen on the channel. This information is used by MIB.

OverrunErrs Keeps track of the overrun errors that happen on the channel. This information is used by MIB.

Error Codes

The following error codes can be returned:

SIOCAOPEN	Channel already open.
SIOCBADCHANNELNUM	Channel does not exist.
SIOCCFGNOTSUPPORTED	Configuration not supported.
SIOCNOTOPEN	Channel not open.
SIOCINVALID	Command not valid.
SIOCBADARG	Argument not valid.
SIOCOPERATIONNOTSUP	Operation not supported.
SIOCOQFULL	Output queue full, send failed.
SIOCBADBAUD	Baud rate not supported.
SIOCBADMINCHAR	MinChar > RBuffSize.
SIOCWAITING	Waiting for previous command to complete.
SIOCNOTINIT	Driver not initialized.

Multiplex Driver Mapping

This describes how the lower chip level serial drivers for different chip types are to be multiplexed under the DISI so they can be used by one upper level driver. This is used when there is more then one type of serial chip. The method described here uses a private table under the **bsp**. The mapping tells the DISI what lower chip specific driver to call and the port number in that driver to use. This table would be set up in **board.c**.

SDRVCNFG Structure

This is an array of structures that maps a channel number to a serial driver and physical port number in that driver. The structure is defined in **include/disi.h** as:

```
    typedef struct
    {
    unsigned long dnum;/* Driver number */
    unsigned long pnum;/* Physical port number */
    Lid   lid;         /* Lower driver port ID */
} SDRVCNFG
```

There is one **SDRVCNFG** for each serial channel in the system.

SERIALFUNCS Structure

In order to multiplex more then one driver the names of the function call entry points for the lower-level driver code need to be different for each driver. For example, instead of SerialOpen function name for the port open function you might use ser_360_Open in a driver for the MC68360 chip. Each lower-level serial driver will have an entry in an array of structures called **SerialFuncs**. The **SerialFuncs** is an array of **SERIALFUNCS** structures. The SERIALFUNCS structure maps the DISI function calls with corresponding driver functions. The structure is defined in **include/disi.h** as follows:

```
    typedef struct
    {
    long (*Init)(void);
    long (*Open)(unsigned long, ChannelCfg *, Lid *, unsigned long) *;
    long (*Send)(Lid, mblk_t *);
    long (*Ioctl)(Lid, unsigned long, void *);
    long (*Close)(Lid);
    }SERIALFUNCS;
```

There is one SERIALFUNCS for each chip level driver.

The file **drivers/ser_mplx.c** contains multiplexed **DISI** driver entry functions. To use these functions requires a **#define int bsp.h** called **BSP_NUM_SER_DRVRS**. This #define should be set to the number of serial drivers that will be multiplexed. Also required to use these functions are two structures that should be declared in board.c **SerialFuncs** and **SDrvCnfg**. The following is an example of these two structures. These examples assumes there are two drivers to multiplex, a MC6836 serial driver called ser_360 and a Zilog 8530 driver that will be called Z8530:

```
/*               Driver             Driver #*/
       #define ser_360              0
       #define Z8530                1
```

SERIAL_FUNCS SerialFuncs[] =

```
/* INIT           OPEN            SEND            IOCTL           CLOSE       */
ser_360_Init, ser_360_Open, ser_360_Send, ser_360_Ioctl,ser_360_Close,
Z8530_Init,   Z8530_Open,   Z8530_Send,   Z8530_Ioctl,  Z8530_Close;
```

SDRVCFNG SDrvCnfg[BSP_SERIAL + 1] =

```
/* Driver      Port        Lid     Channel */
0,             0,          0       /* Channel 0 - not used */
ser_360,       1,          0       /* Channel 1 */
ser_360,       2,          0       /* Channel 2 */
ser_360,       3,          0       /* Channel 3 */
Ser_360,       4,          0       /* Channel 4 */
Z8530,         1,          0       /* Channel 5 */
Z8530,         2;          0       /* Channel 6 */
```

Index zero of the array is not used because **pSOSystem** uses 0 for the default system console. The system console is mapped to an actual channel before being looked up in the **SDrvCnfg** array.

The **Lid** element of the **SDrvCnfg** will be set by the call to **SerialOpen** for each port.

In the above example, channel 4 corresponds to index 4 which is the 68360 driver port 4 and channel 6 corresponds to index 6 the Z8530 driver port 2.

NAME

SCSI -- Small Computer System Interface Driver

DESCRIPTION

The SCSI driver is divided into an upper part and a lower part. The upper part is located the file named **scsi.c**, which resides in the **drivers** directory. The lower part is located in the file named **scsichip.c**, which resides in the **src** directory. The location of the **src** directory depends on the CPU board as the following path illustrates:

bsps/*board*/**src**

where *board* can be m147, m167, and so on.

The **scsi.c** file is a system-supplied file that is an interface to all systems that support a SCSI interface. It contains the I/O switch table subroutine calls that pSOS+ uses as an entry point into the SCSI driver. The **scsi.c** driver responds to a SCSI call by formatting a SCSI command and passing it the lower driver (**scsichip.c**). **scsichip.c** then proceeds to execute the SCSI operation in a method determined by the hardware on the individual board.

The **scsichip.c** driver is the software interface to the SCSI device interface (often one or more SCSI chips). **scsichip.c** takes the SCSI operation passed from the upper driver, then it controls the execution of that operation through the SCSI hardware interface. The status of the operation is subsequently returned to the upper driver.

User Interface

The application interacts with the SCSI driver through system calls to the pSOS+ kernel. The paragraphs that follow describe the contents of the four system calls that apply to SCSI operations.

de_init(long DEV, long *IOPB, long *return, long *data_area)

The **de_init()** call initializes the SCSI driver. It must be the first call to the SCSI driver. **DEV** is the major device number of the SCSI driver. **IOPB**, **return**, and **data_area** are dummy pointers that the SCSI driver does not use and which should be set to null.

NOTE: The following two descriptions are for the de_read() and de_write() calls. These calls are the direct way to access a SCSI device. However, a file system type device (such as a disk drive) can be accessed and managed through pHILE+ by using calls to pHILE+. When pHILE+ is used, it supplies the pointer of the buffer_header structure to the SCSI driver. For more information, refer to the *pSOSystem System Concepts* manual.

de_read(long Dev, long *IOPB, long *return)

The purpose of **de_read** is to read from a SCSI device. **Dev** is the result of an OR of the major device number of the SCSI driver (upper 16 bits) with the minor number of the specific SCSI

device (lower 16 bits); **IOPB** is a pointer to the SCSI-specific I/O parameter block structure; and **return** is a pointer to the long word that receives the return value.

de_write(long Dev, long *IOPB, long *return)

The purpose of **de_write** is to write to a SCSI device. **Dev** is the result of an OR of the major device number of the SCSI driver (upper 16 bits) with the minor number of the specific SCSI device (lower 16 bits); **IOPB** is a pointer to the SCSI-specific I/O parameter block structure; and **return** is a pointer to the long word that receives the return value.

The SCSI-specific I/O parameter block (IOPB) that **de_read** and **de_write** pass resides in the **phile.h** include file. This include file resides in the **include** directory and has the following format:

```
/*-----------------------------------------------------------*/
/* Device Driver Buffer Header Structure */
/*----------------------------------------------------------*/
typedef struct buffer_header{
   ULONG b_device;
   ULONG b_blockno;
   USHORT b_flags;
   USHORT b_bcount;
   void *b_devforw;
   void *b_devback;
   void *b_avlflow;
   void *b_avlback;
   void *b_bufptr;
   void *b_bufwaitf;
   void *b_bufwaitb;
   void *b_volptr;
   USHORT b_blksize;
   USHORT b_dsktype;
} BUFFER_HEADER;
```

The following are the only elements from the preceding structure that a SCSI driver uses:

b_device	Must contain the minor device number (the SCSI id of the device).
b_blockno	Must contain the starting block number of the device to start reading or writing.
b_bcount	Must contain the number of blocks to be read from the device.
b_bufptr	Pointer to the buffer to read or written.

b_blksize Size of the block to be read or written in base 2.

de_cntrl(long Dev, long *IOPB, long *return)

The **de_cntrl** call performs one of three special functions on the SCSI device. Although **de_cntrl** can be optional for other I/O calls, the SCSI interface requires it. This subsection describes the three special function commands.

Dev is the major/minor device number of the SCSI driver; **IOPB** is a pointer to the SCSI-specific I/O parameter block structure; and **return** is a pointer to the long word that receives the return value.

The three special functions the SCSI driver can perform by using **de_cntrl** are:

- Formatting a device (**SCSI_CTL_FORMAT**)
- Getting device information (**SCSI_CTL_INFO**)
- Passing a SCSI command to the lower driver (**SCSI_CTL_CMD**)

The **#define** statements for the preceding control commands reside in the **drv_intf.h** include file. This include file resides in the **include** directory. The SCSI-specific IOPB that the **de_cntrl** call passes appears in the **drv_intf.h** file as follows:

```
struct  scsi_ctl_iopb{
    long function
    union
    {
    struct  scsi_info  info;
    struct  scsi_cmd  cmd;
    } u;
}
```

If the function command **SCSI_CTL_FORMAT** is specified (for example, **scsi_ctl_iopb.function=SCSI_CTL_FORMAT**), no other information in the structure should be specified.

If the function command **SCSI_CTL_INFO** is specified (for example, **scsi_ctl_iopb.function=SCSI_CTL_INFO**), the driver fills in the **info** element of the **scsi_ctl_iopb** structure for the application's use. The **scsi_info** structure appears in the **drv_intf.h** file as follows:

```
struct scsi_info{
    unsigned char devtype;        /* Type of device - values defined below */
    unsigned char scsi_id;      /* Devices address on the SCSI bus */
    unsigned char char lun;      /* Device's LUN */
    unsigned char removable;        /* Removable media */
    char vendor[SCSI_VENDOR_SIZE];  /* Device's manufacturer */
    char product[SCSI_PRODUCT_SIZE];/* Model name */
    long blocks;                    /* Capacity in blockd */
```

```
   long blocksize;                /* Size of each block in bytes */
};
```

If the function command **SCSI_CTL_CMD** is specified (for example, **scsi_ctl_iopb.function=SCSI_CTL_CMD**), the application must fill in the command structure. The **scsi_cmd** structure appears in the **drv_intf.h** file as follows:

```
struct scsi_cmd{
   unsigned int target_id;        /* Target id of SCSI device */
   unsigned char *data_ptr;       /* Pointer to in/out data area */
   unsigned int data_in_len;      /* Max data to take in if in */
   unsigned int data_out_len;     /* Amount of data if out */
   unsigned int command_len;      /* Length of SCSI CDB */
   unsigned char *cdb;            /* Pointer to SCSI CDB */
   };
```

where

target_id SCSI device id.

data_ptr Pointer to the area where data for read or write operations is stored.

data_in_len Length of the buffer (specified as a character count) that stores input data. If the data is output data, this should be 0.

data_out_len Length of the data (specified as a character count) that stores output data. If the data is input data, this should be 0.

command_len Size (specified as a character count) of the SCSI CDB.

cdb Pointer to the SCSI Command Descriptor Block (CDB). The CDB describes the actual command that goes to the SCSI device. The CDB must conform to the ANSI SCSI-2 spec and be supported by the device.

pSOS-to-Driver Interface

The pSOS-to-driver interface consists of data structures and subroutine calls that the pSOS+ kernel uses to send requests to the SCSI device driver.

```
/*---------------------------------------------------------------------*/
/* I/O Driver Parameter Structure */
/*---------------------------------------------------------------------*/
struct ioparms{
   unsigned long used;        /* Set by driver if out_retval/err used */
   unsigned long tid;         /* task id of calling task */
   unsigned long in_dev;      /* Input device number */
   unsigned long status;      /* Processor status of caller */
   void *in_iopb;             /* Input pointer to IO parameter block */
   void *io_data_area;        /* not used */
   unsigned long err;         /* For error return */
   unsigned long out_retval;  /* For return value */
};
```

The upper driver contains the following subroutines, which comprise the pSOS-to-driver interface:

void SdrvSetup(void)
This subroutine initializes variables that require a starting value before the subroutine SdrvInit can be called. pSOS+ calls SdrvSetup during system initialization.

void SdrvInit(register struct ioparms *s_ioparms)
This subroutine initializes the SCSI driver and any DMA driver needed for SCSI operation. pSOS+ calls SdrvInit when a **de_init** call is made for the SCSI driver.

void SdrvCntrl(register struct ioparms *s_ioparms)
This subroutine can issue commands to the SCSI interface; return information about a SCSI device; or format a SCSI device. pSOS+ calls SdrvCntrl when a **de_cntrl** call is made for the SCSI driver.

void SdskRead(struct ioparms *sd_ioparms)
This subroutine reads blocks of data from a SCSI disk. pSOS+ calls SdskRead when a **de_read** call is made for the SCSI driver.

void SdskWrite(struct ioparms *sd_ioparms)
This subroutine writes blocks of data to a SCSI disk. pSOS+ calls SdrvWrite when a **de_write** call is made for the SCSI driver.

The preceding upper driver subroutines create the proper SCSI CDB and call lower driver routines to communicate with the SCSI device interface.

Upper-to-Lower Driver Interface

The lower driver in **scsichip.c** must contain the subroutines in the following descriptions. These subroutines comprise the upper-to-lower driver interface. All board support packages from ISI that support SCSI devices contain the upper-to-lower driver interface. User-created board support packages must contain an appropriate user-supplied upper-to-lower driver interface. In either case, the interface must contain the following subroutines:

long chipinit(void)
: This function should initialize the lower SCSI driver and return status. The upper driver subroutine **SdrvInit** calls **chipinit**.

long dma_init()
: This function should initialize any DMA device that SCSI operation requires and then return status. This subroutine may be empty if no DMA initialization is necessary. The preferable location for this function is the **dma.c** file.

long chipexec(TRANS_blk)
: This function takes a CDB input, executes the SCSI command, and returns status. **chipexec** is the lower entry point from the upper driver.

The status that **chipexec** returns must be one of the following:

```
#define STAT_OK        0   /* Operation was successful */
#define STAT_CHECKCOND 1   /* Contingent allegiance condition (should */
                           /* issue a Request Sense to get cause of  */
                           /* failure) */
#define STAT_ERR       2   /* A Gross error has occurred. A command */
                           /* may be formatted incorrectly: retry may */
                           /* correct problem */
#define STAT_TIMEOUT   3   /* A selection timeout occurred. More than */
                           /* likely no device exists at this SCSI id. */
#define STAT_BUSY      4   /* Device busy: it cannot accept another */
                           /* command. */
#define STAT_SEMFAIL   5   /* An s_mp operation has failed, and the */
                           /* driver cannot continue with the */
                           /* current command */
```

The preceding #defines reside in the **scsi.h** include file. This include file resides in the **include** directory.

The following typedef must be defined in **drv/scsichip.h** (lower driver). It must contain the elements in the list that follows. For individual applications, users can add appropriate elements to this typedef for the lower driver's use.

```
/****************************************/
/* Transaction interface structure */
/***************************************/
typedef struct trans_blk{
   unsigned int id;                  /* SCSI device id/interrupt level */
   unsigned int lun;                 /* Logical Unit Number */
   unsigned int cmdl;                /* Command Descriptor Block Length */
   unsigned char cmd[MAX_CDB];       /* Command Descriptor Block */
   unsigned int data_len;            /* Number of data bytes to transfer */
   unsigned char *data_ptr;          /* Pointer to data area */
   unsigned char *original_data_ptr; /* Data area to use */
   int original_data_len;            /* Data out length (actual) */
   unsigned char *next_trans_blk;    /* Pointer to next trans_blk */
   TARGET_DEV *target_dev;           /* Pointer to target device struct */
}TRANS_BLK
```

The upper driver passes the preceding transaction structure to the lower driver by executing the **chipexec** subroutine. The structure contains all the information needed to perform the requested SCSI operation.

NAME

SLIP- Serial Line Internet Protocol

DESCRIPTION

Serial Line Internet Protocol (SLIP) is a packet-framing protocol that defines a sequence of characters to frame IP packets on a serial line. It does not provides addressing, packet type identification, error detection/correction, or compression mechanisms. SLIP is commonly used on dedicated serial links and is usually used with line speeds between 1200bps and 19.2Kbps. It is useful for allowing a mix of hosts and routers to communicate with one another (host-host, host-router and router-router are all common SLIP network configurations).

The SLIP driver in pSOSystem is an implementation of the SLIP protocol as defined in RFC1055. It also supports Van Jacobson TCP/IP header compression as defined in RFC1144. The driver is implemented as a Network Interface (NI) to the pNA+ component to allow TCP/IP operations over serial lines. It can be used by networking applications or it can be configured as a standard pNA+ Network Interface to support the Integrated Systems source-level debugger.

SLIP driver can be configured into pNA+ either by using a **add_ni()** call or by configuring it into the pNA+ Network Interface table. It must be configured into the initial Network Interface table if it is to be used by the debugger.

Configuration

There are several site-dependent parameters that are required to configure the SLIP driver into pSOSystem. These are defined in the file **slip_conf.h**. The parameters are defined using the C *define* statement.

SLIP_CHANNEL Specifies the serial channel number for the SLIP interface.

SLIP_MTU Specifies the Maximum Transmission Unit (MTU) for the SLIP interface. This value must be equal to the peer's MTU. The parameter additionally defines the size of the buffers allocated at the local node.

CSLIP If set to 1, Van Jacobson TCP/IP header compression is performed on the SLIP interface. If set to 0, Van Jacobson header compression is not performed on the SLIP interface.

SLIP_LOCAL_IP Defines the IP address of the SLIP interface.

SLIP_PEER_IP Defines the peer IP address of the SLIP interface.

SLIPBUFFERS Defines the number of buffers configured in the SLIP driver. This includes both the receive and transmit buffers. The size of the buffers configured will be twice the **SLIP_MTU** value.

SLIP_CONF Example

An example of a **slip_conf.h** is shown below:

```
/**********************************************************************/
/* */
/*   MODULE: slip_conf.h */
/*   PRODUCT: pNA+ */
/*   PURPOSE: User configurations for SLIP driver */
/*   DATE: 93/11/15 */
/* */
/**********************************************************************/
/* */
/*              Copyright 1993, Integrated Systems Inc. */
/*                      ALL RIGHTS RESERVED */
/* */
/*   This computer program is the property of Integrated Systems Inc.*/
/*   Santa Clara, California, U.S.A. and may not be copied */
/*   in any form or by any means, whether in part or in whole, */
/*   except under license expressly granted by Integrated Systems Inc. */
/* */
/*   All copies of this program, whether in part or in whole, and */
/*   whether modified or not, must display this and all other */
/*   embedded copyright and ownership notices in full. */
/* */
/**********************************************************************/
#ifndef __SLIP_CONF_H__
#define __SLIP_CONF_H__

/*====================================================================*/
/* User configuration parameters */
/*====================================================================*/
#define SLIP_CHANNEL3               /* which channel to use as SLIP */
#define SLIP_MTU1006                /* also used for buffer size */
#define CSLIP1                      /* define to 0 for plain SLIP! */
#define SLIP_LOCAL_IP0xc1000002     /* 193.0.0.2 */
#define SLIP_PEER_IP0xc1000004
#define SLIPBUFFERS32               /* Number of slip buffers */

/*====================================================================*/
/* End of user configurations */
/*====================================================================*/

#endif /* __SLIP_CONF_H__ */
```

NAME

intro -- Introduction to Section 3: Configuration Tables

DESCRIPTION

The pSOSystem components configure themselves at startup based on information contained in a collection of user-supplied Configuration Tables. These tables contain parameters that characterize the hardware and application environment.

The pSOSystem software contains functions that build all configuration tables for you. A user supplied file called sys_conf.h is used for this purpose.This file contains **#define** statements for the various parameters needed to construct the configuration tables. The “Configuration and Startup” chapter in *pSOSystem Getting Started* details the use of the **sys_conf.h** file. Also examples of its use appear in all the sample applications. This section explains the configuration tables on a more basic level for those who want to build their own configuration tables or just want more detailed information on it. It describes the structures for the following Configuration Tables:

- Node (See page 3-3)
- Multiprocessor (See page 3-5)
- pSOS+ (See page 3-9)
- pROBE+ (See page 3-15)
- pHILE+ (See page 3-25)
- pREPC+ (See page 3-29)
- pNA+ (See page 3-31)
- pRPC+ (See page 3-39)

The structure definitions for these configuration tables reside in the **include/configs.h** directory.

The Configuration Tables can be located anywhere in memory. pSOSystem locates the tables via a *Node Configuration Table*, from which a set of pointers fans out to the individual component configuration tables.

The Node Configuration Table can also be located anywhere in memory; it is located via the *Node Anchor*, which is the one fixed point of reference. The Node Anchor exists to enable any component to locate the Node Configuration Table, and subsequently the individual configuration tables. Figure 4 on page 3-2 shows the relationships between the Node Anchor and the various tables.

pSOSystem components expect the Node Anchor to be set up at memory address 0x44. It may be moved, if necessary, by making a patch within each individual component.

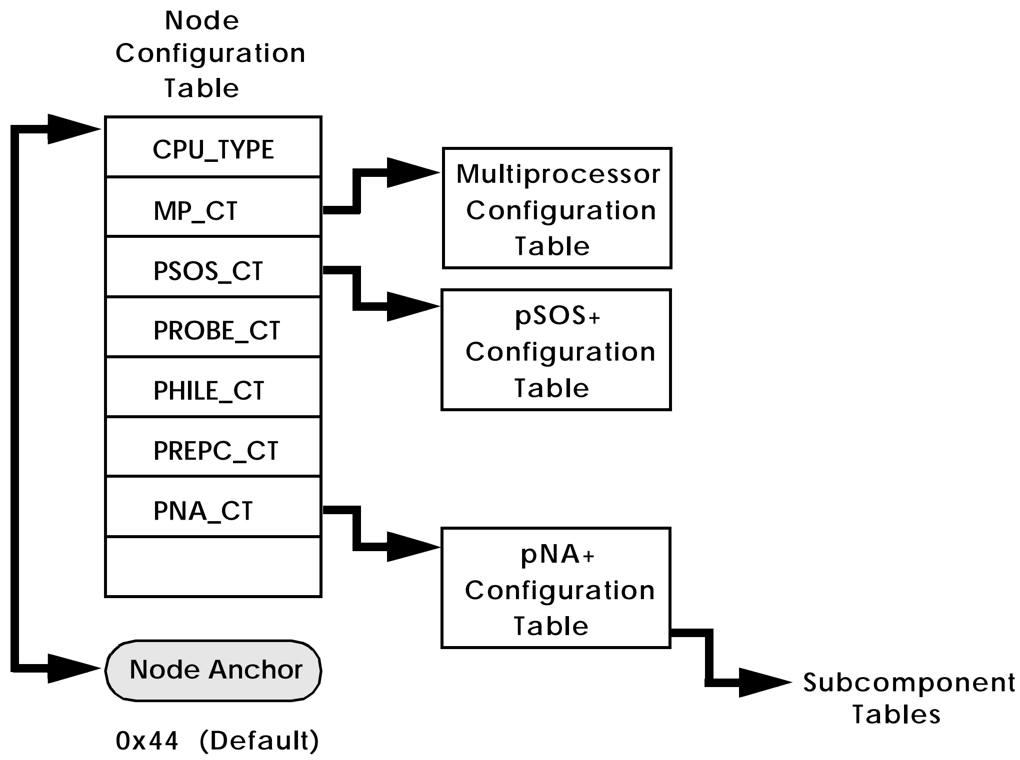

Figure 4 Configuration Tables

NAME

Node -- Pointers to Configuration Tables of software components

SYNTAX

```
typedef struct NodeConfigTable {
    unsigned long  cputype;      /* CPU type */
    MP_CT          *mp_ct;       /* Pointer to Multiprocessor Config Table */
    pSOS_CT        *psosct;      /* Pointer to pSOS+ Configuration Table */
    pROBE_CT       *probect;     /* Pointer to pROBE+ Configuration Table */
    pHILE_CT       *philect;     /* Pointer to pHILE+ Configuration Table */
    pREPC_CT       *prepct;      /* Pointer to pREPC+ Configuration Table */
    unsigned long  rsvd1;        /* Unused entry */
    pNA_CT         *pnact;       /* Pointer to pNA+ Configuration Table */
    pSE_CT         *psect;       /* Pointer to pSE+ Configuration Table */
    pMONT_CT       *pmct;        /* Pointer to pMONT+ Configuration Table */
    unsigned long  rsvd2 [2];    /* Unused entries. */
    } NODE_CT;
```

DESCRIPTION

The Node Configuration Table is a user-supplied table that is used to locate each individual component configuration table; it contains a list of pointers, one for each component configuration table. This table can reside anywhere in RAM or ROM. The Node Anchor must contain a pointer to the location of the Node Configuration Table. The C language template for the Node Configuration Table is located in **include/configs.h**:

Definitions of the Node Configuration Table entries are as follows:

cputype CPU type and has the following meaning:

BITS	MEANING
31 - 10	Must be all 0's
9	1 = Use MMU; 0 = No MMU used
8	1 = Use FPU; 0 = No FPU used
7 - 0	Processor type, as follows: 0 = 68000, 68008, or 68302 1 = 68010 2 = 68020 3 = 68030 4 = 68040 6 = 68060 7 = 68070 33 = CPU32 family: 68332, 68340 36 = 68360

The MMU or FPU bit should be 1 only if these units exist in the system.

mp_ct Starting address of the Multiprocessor Configuration Table: it should be 0 if the system is single-processor.

probe_ct Starting address of the pROBE+ Configuration Table: it should be 0 if pROBE+ is not installed.

phile_ct Starting address of the pHILE+ Configuration Table: it should be 0 if pHILE+ is not installed.

prep_ct Starting address of the pREPC+ Configuration Table: it should be 0 if pREPC+ is not installed.

rsvd1 Reserved for future use and should be set to 0.

pna_ct Starting address of the pNA+ Configuration Table: it should be 0 if pNA+ is not installed.

pse_ct Starting address of the pSE+ Configuration Table: it should be 0 if pSE+ is not installed.

pm_ct Starting address of the pMONT+ Configuration Table: it should be 0 if pMONT+ is not installed

rsvd2[2] Reserved for future use and should be set to 0.

NAME

Multiprocessor — Hardware and application-specific parameters for multiprocessor systems

SYNTAX

```
typedef struct {
    unsigned long  mc_nodenum;       /* This node's node number */
    void  (*mc_kicode)();            /* Address of this node's kernel interface */
    unsigned long  mc_nnode;         /* Max number of nodes in system */
    unsigned long  mc_nglbobj;       /* Max num of global objects on each node */
    unsigned long  mc_nagent;        /* Max number of RSC agents in this node */
    unsigned long  mc_flags;         /* Operating mode flags */
    void  (*mc_roster)();            /* Address of roster change callout */
    void  *mc_dprext;                /* Dual-port ram external starting address */
    void  *mc_dprint;                /* Dual-port ram internal starting address */
    unsigned long  mc_dprlen;        /* Dual-port ram length in bytes */
    unsigned long  mc_kimaxbuf;      /* Max KI buffer length */
    void  (*mc_asyncerr)();          /* Asynchronous call error callout */
    unsigned long  mc_reserved[6];   /* Unused, set to 0 */
} mp_ct;
```

DESCRIPTION

The Multiprocessor Configuration Table is a user-supplied table that is used to specify hardware- and application-specific parameters in a multiprocessor system. This table can reside anywhere in RAM or ROM. The **nc_mpct** entry in the Node Configuration Table must contain the starting address of the Multiprocessor Configuration Table. The C language template for this table is located in **include/psoscfg.h**.

Parameters in this table describe characteristics, some of which are system-wide, some of which are local to the node. Some of the parameters are verified by the master node as part of the multiprocessor system startup verification procedure. Definitions for parameters in the Multiprocessor Configuration Table entries are as follows:

mc_nodenum Specifies the node number of the local node. The following rules must be observed:

- Node number 0 is reserved and must not be used.
- Node number 1 is the master node.
- The node number must be unique.
- The node number must be less than or equal to **mc_nnode**, which specifies the maximum number of nodes in the system.

mc_kicode Contains the address of the entry point for the user-supplied Kernel Interface (KI) functions. See Section 2, “Interfaces and Drivers” for detailed descriptions of the eight KI functions.

mc_nnode Specifies the maximum number of nodes in the system (must not exceed 16383). This is a maximum number. Not all nodes need to be present.

mc_nglbobj Specifies the maximum number of global objects that may be created and exported by any one node in the system. **mc_nglbobj** must be the same on every node, so it should be chosen to accommodate the node that creates the maximum number of such exported objects. **mc_nglbobj** is used during pSOS+m initialization to calculate the amount of memory that must be reserved for the Global Object Table

mc_nagent Specifies the number of agents allocated for this node. Agents operate on behalf of RSCs that have been received from other nodes in the system. In particular, if an RSC must be blocked (e.g. a **q_receive()** call from a remote node), then one agent is tied up until the RSC completes or times out. The number of agents required may vary from one node to another. In general, the more RSCs that are expected to be directed at a node, the more agents that may be needed.

Agents are described in detail in the *pSOSystem System Concepts* manual.

mc_flags Assigns values to either of two flags that control the operation of the pSOS+m kernel:

KIROSTER: If set, the pSOS+m kernel will call the **ki_roster** service whenever the node roster changes.

SEQWRAP: On a slave node, this flag determines the action taken when its sequence number reaches the maximum allowable value. If **SEQWRAP** is set, then the sequence number wraps around to 1. If clear, the node fails to restart and shuts down instead. On the master node, this bit is meaningless, since the master node may not fail.

mc_roster Contains the address of an optional user-provided routine that is used to provide roster information to the KI. If **mc_roster** is not NULL, then the pSOS+m kernel calls this routine whenever the node roster changes. The **mc_roster** routine is called with a JSR instruction and should return with an RTS instruction. **mc_roster** should preserve all register values> On entry, D1 specifies the type of roster change as follows (note that this interface is identical to that of **ki_roster**):

D1	Type of Change
0	This is the initial roster. A0points to the internal pSOS+m roster.
1	A node has joined. D2 and D3 contain, respectively, the node number and sequence number of the new node.
2	A node has failed. D2, D3, and D4 contain, respectively, the node number of the failed node, the failure code, and the node number of the node that initiated removal of the node from the system (which may be the failed node itself).

mc_dprext, **mc_dprint**, and **mc_dprlen**
Specify the local node's dual-ported memory, if any. If there is none, then all three entries must be set to 0. Note that only one contiguous dual-port memory block can be entered here for automatic address conversion by the pSOS+ kernel. See the *pSOSystem System Concepts* manual for a discussion on the use of dual-ported memory.

mc_kimaxbuf Specifies the maximum size packet buffer that the KI can allocate. Refer to Section 2, "Interfaces and Drivers," for a description of the this value. Also, note the following:

1. As explained in Section 2, "Interfaces and Drivers for most KI implementations, a value of 100 is sufficient.
2. Recall from the *pSOSystem System Concepts* manual that this value must be the same on all nodes.
3. For compatibility with previous versions of the pSOS+m kernel, a value of 0 means 100.

mc_asyncerr Contains the address of an user provided callout routine described in the *pSOSystem System Concepts* manual. If no **mc_asyncerr** is provided, this entry should be 0 (NULL), in which case the pSOS+m kernel generates a fatal error.

reserved3[6] Reserved for future use and must be set to 0.

NOTE: The parameters mc_nnode, mc_nglbobj and mc_kimaxbuf must be identical on every node in a multiprocessor configuration. pSOS+m startup validates this coherency; any discrepancy causes a fatal error.

NAME

pSOS+ -- Hardware and application-specific parameters required by pSOS+

SYNTAX

```
typedef struct pSOSConfigTable {
    void            (*kc_psoscode) ( );  /* Start address of pSOS+ */
    void            *kc_rn0sadr;         /* Region 0 start address */
    unsigned long   kc_rn0len;           /* Region 0 length */
    unsigned long   kc_rn0usize;         /* Region 0 unit size */
    unsigned long   kc_ntask;            /* Maximum number of tasks */
    unsigned long   kc_nqueue;           /* Maximum number of message queues */
    unsigned long   kc_nsema4;           /* Maximum number of semaphores */
    unsigned long   kc_nmsgbuf;          /* Maximum number of message buffers */
    unsigned long   kc_ntimer;           /* Maximum number of timers */
    unsigned long   kc_nlocobj;          /* Maximum number of local objects */
    unsigned long   kc_ticks2sec;        /* Clock tick interrupt frequency */
    unsigned long   kc_ticks2slice;      /* Time slice quantum, in ticks */
    unsigned long   kc_nio;              /* Number of I/O devices in system */
    struct pSOS_IO_Jump_Table  *kc_iojtable;      /* Address of I/O switch table */
    unsigned long   kc_sysstk;           /* pSOS+ sys. stack size (bytes) */
    void            (*kc_rootsadr) ( );  /* ROOT start address */
    unsigned long   kc_rootsstk;         /* ROOT supervisor stack size */
    unsigned long   kc_rootustk;         /* ROOT user stack size */
    unsigned long   kc_rootmode;         /* ROOT initial mode */
    void            (*kc_startco) ( );   /* Callout at task activation */
    void            (*kc_deleteco) ( );  /* Callout at task deletion */
    void            (*kc_switchco) ( );  /* Callout at task switch */
    void            (*kc_fatal) ( );     /* Fatal error handler address */
    void            (*kc_idleco) ( );    /* Idle task callout */
    void            (*kc_rtcinit) ( );   /* Real-time clock initialization callout */
    unsigned long   kc_reserved1;        /* Reserved */
    unsigned long   kc_rootpri;          /* ROOT priority */
    unsigned long   kc_reserved2[5];     /* Reserved for future use */
} pSOS_CT;
```

DESCRIPTION

The pSOS+ Configuration Table is a user-supplied table used to specify hardware and application-specific parameters required by pSOS+. This table can reside anywhere in RAM or ROM. The starting address of the pSOS+ Configuration Table must be specified as the **nc_psosct** entry in the Node Configuration Table. The C language template for the pSOS+ Configuration Table is located in **include/psoscfg.h**

The definition of the pSOS+ Configuration Table entries are as follows:

kc_psoscode Defines the starting address of pSOS+ code.

kc_rn0sadr Defines the starting address of region 0. This address must be long word aligned.

kc_rn0len Defines the length of region 0 (in bytes). The value of **kc_rn0len** depends on the values of various entries in the pSOS+ Configuration Table and, in a multi-processor configuration, some values from the Multi-processor Configuration Table. The sections of this manual that describe the memory considerations for individual processors explain how to calculate **kc_rn0len** by using these configuration table entries.

kc_rn0usize Defines the unit size (in bytes) of region 0.

kc_ntask Defines the number of Task Control Blocks (TCB) that will be statically preallocated by pSOS+ at startup. This value must accommodate the expected number of simultaneously active tasks (excluding ROOT and IDLE).

kc_nqueue Defines the number of Queue Control Blocks that will be statically preallocated by pSOS+ at startup.

kc_nsema4 Defines the number of Semaphore Control Blocks that will be statically preallocated by pSOS+ at startup.

kc_nmsgbuf Defines the number of Message Buffers that will be statically preallocated by pSOS+ at startup. If a task sends a message to a queue where no task is presently waiting, the message (4 long words) must be copied to a message buffer (5 long words, to hold the message plus a link) obtained either from the system-wide pool, or from the queue's private pool, if any. If the system or private buffer pool is temporarily exhausted, the message cannot be posted, and an error condition is returned to the message sender. Thus, **kc_nmsgbuf** should reflect the anticipated number of system message buffers needed to buffer messages under the worst operating conditions.

One fail safe method for handling worst case scenario is to always create queues with private buffers. Another method is to set length limits on all queues, and then set the sum of all queue limits as the size of the system message buffer pool.

kc_ntimer Defines the number of Timer Control Blocks that will be statically preallocated by pSOS+ at startup.

kc_nlocobj Defines the *size* of the Local Object Table for the current node. The size of the Local Object Table is specified as the number of object entries. Every task, queue, semaphore, partition, and region created on a node (but not exported) requires an entry in the Local Object Table. The size that **kc_nlocobj** represents is the sum of **kc_ntask**, **kc_nqueue**, and **kc_sema4** plus the maximum number of memory partitions and regions expected on the node (including region 0). **kc_nlocobj** may not exceed 16383.

kc_ticks2sec Defines the number of clock ticks in one second (that is, the frequency of the **tm_tick** system call).

kc_ticks2slice Defines the number of clock ticks in a timeslice. If **kc_ticks2slice** is defined to be 5 and **kc_ticks2sec** is 10, for example, then pSOS+ performs roundrobin scheduling among tasks of equal priority approximately every half-second, other circumstances permitting.

kc_iojtable Contains the starting address of an I/O Switch Table.

kc_nio Specifies the number of major devices in the system (and therefore the size of the I/O Switch Table).

kc_sysstk Specifies the size of the pSOS+ system stack. It must be large enough to accommodate the worst case, nested interrupt usage. The sections of this manual that describe processor-specific memory considerations explain how to determine **kc_sysstk**.

kc_rootsadr Starting address of the ROOT task. The next three parameters are used by pSOS+ when it internally calls **t_create** and **b_start** to create and activate the ROOT task. pSOS+ defaults the task's priority and flags to 240, local, and no FPU.

kc_rootsstk Defines the size (in bytes) of the ROOT task's supervisor stack (must be at least 128).

kc_rootustk Defines the size (in bytes) of the ROOT task's user stack.

kc_rootmode ROOT task's initial execution mode.

kc_startco Supplies the address of a user-defined, optional procedure that is called during task startup. See below for additional details.

kc_deletco Supplies the address of a user-defined, optional procedure that is called during task deletion. See below for additional details.

kc_switchco Supplies the address of a user-defined, optional procedure that is called during task context switching.

The **kc_startco**, **kc_deletco**, and **kc_switchco** pSOS+ callout procedures allow you to perform special functions at the designated points within the normal execution of pSOS+. A zero in any of the three callout entries indicates to pSOS+ that no such procedure is necessary.

When implemented, callout procedures must observe the following conventions:

(1) Upon entry, the CPU is in the supervisor state. The user procedure must not at any time cause the CPU to exit this state. In addition, the hardware mask level is typically, but not necessarily at 0. The user procedure must not drop this mask level. However, it may raise the level, provided that it also restores the original level before exiting.

(2) Upon return, all registers and the stack must be restored.

(3) Only those system calls that are allowed from ISRs are allowed from callout procedures. See the *pSOSystem System Concepts* manual for a list of such calls.

(4) **kc_startco** is called after the target task has been put into the ready state, but before any of its context has been loaded. Upon entry, register D1.L contains the task identifier **tid** and A6.L contains the Task Control Block (TCB) address of the task being started. A5.L points at the pSOS+ system data area.

(5) **kc_deletco** is called after the target task has been removed from all active-task structures, its stack segment reclaimed, and the TCB returned to the free-TCB list. Upon entry, register D1.L contains the task identifier **tid** and A6.L contains the TCB address of the task being deleted. A5.L, as usual, points at the pSOS+ system data area.

(6) **kc_switchco** is called after the context of the old running task has been completely saved, and before the context of the next task to be run is loaded. Upon entry, register D1.L contains the task id **tid** and A6.L the TCB address of the next task to run. D4.L contains the task identifier and A4.L the TCB address of the last running task. A5.L, as usual, points at the pSOS+ system data area.

Due to varying compiler procedure-linkage conventions, some of which may alter register contents, you should exercise caution if you program any callout procedure in a high-level language.

kc_fatal

Contains the address of an optional, user-specified procedure that is invoked by the pSOS+ shutdown procedure. **kc_fatal** processes fatal errors detected during pSOS+ execution; these result from several sources:

(a) Explicit **k_fatal** system calls from the user's application code;

(b) Configuration defects detected during pSOS+ startup;

(c) Certain non-recoverable run-time errors.

After a fatal error, pSOS+ consults the **kc_fatal** entry. If this entry is non-zero, pSOS+ jumps to this address. If **kc_fatal** is zero, and the pROBE+ System Debug/Analyzer is present, then pSOS+ simply passes control to the System Failure entry of pROBE+. If pROBE+ is absent, pSOS+ internally executes a divide-by-zero to cause a deliberate divide-by-zero exception. In all cases, pSOS+ pre-loads the following:

(a) D1.L with the error code; and

(b) D2.L = 0, unless the fatal error is caused by a remote **k_fatal** system call with the global bit set, in which case D2.L equals the node number of the node from which the **k_fatal** call was made. For details regarding global errors, see *pSOSystem System Calls*.

kc_idleco (68360 only). **kc_idleco** is used only for the 68360 kernel. For all other versions, this entry must be 0.

This entry supplies the starting address of the user-defined **IDLE** task. This callout procedure allows you to perform special functions when no other tasks are running in the system. A zero in this entry instructs pSOS+ to use its own default **IDLE** task.

The user-supplied **IDLE** task can operate the device in low-power standby mode by executing the **LPSTOP** instruction in a software loop. The following is an example of a user **IDLE** task:

```
MyIdle:
      LPSTOP            #$2000
      BRA.S             MyIdle
```

For more information on the **LPSTOP** instruction, see the *MC68360 Quad Integrated Communications Controller Manual*.

Upon entry to the user **IDLE** task callout, the CPU is in the supervisor state. Note that the user **IDLE** task must never return.

kc_rtcinit (68360 only). **kc_rtcinit** is used only for the 68360 kernel. For all other versions, this entry must be 0.

This entry supplies the address of the real-time clock initialization routine. At startup, pSOS+ calls this procedure, if provided. A zero indicates that no such procedure is provided. When implemented, the procedure must adhere to the following conventions:

(a) Upon entry, the CPU is in the supervisor state and the interrupt mask level is 7. The initialization routine must not lower the interrupt level.

(b) Upon entry, register A0 contains the address of the clock interrupt service routine (ISR). The callout should install this ISR at an appropriate interrupt vector.

(c) Upon return, register A5 must be restored.

On each timer interrupt, the timer ISR within pSOS+ processes the clock tick and exits through the scheduler.

kc_reserved1 Should be 0 for upward compatibility.

kc_rootpri Defines the initial priority of the ROOT task. For backward compatibility, if this entry is zero, the ROOT task is assigned a priority of 255.

kc_reserved2 Should be all zeros for upward compatibility.

NAME

pROBE+ -- Hardware and application-specific parameters required by pROBE+

SYNTAX

```
typdef struct pROBEConfigTable {
    void (*rc_probecode) ( );               /* Address of pROBE+ code module */
    void *rc_data;                          /* Addrress of pROBE+ data area */
    void (*rc_ioinit) ( );                  /* Address of IOINIT procedure */
    unsigned long(*rc_consts) ( );          /* Address of CONSTS procedure*/
    unsigned char (*rc_conin) ( );          /* Address of CONIN procedure */
    void (*rc_conout)  (unsigned char c);   /* Address of CONOUT procedure */
    unsigned long (*rc_hststs) ( );         /* Address of HSTSTS procedure */
    unsigned char (*rc_hstin) ( );          /* Address of HSTIN procedure */
    void (*rc_hstout)  (unsigned char c);   /* Address of HSTOUT procedure */
    unsigned long rc_brkopc;                /* Instruction break opcode */
    unsigned long rc_ilevel;                /* pROBE+ interrupt mask */
    unsigned long rc_flags                  /* Initial flag settings */
    unsigned long (*rc_symval) ( );         /* Address of SYMVAL procedure */
    unsigned long (*rc_valsym) ( );         /* Address of VALSYM procedure */
    unsigned long (*rc_urcom) ( );          /* Address of URCOM */
    unsigned long rc_smode;                 /* Start mode */
    unsigned long (*rc_dicode) ( );         /* Address of DI code */
    void (*rc_entry) ( );                   /* Address of ENTRY procedure */
    void (*rc_exit) ( );                    /* Address of EXIT procedure */
    unsigned long reserved[4];              /* All must be 0's (4 words long */
} pROBE_CT;
```

DESCRIPTION

The pROBE+ Configuration Table is a user-supplied table used to specify hardware and application-specific parameters required by pROBE+. The table can reside anywhere in RAM or ROM. The starting address of the pROBE+ Configuration Table must be specified as the **nc_probect** entry in the Node Configuration Table. The C language template for the pROBE+ Configuration Table is located in **include/probecfg.h**:

Definitions for the pROBE+ Configuration Table entries are as follows:

rc_probecode Contains the starting address of the pROBE+ code.

rc_data Define an area in RAM that pROBE+ uses for a data area and a stack. pROBE+ requires 5 Kbytes for static data structures and *at least* 1.5 Kbytes for its stack. **rc_data** points to the start of the 5-Kbyte data area, with the memory below reserved for the stack. Figure 5 illustrates the pROBE+ Data Area Organization.

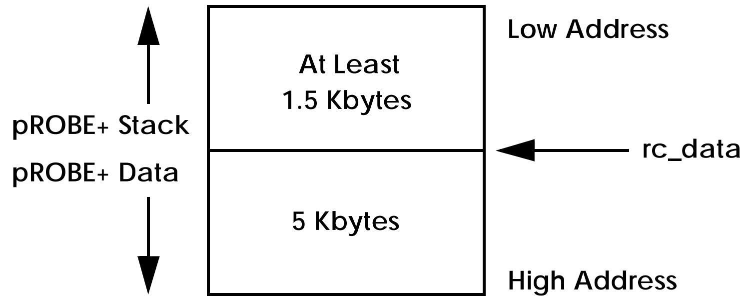

Figure 5 pROBE+ Data Area Organization

If interrupt activity can occur while pROBE+ is running, the stack size must be large enough to accommodate the worst-case stack requirements for all nesting ISRs. This is true even on CPUs with a Master Stack, because pROBE+ executes with the MASTER bit OFF, so that it always uses the interrupt stack.

rc_ioinit Points to a user-supplied initialization procedure (IOINIT), which is called once by pROBE+ during startup to initialize console of pROBE+ and host communication ports. IOINIT has no input or output parameters.

rc_consts Points to a user-defined console status procedure (CONSTS). pROBE+ calls CONSTS when it wants to determine the input status of the console port. The interface to this procedure is:

INPUT:

D0 = pROBE+ is polling for:

0 = any console input

1 = BREAK, CTRL-S or CTRL-Q only

2 = BREAK only

OUTPUT:

D0 = console status:

0 = no character received

1 = a character received

2 = BREAK detected

On input, pROBE+ indicates to your driver the type of information for which it is polling. In installations where the port is connected to a character-oriented device, you can ignore this parameter. It is useful only in special configurations, notably those where the I/O device is line or block-oriented.

On output, status = 2 has priority over status = 1, and should be reported first when both a character and a break has been detected.

BREAK, known elsewhere as a console-induced break, is user-implementation dependent. You can define it to be a true RS-232 break-detect. Or, alternatively, you can use a special character (e.g. CTRL-C). True RS-232 breaks require special treatment, since terminals normally generate break pulses with a duration of 250 ms or longer. Your driver must wait long enough for the pulse to end, before returning status = 2. Otherwise, pROBE+ will probably call CONSTS again, only to see the same break.

CONSTS should only return status = 2 once for each break detected. However, status = 1 should be latched (i.e. not cleared) until the received character is read when pROBE+ calls CONIN.

CONSTS, with input D0 = 2, is regularly called during execution of a pSOS+ application to poll for a manual break. You should, at least for this case, code CONSTS to execute as quickly as possible so that intrusion on the application is minimized.

rc_conin Points to a user-supplied console input procedure (CONIN). pROBE+ calls CONIN when it wants to read a character from the console, but only after receiving a status of 1 from CONSTS. The interface to this procedure is:

INPUT:

none

OUTPUT:

D0.B = character entered on console

rc_conout Points to a user-supplied console output procedure (CONOUT). pROBE+ calls CONOUT when it wants to send a character to the console. The interface to this procedure is:

INPUT:

D0.B = character to print

OUTPUT:

none

CONOUT should wait for the output channel to become ready, send the character, and return immediately to pROBE+ without waiting for the transmission to complete. Implementations that send the character and then wait for completion of transmission before returning may suffer dropped characters under certain operating conditions.

rc_hststs Points to a user-supplied host status procedure (HSTSTS) if the host port is used; otherwise, it can be zero. pROBE+ calls HSTSTS when it wants to determine the input status of the host port. The interface to this procedure is:

INPUT:

none

OUTPUT:

D0.B = host channel status:

0 = no character received

1 = character received

Status = 1 should be latched (i.e. not cleared), until the character has been read by pROBE+ (using HSTIN).

This procedure is used by the DL, VL, UL, and HO commands if the HOST flag is set, and by all commands if the ECHO and HOST flags are both set. If you do not need a host link, set this entry to zero. Also, it is possible for the console and host ports to share a single I/O connection. In such cases,

you can set the **rc_flags** HOST bit to zero, or alternatively set **rc_consts** and **rc_hststs** to point to the same driver procedure.

rc_hstin Points to a user-supplied host input procedure (HSTIN) if the host port is used. pROBE+ calls HSTIN when it wants to read a character from the host port, but only after receiving a status = 1 from HSTSTS. The interface to this procedure is:

INPUT:

none

OUTPUT:

D0.B = character received

This procedure is used by the DL, VL, UL, and HO commands if the HOST flag is set, and by all commands if the ECHO and HOST flags are both set. If you do not need a host link, set this entry to zero. Also, it is possible for the console and host ports to share a single I/O connection. In such cases, you can set the **rc_flag** HOST bit to zero, or alternatively set **rc_conin** and **rc_hstin** to point to the same driver procedure.

rc_hstout Points to a user-supplied output procedure (HSTOUT) if the host port is used. pROBE+ calls HSTOUT when it wants to send a character to the host port. The interface to this procedure is:

INPUT:

D0.B = character to send

OUTPUT:

none

This procedure is used by the DL, VL, UL, and HO commands if the HOST flag is set, and by all commands if the ECHO and HOST flags are both set. If you do not need a host link, set this entry to zero. Also, it is possible for the console and host ports to share a single I/O connection. In such cases, you can set the **rc_flag** HOST bit to zero, or alternatively set **rc_conout** and **rc_hstout** to point to the same driver procedure.

Like CONOUT, HSTOUT should wait for the channel to become ready, send the character, and return immediately to pROBE+ without waiting for the transmission to complete. Implementations that send the character and then wait for completion of transmission before returning may suffer dropped characters under certain operating conditions.

rc_brkopc Must be zero in the high half and contain a two-byte opcode - the breakpoint opcode - in the low half of this long-word entry, which pROBE+ will use to implement instruction breakpoints. You can use one of the following:

Preferred: A TRAP instruction - 4E4n where n = 0,1...F

Alternative: The Illegal instruction - 4AFC

Depending on the choice, you must set the corresponding exception vector to point to the address of the Breakpoint Entry of pROBE+.

rc_ilevel Specifies the initial pROBE+ interrupt level. The interrupt level should initially be 7, unless your application has special requirements. This will disable all interrupt activity (except NMI, of course) while pROBE+ is in control.

rc_flags Specifies the initial settings for the pROBE+ flags. One bit in **rc_flags** corresponds to each pROBE+ flag, as follows:

	RC_FLAGS	7 6 5 4 3 2 1 0
RBUG:	disabled	0
	enabled	1
HOST:	disabled	..........................0....
	enabled	..........................1....
TRFR:	disabled	......................0........
	enabled	......................1........
NODOTS:	disabled	..................0............
	enabled	..................1............
NOMANB:	disabled	.............0................
	enabled	.............1................
NOPAGE:	disabled	.........0.....................
	enabled	.........1.....................
ECHO:	disabled	....0..........................
	enabled	....1..........................
PROFILE:	disabled	0..............................
	enabled	1..............................

For example, if you want to initialize the NOMANB and ECHO bits to be on, set **rc_flags = 0x00000050**. The normal setting for each flag is off. Therefore, unless you specifically want one or more to come up in the on state, simply set **rc_flags = 0**. It is recommended that RBUG, NOMANB, NOPAGE, ECHO, and PROFILE be set to 0 (off). Note that all the flags can be changed interactively with the pROBE+ FL (flag) command.

rc_symval Specifies the address of an optional user-supplied procedure (SYMVAL), used to implement symbolic references within your command inputs. pROBE+ calls SYMVAL when it encounters a symbol, defined as a character string prefixed by a dollar sign. SYMVAL should translate the symbol into a 32-bit value, and return this value to pROBE+. This procedure is optional. If not used, set **rc_symval** to zero.

The interface to this procedure is as follows:

INPUT:

A0 = points to the character after the '$' within the input buffer.

OUTPUT:

A0 = updated input buffer pointer, or 0 if symbol not found.

D0 = value of symbol if found.

For example, when you enter the command

PM 3+$VAR1-10

then upon entry to SYMVAL, A0 points to the 'V' (in $VAR1) in the input buffer. SYMVAL should translate the symbol VAR1, return its value in D0, and advance A0 to point to the minus sign (after the token $VAR1). If not found, then A0 should return 0, which causes pROBE+ to abort the command and output the following error message:

`"Symbol Not Found"`

Note that the pointer returned in A0 must not be positioned past the carriage return at the end of the input line.

rc_valsym Specifies the address of an optional user-supplied procedure (VALSYM), used to implement symbolic output in pROBE+ displays. pROBE+ calls VALSYM within certain commands to translate a 32-bit value to a symbol name. This procedure is optional. If not used, set **rc_valsym** to zero.

The interface to this routine is as follows:

INPUT:

D0 = 32-bit value

OUTPUT:

A0 = pointer to a null-terminated string containing the symbol name associated with the value, or 0 if there is no matching symbol.

Note that the string returned can have any format, depending on the scope of your symbolic translation. For example, you may return `'VAR1+4'`, indicating an offset within a structure.

rc_urcom Specify the address of an optional user-supplied procedure (URCOM). pROBE+ calls URCOM whenever it encounters an unrecognized command,

allowing you to extend the command set by pROBE+. This procedure is optional. If not used, set **rc_urcom** to zero. In this case, any unrecognized command is considered an error.

The interface to this procedure is:

INPUT:

A0 = pointer to command line, as entered by user

OUTPUT:

D0 = 0 if command accepted.

D0= -1 if command unrecognized.

Any command line parsing and console output is the responsibility of this URCOM procedure. If -1 is returned, pROBE+ displays the message `"Unrecognized Command".`

rc_smode Starts pROBE+ ahead of pSOS+, and pROBE+ normally takes control and prompts you for your first command. At any point thereafter, you can use the GS command to start pSOS+. However, in an operational system, you may want to start pROBE+ but pass control to pSOS+ without waiting for a command. With this method, pROBE+ can be present, initialized and ready to run, but live in a quiescent state until an exception or manual break passes control to it. Note that in the silent mode, pROBE+ will not display its sign-on banner or prompt.

rc_smode can be used to specify the pROBE+/pSOS+ startup method, as follows:

0 = normal start-up, stops in pROBE+

1 = silent start-up, passes control to pSOS+

rc_dicode Should be set to 0 if the XRAY debugger for pSOSystem is not used. (If **rc_dicode** has a non-zero value, refer to the XRAY documentation for instructions on setting the **rc_dicode** value.)

rc_entry Specifies the address of an optional user-supplied procedure (ENTRY). pROBE+ calls ENTRY whenever it receives control via startup, breakpoint, or exception occurrence. ENTRY is not called in cases where pROBE+ has not exited, such as single stepping. This procedure is optional. If not used, set **rc_entry** to zero. The procedure you supply may be used to flush or disable off-CPU caches, or disable write-protection, or any other steps that may be needed to give pROBE+ a proper working environment.

The interface to this procedure is:

INPUT: None.

OUTPUT: None

rc_exit Specifies the address of an optional user-supplied procedure (EXIT). pROBE+ calls EXIT whenever it returns control back to user code (using the GO command, and so forth). In the case of single stepping, EXIT is not called. This procedure is optional; if not used, set **rc_exit** to zero. This procedure typically performs the opposite actions of the ENTRY procedure.

The interface to this procedure is:

INPUT: None.

OUTPUT: None.

reserved[4] Should be set to 0.

NAME

pHILE+ -- Hardware and application-specific parameters required by pHILE+

SYNTAX

```
typedef struct pHILEConfigTable{
    void (*fc_phile)();               /* Address of pHILE+ module */
    void *fc_data;                    /* Address of pHILE+ data area */
    unsigned long  fc_datasize;       /* Size of pHILE+ data area */
    unsigned long  fc_logbsize;       /* Block size (base-2 exponent) */
    unsigned long  fc_nbuf;           /* Number of cache buffers */
    unsigned long  fc_nmount;         /* Max # of mounted volumes */
    unsigned long  fc_nfcb;           /* Max # of opened files per system */
    unsigned long  fc_ncfile;         /* Max # of opened files per task */
    unsigned long  fc_ndnlc;          /* Max # of cached directory entries */
    unsigned long  fc_msdos;          /* MS-DOS volume mount flag */
    unsigned long  fc_cdrom;          /* CD_ROM volume mount flag */
    unsigned long  res[5];            /* Must be 0 */
} pHILE_CT;
```

DESCRIPTION

The pHILE+ Configuration Table is a user-supplied table that provides hardware and application-specific information required by pHILE+. It can reside anywhere in RAM or ROM. The starting address of the pHILE+ Configuration Table must be specified as the **nc_philect** entry in the Node Configuration Table. The C language template for the pROBE+ Configuration Table is located in **include/configs.h**:

Definitions for the pHILE+ Configuration Table entries are as follows:

fc_phile Defines the starting address of the pHILE+ code.

fc_data Defines the starting address of the pHILE+ data area, which must be located in RAM. You must reserve enough space for the memory requirements of pHILE+.

If the **fc_data** and **fc_datasize** entries are both 0, pHILE+ automatically allocates the required amount of memory from pSOS+ region 0 during initialization. In this case, **fc_data** is ignored. pHILE+ calculates the amount of memory it requires by examining entries in its Configuration Table.

fc_datasize Defines the size of the pHILE+ data area. The value of **fc_datasize** depends on various pHILE+ Configuration Table entries.

fc_logbsize Defines the system-wide block size for pHILE+ formatted volumes. Note that this parameter is specified as a base 2 exponent. For example, if the desired block size is one Kbyte, **fc_logbsize** is 10. The range for **fc_logbsize**

8 through 15 (the smallest permissible block size is 256 bytes, and the largest is 32 Kbytes).

A block is the basic unit of data transfer and all I/O requests are made by pHILE+ in terms of blocks. Therefore, its size should be selected carefully. To avoid unnecessary blocking/deblocking by device drivers, **fc_logbsize** should be equal to or greater than the natural sector or cluster size of a physical device. Within limits, a larger block size will improve throughput. On the other hand, a larger block size requires more memory for buffering purposes and can waste disk space. In the absence of other considerations, a block size of 512 bytes or 1 Kbyte appears reasonable and is thus recommended.

fc_logbsize has no effect on NFS volumes. MS-DOS volumes always have a 512-byte block size.

fc_nbuf Defines the number of cache buffers used by pHILE+. The size of each buffer is defined by **fc_logbsize** but will not be less than 512 if MS-DOS volumes are in use (**fc_msdos = 1**). A minimum of two cache buffers is required for proper operation of pHILE+.

This value is the single most influential parameter with respect to optimizing overall file system throughput. With few exceptions, file data transfers always go through the buffer cache. Therefore, the larger the number of cache buffers, the more likely that a read or write request will find its data *lingering* in a cache buffer, thus obviating the need to execute a physical read operation. Increasing the number of buffers will directly improve the throughput of the file system.

Experimentally determine the optimum number of cache buffers for an application. In applications where file throughput is important, one approach might be to allocate to the cache as much memory as can be spared. Note that cache buffers are not used with NFS volumes.

fc_nmount Specifies the maximum number of volumes that can be mounted simultaneously. It defines the number of entries in the mounted volume table.

fc_nfcb Defines the maximum number of files that can be open simultaneously; it is used to allocate space for file control blocks (FCBs).

Note that this parameter should not be confused with the number of open files attached to each task. In particular, each FCB may be connected to one or more tasks.

fc_ncfile Defines the maximum number of simultaneously open files that a task can have. It determines the number of entries in each task's open file table.

fc_msdos Indicates the intention to mount MS-DOS volumes. A 0 means no MS-DOS volumes will be mounted. A 1 means MS-DOS volumes will be mounted, and pHILE+ will ensure that cache buffers are at least 512 bytes long,

fc_cdrom Indicates the intention to mount CD-ROM volumes. A 0 means no CD-ROM volumes will be mounted. A 1 indicates that CD-ROM volumes will be mounted and pHILE+ will ensure that cache buffers are at least 2048 bytes long.

reserved Should be 0 for upward compatibility.

NAME

pREPC+ -- Hardware and application-specific parameters required by pREPC+

SYNTAX

```
typedef struct pREPCConfigTable{
    void (*lc_code) ( );            /* Start address of pREPC+ code */
    void *lc_data;                  /* Start address of pREPC+ data area */
    unsigned long lc_datasize;      /* Size of pREPC+ data area */
    unsigned long lc_bufsize;       /* I/O buffer size */
    unsigned long reserved1;        /* Reserved entry; must be 0 */
    unsigned long lc_numfiles;      /* Maximum number of open files per task */
    unsigned long lc_waitopt;       /* Wait option for memory allocation */
    unsigned long lc_timeopt;       /* Timeout option for memory allocation */
    char *lc_tempdir;               /* Pointer to temporary file directory */
    char *lc_stdin;                 /* Pointer to stdin */
    char *lc_stdout;                /* Pointer to stdout */
    char *lc_stderr;                /* Pointer to stderr */
    unsigned long lc_ssize;         /* Size of print buffer */
    unsigned long reserved[3];      /* Reserved, must be zero */
}; pREPC_CT;
```

DESCRIPTION

The pREPC+ Configuration Table is a user-supplied table that provides hardware and application-specific information required by pREPC+. The table can reside anywhere in RAM or ROM. The starting address of the table must be specified as the **nc_prepct** entry in the Node Configuration Table. The C language template for the pREPC+ Configuration Table is located in **include/prepccfg.h**.

Definitions for the pREPC+ Configuration Table entries are as follows:

lc_code Defines the starting address of the pREPC+ code.

lc_data Defines the starting address of the pREPC+ data area, which must be located in RAM. You must reserve enough space for the memory requirements of pREPC+.

If **lc_data** and **lc_datasize** are both 0, pREPC+ automatically allocates the required amount of memory for its data area from pSOS+ region 0 during initialization. In this case, **lc_data** is ignored. pREPC+ calculates the amount of memory it requires by examining entries in its Configuration Table.

lc_datasize Defines the length of the data area. If **lc_data** is non-zero, **lc_datasize** should be 256. The value of **lc_datasize** depends on various pREPC+ Configuration Table entries.

lc_bufsize Specifies the size of the buffers allocated for open files.

lc_numfiles Defines the maximum number of files that a task can have open at the same time. (This number excludes **stdin**, **stdout**, and **stderr**.) This entry determines the number of file control blocks that pREPC+ allocates.

lc_waitopt Input to **rn_getseg** when pREPC+ calls pSOS+ to allocate memory. If **lc_waitopt** is 0 and a request is not satisfied, the caller is blocked until the first of two possible conditions occurs. The caller remains blocked until either a segment is allocated or a timeout occurs (if **lc_timeopt** is non-zero). If **lc_waitopt** is 1, **rn_getseg** returns unconditionally.

lc_timeopt Clock tick count that is input to **rn_getseg** when pREPC+ calls pSOS+ to allocate memory. It is relevant only if **lc_waitopt** is 0.

lc_tempdir Supplies the address of a string that names a file directory. If pHILE+ is not in the system or if the **tmpfile()** function is not used, this entry should point to a NULL string.

lc_stdin Supplies the address of a string that contains the pathname for **stdin**. It is opened automatically for every task that issues a pREPC+ system call. **stdin** can be an I/O device or disk file. If **stdin** cannot be opened, a fatal error results.

lc_stdout Supplies the address of a string that contains the pathname for **stdout**. It is opened automatically for every task that issues a pREPC+ system call. **stdout** can be I/O devices or disk files. If **stdout** cannot be opened, a fatal error results.

lc_stderr Supplies the address of a string that contains the pathname for **stderr**. It is opened automatically for every task that issues a pREPC+ system call. **stderr** can be an I/O device or disk file. If **stderr** cannot be opened, a fatal error results.

lc_ssize Defines the size of the temporary print buffer required by “printing” functions for storing characters generated by pREPC+ output functions (for example, **printf**). The size of the print buffer determines the maximum length of the output string generated for each conversion. If a converted string is longer than **lc_ssize**, it will be truncated.

For each task, a buffer is allocated from pSOS+ region 0 the first time it is needed. Once allocated, a buffer is held indefinitely by the task. An **fclose(0)** can be used to free the buffer. A single task never holds or uses more than one buffer.

If many tasks do printing, this parameter may have a significant impact on the amount of RAM used by pREPC+. If **lc_ssize** is 0, a default value of 512 is used. This conforms to the ANSI standard maximum length output string; however, most applications can use a much smaller value.

reserved Should be 0 for upward compatibility.

NAME

pNA+ -- Hardware and application-specific parameters required by pNA+

SYNTAX

```
typedef struct pNAConfigTable {
    void (*nc_pna) ( );               /* Address of pNA+ code module */
    void *nc_data;                    /* Address of pNA+ data area */
    long nc_datasize;                 /* Size of pNA+ data area */
    long nc_nni;                      /* Size of pNA+ NI Table */
    struct ni_init *nc_ini;           /* Pointer to Initial pNA+ NI Table */
    long nc_nroute;                   /* Size of pNA+ Routing Table */
    struct route *nc_iroute;          /* Pointer to Initial pNA+ Routing Table */
    long nc_defgn;                    /* Address of default gate node */
    long nc_narp;                     /* Size of pNA+ ARP Table */
    struct arp *nc_iarp;              /* Pointer to Initial pNA+ ARP Table */
    void (*nc_signal) ( );            /* Pointer to signal handling routine */
    long nc_defuid;                   /* Default user ID of a task */
    long nc_defgid;                   /* Default group ID of a task */
    char *nc_hostname;                /* Hostname of the node */
    long nc_nhentry;                  /* Number of Host Table entries*/
    struct htentry *nc_ihtab;         /* Pointer to Initial Host Table */
    pNA_SCT *nc_sct;                  /* Address of pNA+ subcomponent config. table */
    long nc_mblks;                    /* Number of mblks*/
    struct pna_bufcfg *nc_bcfg;       /* Pointer to buffer configuration table */
    long nc_nsockets;                 /* Number of sockets*/
    long nc_ndescs;                   /* Number of descriptors per task*/
    unsigned long nc_nmc_socs;        /* Number of multicast sockets*/
    unsigned long nc_nmc_memb;        /* Number of multicast group memberships*/
    unsigned long nc_nnode_id;        /* Network node ID or router ID*/
    long reserved[3];                 /* Reserved for future use */
} pNA_CT;
```

DESCRIPTION

The pNA+ Configuration Table is a user-supplied table that provides hardware and application-specific information required by pNA+. The table can reside anywhere in RAM or ROM. The starting address of the pNA+ Configuration Table must be specified as the **nc_pnact** entry in the Node Configuration Table. The C language template for the pNA+ Configuration Table is located in **include/pnacfg.h**.

Definitions for the pNA+ Configuration Table entries are as follows:

nc_pna Defines the starting address of pNA+ code.

nc_data Defines the starting address of the pNA+ data area, which must be located in RAM. You must reserve enough space for the memory requirements of pNA+.

If **nc_data** and **nc_datasize** are both 0, pNA+ automatically allocates the required amount of memory for its data area from pSOS+ Region 0 during initialization. In this case, **nc_data** is ignored. pNA+ calculates the amount of memory required by examining its Configuration Table entries.

Note that if pNA+ is used by pROBE+ to communicate with the XRAY debugger for pSOSystem, **nc_data** must be used to specify a pNA+ data area, and **nc_datasize** must be non-zero.

nc_datasize Defines the size of the pNA+ data area. The value of **nc_datasize** depends on various pNA+ Configuration Table entries.

nc_nni Specifies the maximum number of Network Interfaces (NIs) to be installed in your system (that is, the maximum number of networks connected to pNA+). This entry is used by pNA+ to define the size of its NI Table.

nc_ini Should point to an Initial Network Interface (NI) Table, which defines the characteristics of the network interfaces that are initially installed in your system. The contents of the Initial NI Table will be copied to the actual NI Table during pNA+ initialization. Note that the Initial NI Table may be smaller than the actual NI Table. In other words, the Initial NI Table may have less than **nc_nni** interfaces defined. This is possible because network interfaces may be added dynamically after pNA+ has been started, using the **add_ni** system call. Of course, it can never have more.

The Initial NI Table contains a set of eight 32-bit entries for each initially installed network interface. The table must be terminated by a 0. The **ni_init** structure is defined in the file **include/pna.h**. A template for one entry in the table is as follows:

```
struct ni_init{
    int (*entry)();   /* Addr of NI entry point */
    int ipadd;        /* Internet addr of the NI */
    int mtu;          /* MaxIMUM transmission unit */
    int hwalen;       /* Length of hardware address */
    int flags;        /* Defines NI flags */
    int subnetaddr;   /* Netmask */
    int dstipaddr;    /* Destination network address */
    int reserved[1];  /* Reserved for future use */
};
```

where

entry Defines the address of the NI driver's entry point.

ipadd Defines the internet address assigned to the network interface.

mtu Specifies the maximum transmission unit for the NI (minimum 64).

hwalen Specifies the length of the NI hardware address in bytes (maximum 14).

flags Specifies the initial setting of the NI flags, as follows (all unlisted bits must be 0):

Flag	Bit	Meaning
BROADCAST	0:	0 = Disabled 1 = Enabled
ARP	1:	0 = Enabled 1 = Disabled
POINTTOPOINT	4:	0 = Disabled 1 = Enabled
MULTICAST	11:	0 = Disabled 1 = Enabled
UNNUMBERED	12:	0 = Disabled 1 = Enabled
RAWMEM	13:	0 = Disabled 1 = Enabled
EXTLOOPBCK	14:	0 = Disabled 1 = Enabled
POLL	15:	0 = Disabled 1 = Enabled

subnetaddr Defines the netmask (the netmask consists of the bits in the internet address that should be included when extracting the network identifier from an internet address).

dstipaddr Defines the IP address of the host on the other side of a point-to-point network.

reserved Must be 0.

nc_nroute Determines the amount of memory required for the Routing Table. It should be set equal to 1 plus the number of network interfaces planned for the system, plus the number of additional user-supplied routes. In other words, the following formula can be used to calculate the value of **nc_nroute**: (**nc_nroute = 1** + **nc_nni** + User Supplied Routes). The User

Supplied Routes can be supplied by the Initial Routing Table (see **nc_iroute**), or by an **ioctl()** system call.

nc_iroute Should point to the Initial Routing Table (if one exists). pNA+ copies the contents of the Initial Routing Table to the actual Routing Table during initialization. If no routes are to be supplied during initialization, **nc_iroute** should be 0. It is possible to add routes dynamically after pNA+ has been started, using the **ioctl()** system call.

The Initial Routing Table contains a set of four 32-bit variables for each route. The table is terminated by a 0.

The following is a template for one entry in the Initial Routing Table:

```
struct route{
    unsigned long nwipadd;/* Host or Network address */
    unsigned long gwipadd;/* Gateway internet address */
    unsigned long flags;   /* Route type */
    unsigned long netmask;/* Subnet mask use */
};
```

where

nwipadd Specifies an IP address of the destination.

gwipadd Defines the internet address of a gateway node that should be used to route packets to the destination given by **nwipadd**.

flags Specifies the type of route (which can be the value of either **rt_host** or **rt_network** defined in the **pna.h** file).

netmask Specifies the subnet mask associated with the route. This field is ignored if the RT_MASK flag is not set in **flags**.

If the number of Initial Routing Table entries is greater than the number specified by **nc_nroute**, a fatal error occurs during pNA+ initialization.

nc_defgn Specifies the internet address of a default gateway node (if one is used). The **nc_defgn** entry should be 0 if no default gateway exists on the system.

nc_narp Determines the amount of memory required for the ARP Table. **nc_narp** must be at least 1 plus the number of network interfaces planned for the system.

nc_iarp Should point to the initial ARP Table (if one is supplied). pNA+ copies the contents of the Initial ARP Table to the actual ARP Table during initialization. **nc_iarp** can be 0 if no Initial ARP Table is supplied.

The Initial ARP Table contains four 32-bit entries to support each internet address-to-hardware address mapping. A template for one entry in the Initial ARP Table is as follows:

```
struct arp {
           int arp_ipadd;     /* Internet addr for NI */
           char *arp_hadd;    /* Hardware addr for NI */
           int reserved[2];   /* Reserved for future use */
           };
```

where

arp_ipadd Specifies the internet address of a network interface.

arp_hadd Supplies the address of the corresponding hardware address for that NI.

reserved Values must be 0.

One question that arises is how to determine the size of the ARP Table. Unfortunately, there is no definitive answer. The larger the table, the more memory is consumed, but the better the performance. If pNA+ does not find an <IP address, hardware address> tuple in the table, it must execute ARP, which takes time and creates network traffic. This suggests that the size of the table should be equal to the number of nodes with which pNA+ will communicate.

Of course, this has to be balanced against memory consumption (that is, the table takes space). It may not be necessary to have one entry for every other node on a network, if your application rarely communicates with every node. However, the number of ARP entries should at least be equal to **1** + **nc_nni**.

nc_signal Contains the address of the user signal handler, if provided. This entry should be 0 if no handler is present.

When implemented, the handler must observe the following conventions:

1) Upon entry, the CPU is in the supervisor state. The handler must not at any time cause the CPU to exit this state.

2) Upon return, all registers and the stack must be restored.

3) Only pSOS+ system calls that are allowed from ISRs are allowed.

4) Upon entry, the stack is setup as follows:

stack ptr + 0	return address
+ 4	signals number
+ 8	tid
+ 12	socket descriptor

NOTE: Due to varying compiler procedure-linkage conventions, some of which may alter register contents, exercise caution if programming your signal handler in C.

nc_defuid Defines the user ID. This ID is assigned to a task upon the task's creation. Every task that uses NFS services must have a user ID. An NFS server uses this value to recognize a client task and either grant or deny services based on its identity. These default values may be changed by the set_id system call. If pHILE+ NFS services are not used, **nc_defuid** can be 0.

nc_defgid Defines the group ID. Every task that uses NFS services must have a group ID. An NFS server uses this value to recognize a client task and either grant or deny services based on its identity. These default values may be changed by the set_id system call. If pHILE+ NFS services are not used, **nc_defgid** can be 0.

nc_hostname Points to a null terminated string that contains the hostname for the node. The maximum length for the hostname is 32 characters (including the terminating null character). The **nc_hostname** value can be 0, in which case a null hostname is used.

nc_nhentry Determines the amount of memory required for the Host Table. **nc_nhentry** must be at least the number of hostname-to-IP address mappings installed in the system.

nc_ihtab Points to the Initial Host Table (if supplied). pNA+ copies the contents of the Initial Host Table to the actual Host Table during initialization. If no Initial Host Table is present, **nc_ihtab** can be 0. The Initial Host Table contains four 32-bit variables for each hostname-to-IP address mapping. The following is a template for the Initial Host Table:

```
struct hentry{
        long ipadd;       /* IP address of host */
        char *hname;      /* Hostname */
        long reserved[2]; /* Reserved for future use */
        };
```

where

ipadd Specifies the internet address of the host associated with the **hname** field.

hname Character pointer to a null terminated string specifying the host name (maximum 32 bytes).

reserved Are each 0.

nc_sct Points to a table that contains pointers to configuration tables for pNA+ subcomponents. The table is defined as follows:

```
typedef struct{
          pXLIB_CT *px_cfg;          /* pX11+ Cfg. Table */
          struct nr_cfg *nr_cfg.;    /* pRPC+ Cfg. Table */
          long reserved[6];          /* for future use */
          } pNA_SCT;
```

where

px_cfg Points to the pX11+ Configuration Table.

nr_cfg Points to the pRPC+ Configuration Table.

reserved Entries should be 0.

nc_nmblks Defines the number of mblks configured in the system.

nc_bcfg Pointer to the buffer configuration table, which contains entries that define the data buffers configured in pNA+. Each entry contains four 32-bit variables describing the characteristics of a buffer. The table is zero terminated.

The structure of each buffer configuration table entry is as follows:

```
struct pna_bcfg
       {
       unsigned long pna_nbuffers; /* Number of buffers */
       unsigned long pna_bsize;    /* Size of buffer */
       unsigned long reserved[2];  /* Reserved entries */
       };
```

pna_nbuffers Defines the number of data buffers in the system.

pna_bsize Defines the size of the data buffers to be configured.

reserved Entries should be 0; this table is zero terminated.

nc_nsockets Defines the maximum number of sockets configured in the system.

nc_ndescs Defines the maximum number of socket descriptors per task.

nc_nmc_socs Specifies the number of sockets that may be used for multicast IP. This does not allocate new sockets in addition to **nc_nsockets**.

nc_nmc_memb Specifies the total number of distinct multicast IP group memberships that can be added in the system. A maximum of 20 distinct group memberships (an internal constant) can be added per multicast socket. Adding an existing group membership address on the same interface is counted as one membership, except that the reference count is incremented.

nc_nnode_id Defines the Network node ID or the Router ID. This is required when configuring unnumbered links in the system. It could be set to one of the IP addresses of the node.

reserved (bytes at the end of the pNA+ Configuration Table) Should all be 0.

NAME

pRPC+ -- Hardware and application-specific parameters required by pRPC+

SYNTAX

```
typedef struct nr_cfg {
    void (*nr_code) ( );          /* pRPC+ code address */
    char *nr_data;                /* Address of pRPC+ data area */
    long nr_datasize;             /* Length of pRPC+ data area */
    long reserved[10];            /* Reserved entries of pRPC+ */
    } pRPC_CT;
```

DESCRIPTION

The pRPC+ Configuration Table is a user-supplied table that provides hardware and application-specific information required by pRPC+. The table can reside anywhere in RAM or ROM. The starting address of the pRPC+ Configuration Table must be specified as the **nr_cfg** entry in the pNA+ Subcomponent Configuration Table. (Refer also to the pNA+ Configuration Table in this manual.) The C language template for the pRPC+ Configuration Table is located in **include/prpccfg.h**.

The meaning of this table's entries are as follows:

nr_code Contains the starting address of the pRPC+ code.

nr_data Defines the starting address of the pRPC+ data area, which must be located in RAM. You must reserve enough space for the memory requirements of pRPC+.

If **nr_data** and **nr_datasize** are both 0, pRPC+ automatically allocates the required amount of memory for its data area from pSOS+ region 0 during initialization. In this case, **nr_data** is ignored. pRPC+ calculates the amount of memory it requires by examining entries in its Configuration Table.

nr_datasize Defines the size of the pRPC+ data area. The current value for **nr_datasize** is fixed at 2 Kbytes.

reserved Should all be 0 for upward compatibility.

NAME

intro -- Introduction to Section 4: Memory Usage

DESCRIPTION

The amount of RAM required by each pSOSystem software component depends on the user's application. This section provides formulas for calculating these requirements based on application parameters. The following components are discussed:

- pSOS+ Real-Time Kernel (See page 4-3)
- pHILE+ File System Manager (See page 4-7)
- pREPC+ Run-Time C Library (See page 4-9)
- pNA+ TCP/IP Network Manager (See page 4-11)
- pRPC+ Remote Procedure Call Library (See page 4-15)

NAME

pSOS+ -- RAM requirements

DESCRIPTION

pSOS+ needs RAM for the following elements:

- Data Area
- Task and System Stacks
- Region Header Memory
- Partition Header Memory
- TCB Extensions
- Variable Length Queue Message Storage

Data Area

pSOS+ uses the beginning of the user-defined memory Region 0 to build its data area. The size of this data area is calculated as the sum of the items in the table that follows. In the Size column of this table, the parameters that begin with **kc** and **mc** are entries in the pSOS+ Configuration Table and the Multiprocessor Configuration Table, respectively:

Usage	Size (bytes, decimal)
Internal Variables	3500
System stack	**kc_sysstk**
Task Control Blocks (TCBs)	(**kc_ntask** + 2) x 348
Queue Control Blocks (QCBs)	**kc_nqueue** x 76
Semaphore Control Blocks (SCBs)	**kc_nsema4** x 36
Message Buffers	**kc_nmsgbuf** x 20
Timer Control Blocks (TMCBs)	**kc_ntimer** x 48
Local Object Table	(**kc_nlocobj** x 32) + 84
Global Object Table (Master Node)	**mc_nnode** x ((**mc_nglbobj** x 32) + 84)
Global Object Table (Slave Node)	(**mc_nglbobj** x 32) + 84
Agents	**mc_nagent** x 56
IO Devices	**kc_nio** x 32

If Region 0 is not large enough to contain the pSOS+ data area and the Region 0 header, a fatal error occurs during pSOS+ startup. To accommodate future expansion, it is recommended that the pSOS+ data area be padded with an extra 20% of space.

Task and System Stacks

Every task must have a supervisor stack. A task must also have a user stack if it executes in the user state (otherwise, it fails). Furthermore, pSOS+ must have its own system stack. The CPU automatically selects and uses the right stack, and memory for all stacks is allocated from Region 0. The sizes of a task's supervisor and user stacks are defined by parameters passed in the **t_create()** system call. The size of the system stack is defined in the pSOS+ Configuration Table. The following paragraphs describe issues that must be considered when sizing these stacks.

A task's user stack is used exclusively by any task code that executes in the user state. Sizing this task requires a determination of the worst case, nested procedure stack usage.

Sizing a task's supervisor stack is more involved because both pSOS+ and interrupts can use it. Any of the following can use a task's supervisor stack:

- The task's code (including its ASR), if the task runs in the supervisor state.
- pSOS+, if the task makes pSOS+ system calls (The worst case use within any pSOS+ system call is 72 bytes.)
- Device drivers, if the task makes pSOS+ I/O calls (The worst case use within an I/O call is 68 bytes plus the additional use within the user's drivers.)
- ISRs on 68000/68010/68060 systems.

On the 68000/68010/68060, since interrupt activities use the supervisor stack of the running task, every task's supervisor stack must be large enough to accommodate worst case interrupt usage. This usage must include the nesting of all possible interrupt levels. If an ISR makes a pSOS+ system call, pSOS+ usage of the stack must also be considered.

On the 68020/68030/68040, ISRs automatically use a separate interrupt stack, which is also the system stack, so duplication of stack space does not occur. Therefore, each task's supervisor stack must be able to accommodate only the first three stack uses shown in the preceding list.

On the 68000/68010/68060, the system stack is used only briefly, and a size of 72 bytes is adequate.

As previously stated, on the 68020/68030/68040, the system stack is used as the interrupt stack. It must be large enough to accommodate the worst case interrupt usage, and this usage must include the nesting of all possible interrupt levels. If an ISR makes a pSOS+ system call, pSOS+'s usage must also be considered.

Region Header Memory Usage

When a region is created, some memory for its management is reserved at the beginning of the region memory. This memory space is the Region Header. The size of a Region Header is computed from the following formula:

70 + ((*length/ unit_size*) x 6) bytes, rounded up to the next even multiple of *unit_size*

where *length* and *unit_size* are parameters to the **rn_create()** call.

In addition to regions in general, this formula is also valid for Region 0 if *length* has the value of the pSOS+ Configuration Table entry **kc_r0len** minus the memory requirements of the pSOS+ Data Area. For example, if **kc_r0len** is 10 Kbytes and the pSOS+ Data Area is 3 Kbytes, then *length* should be 7 Kbytes in the preceding formula to calculate the Region 0 header size.

Segments obtained from a region have no additional memory overhead.

Partition Header Memory Usage

A Partition Header is the memory reserved in a partition for management of partition buffers. The following formula gives the amount of memory reserved for a Partition Header:

(52 + (((*length*/*bsiz*) + 7) / 8)) bytes, rounded up to the next even multiple of *bsiz* + (*length* modulo *bsiz*)

where *length* and *bsiz* are parameters passed to the **pt_create()** call, and / means integer division. Buffers allocated from a partition have no additional memory overhead.

TCB Extensions

At task creation, pSOS+ can add memory blocks called TCB extensions to the task's Task Control Block (TCB) for specific functions. Example functions of a TCB extension are to save FPU status and to support the needs of other components in the system.

If a task uses the FPU, 328 bytes are allocated for a TCB extension. The sizes of other TCB extensions appear in each component's Memory Usage section.

Variable Length Queue Message Storage

When a variable length message queue is created, pSOS+ allocates memory from Region 0 to store any messages that are pending at the queue during use. The following formula gives the amount of memory requested from Region 0:

```
maxnum x ((maxlen + 11) & -4)
```

where '&' is the bit-wise AND operator, and **maxnum** and **maxlen** are input parameters to **q_vcreate()**. No memory is allocated when either **maxnum** or **maxlen** is zero. The actual amount of memory allocated depends on the *unit size* for Region 0. The pSOS+ Configuration Table entry **kc_rn0usize** specifies the unit size for Region 0.

NAME

pHILE+ -- RAM requirements

DESCRIPTION

pHILE+ needs RAM for the following elements:

- Data Area
- Stack
- TCB Extensions

Data Area

Data area requirements for pHILE+ depend on user-supplied entries in the pHILE+ Configuration Table. The size of the data area is the sum of the values generated by incorporating the relevant configuration table entries (each of which begins with **fc**) in the following formulas:

Usage	Size in bytes
Static pHILE+ variables	636
Buffer headers	**fc_nbuf** x 48
Cache buffers	**fc_nbuf** x BUFFSIZE
Mounted volume table	**fc_nmount** x 356
File control blocks (FCB)	(**fc_nfcb** + **fc_nmount**) x 172

Refer to *pSOSystem System Concepts* to determine the buffer size. Normally it is $2^{FC_LOGBSIZE}$.

Memory for pHILE+'s data area can be allocated from Region 0, or it can be allocated from a fixed location. The location depends on the pHILE+ Configuration Table entry **fc_data**.

Stack Requirements

pHILE+ executes in supervisor mode and uses the caller's supervisor stack for temporary storage and automatic variables. pHILE+'s worst case usage of the caller's stack is fewer than 4096 bytes. Therefore, a task that uses pHILE+ should be created with at least that much stack space.

Task Extension Areas

With pHILE+ in a system, pSOS+ allocates a pHILE+ TCB extension for each task at task creation. Memory for a TCB extension comes from Region 0, and the following formula gives its size:

156 + (34 x **fc_ncfile**)

where **fc_ncfile** is the entry in the pHILE+ Configuration Table that specifies the maximum number of open files allowed per task.

NAME

pREPC+ -- RAM requirements

DESCRIPTION

pREPC+ needs RAM for the following elements:

- Data Area
- Stack
- TCB Extensions

Data Area

pREPC+ requires a fixed-size data area of 256 bytes. Memory for pREPC+'s data area can be allocated from Region 0, or it can be allocated from a fixed location. The location depends on the pREPC+ Configuration Table entry **lc_data**.

Stack Requirements

pREPC+ executes in supervisor mode and uses the caller's supervisor stack for temporary storage and automatic variables. pREPC+ requires a maximum of 1 Kbyte of stack space. The 1 Kbyte specification assumes that no more than 10 floating point numbers are passed to a formatted I/O function. If more than 10 floating point numbers must be able to pass, the size of the calling task's supervisor stack must be increased to make room for the additional arguments.

TCB Extensions

With pREPC+ in a system, pSOS+ allocates a pREPC+ TCB extension for each task at task creation. Memory for a pREPC+ TCB extension comes from Region 0, and the following formula gives its size:

184 + (28 x (**lc_numfiles** + 3)) bytes

where **lc_numfiles** is the pREPC+ Configuration Table entry for the maximum number of simultaneously open files that a task can have (excluding **stdin**, **stdout**, and **sterr**).

NAME

pNA+ -- RAM requirements

DESCRIPTION

pNA+ needs RAM for the following elements:

- Data Area and Buffers
- Stack
- TCB Extensions

Data Area and Buffer Requirements

Data area requirements for the pNA+ data area depend on user-specified entries in the pNA+ and pSOS+ Configuration Tables. The size of the data area is the sum of the values generated from the following formulas. The pNA+ Configuration Table entries begin with the letters **nc**, and **kc_ntask** is a pSOS+ Configuration Table entry. The parameters passed to **pna_init()** are **npages** and **nmbufs**.

Usage	Size in Bytes
Static pNA+ variables	4668
Network Interface Table	**nc_nni** x 120
Routing Table	**nc_nroute** x 100
ARP Table	**nc_narp** x 40
Host Table	**nc_nhentry** x 42
Socket Control Blocks	**nc_nsockets** x 152
Protocol Control Blocks	**nc_nsockets** x 68
Open Socket Tables	(**kc_ntask** + 2) x 4 x (**nc_ndescs**)
Multicast sockets	**nc_nmc_socs** x 92
Multicast memberships	**nc_nmc_memb** x 24

The sum of the following is the total memory needed for pNA+ buffer configuration:

Usage	Size in Bytes
Message Blocks (mblks)	**nc_mblks** x 24
Data Block Table	Number of different buffer sizes x 40
Nonzero-Sized Buffers	**pna_nbuffers** x **pna_bsize**
Data Blocks for Nonzero-Sized Buffers	**pna_nbuffers** x 24
Data Blocks for Zero-Sized Buffers	**pna_nbuffers** x 32

Memory for pNA+'s data area can be allocated from Region 0, or it can be allocated from a fixed location. The location depends on the pNA+ Configuration Table entry **nc_data**.

Stack Requirements

pNA+ executes in supervisor mode and uses the caller's supervisor stack for temporary storage and automatic variables. The worst case supervisor stack usage by pNA+ is 900 bytes plus the worst case stack usage for network interface drivers. pNA+ does not use any user stack space.

If pROBE+ is using pNA+ for communication purposes, the pROBE+ stack size must be increased by 1.5 Kbytes.

TCB Extensions

With pNA+ in a system, pSOS+ allocates a pNA+ TCB extension for each task at task creation. Memory for a pNA+ TCB extension comes from Region 0, and its size is 28 bytes.

pNA+ uses STREAMS memory management internally for data transfer. Data is represented in the form of messages. Each message is a three-structure triplet: Message Block, Data Block, and Data Buffer.

Message Blocks

A packet in pNA+ consists of a linked list of *mblks* (message blocks). Each message block represents part of the packet. The message structure is defined as follows:

```
struct msgb {
        struct msgb *b_next;      /* Next message on the queue */
        struct msgb *b_prev;      /* Previous message on the queue */
        struct msgb *b_cont;      /* Next message block */
        unsigned char *b_rptr;    /* First unread byte in buffer */
        unsigned char *b_wptr;    /* First unwritten byte in buffer */
        struct datab *b_datap;    /* Pointer to data block */
        };
typedef struct msgb mblk_t;
```

where

b_next Contains a pointer to the next message in the queue.

b_prev Contains a pointer to the previous message in the queue.

b_cont Contains a pointer to the next message block of the message (packet).

b_rptr Pointer to the first unread byte in the data buffer referred by the message block.

b_wptr Pointer to the first unwritten byte in the data buffer referred by the message block.

b_datap Pointer to the data block referred by the message block. The data block specifies the characteristics of the data buffer.

Data Blocks

A data block specifies the characteristics of the data buffer to which it refers. The structure is defined as follows:

```
struct datab {
   struct datab *db_freep;          /* Internal Use */
   unsigned char *db_base;          /* First byte of the buffer */
   unsigned char *db_lim;           /* Last byte+1 of buffer */
   unsigned char db_ref;            /* Number of refs to data buffer */
   unsigned char db_type;           /* Message type */
   unsigned char db_class;          /* Used internally */
   unsigned char db_debug;          /* Used internally */
   unsigned char db_frtn;           /* Free function and argument */
   };
typedef struct datab dblk_t;
```

where

db_freep Used internally by pNA+.

db_base Points to the first byte in the data buffer.

db_lim Points to the last byte + 1 of the data buffer.

db_ref Number of references to the data buffer.

db_type Type of data buffer.

db_class Used internally by pNA+.

db_debug Used internally by pNA+.

db_frtn Free function and argument.

Data Buffers

A data buffer is a contiguous block of memory used for storing packets/messages.

NAME

pRPC+ -- RAM requirements

DESCRIPTION

pRPC+ requires:

- Two Kbytes for a data area to store static variables.
- Two Kbytes of supervisor stack (pRPC+ executes in supervisor mode only, uses the calling task's supervisor stack, and requires no user stack space).
- Thirty six bytes for pRPC+ extensions to each task control block.

NAME

intro -- Introduction to Appendix A: Assembly Language Information

DESCRIPTION

This section gives helpful information to system programmers who understand 68K assembly language.

The 68K Assembly Language Interface is described in relation to the following:

- pSOS+ Real-Time Kernel (page A-3)
- I/O (page A-11)
- pHILE+ File System Manager (page A-15)

NAME

pSOS+ -- 68K Assembly Language Interface for pSOS+

DESCRIPTION

This section describes the assembly language interface for pSOS+. Although it is possible to write almost an entire pSOS+ application in C or a C-compatible language, various elements of a pSOS+ system can optionally be coded in assembly language. The following important assembly language programming topics are discussed:

- Calling Convention
- Table of System Calls
- Shutdown Register Usage

Calling Convention

All pSOS+ system calls except **i_return()** are invoked identically, as follows:

1. Input parameters are loaded into the processor's registers.
2. Register D0.L is loaded with a system call function number, which is unique for each call.
3. The processor executes a Trap #11 instruction, which puts it in the supervisor state. Through the previously installed *Trap #11 vector* in the vector page, program execution is passed to pSOS+.

Each system call has a unique set of input and output parameters, and all parameters occupy 32 bits.

Input parameters must be formatted and loaded into the specified registers before the system call is made. All address parameters should be on word boundaries, and pSOS+ checks for address and other parametric errors.

Output parameters are returned in registers and, in some cases, overwrite input parameters that use the same registers. System calls use register D0.L to return an error code. In general, this code indicates error conditions relevant and unique to the system service.

Most registers, including the processor status register and stack pointers, are preserved by pSOS+ and restored to their original content upon return to the caller--except the following:

- Registers used to return output parameters.
- Register D0.L, which returns an error code.
- Where specially noted.

Table of System Calls

The following table shows the calling conventions for each pSOS+ service call. The value shown in the Function Number column contains the system call function number (in decimal) that is loaded into D0.L. The Parameters column shows the service calls' other input and output parameters.

System Call	Function Number	I/O List & Parameters	
as_catch	47	Input:	A0: start_addr; D2: mode
		Output:	D0: 0 or error
as_return	49	Input:	None
		Output:	If no error, doesn't return. Otherwise, D0: error
as_send	48	Input:	D1: tid; D2: signals
		Output:	D0: 0 or error
errno_addr	12	Input:	None
		Output:	D0: address of task's errno
ev_asend	74	Input:	D1: tid; D4: events
		Output:	D0: 0 or error
ev_receive	45	Input:	D4: events; D5: flags; D6: timeout
		Output:	D0: 0 or error; D4: events_r
ev_send	44	Input:	D1: tid; D4: events
		Output:	D0: 0 or error
k_fatal	67	Input:	D1: err_code; D5: flags
		Output:	doesn't return
k_terminate	68	Input:	D1: err_code; D2: node; D5: flags
		Output:	D0: 0 or error
m_ext2int	69	Input:	A1: ext_addr
		Output:	D0: 0; A1: int_addr
m_int2ext	70	Input:	A1: int_addr
		Output:	D0: 0; A1: ext_addr
pt_create	20	Input:	D1: name; D2: length; D3: bsize; D5: flags; A0: laddr; A1: paddr
		Output:	D0: 0 or error; D1: ptid; D4: nbuf

System Call	Function Number	I/O List & Parameters	
pt_delete	22	Input:	D1: ptid
		Output:	D0: 0 or error
pt_getbuf	23	Input:	D1: ptid
		Output:	D0: 0 or error; A0: buf_addr
pt_ident	21	Input:	D1: name; D2: node
		Output:	D0: 0 or error; D1: ptid
pt_retbuf	24	Input:	D1: ptid; A0: buf_addr
		Output:	D0: 0 or error;
pt_sgetbuf	25	Input:	D1: ptid;
		Output:	D0: 0 or error; A0: laddr; A1: paddr
q_asend	75	Input:	D1: qid; D2: msg_buf[0]; D3: msg_buf[1]; D4: msg_buf[2]; D5:msg_buf[3]
		Output:	D0: 0 or error
q_aurgent	76	Input:	D1: qid; D2: msg_buf[0]; D3: msg_buf[1]; D4: msg_buf[2]; D5: msg_buf[3]
		Output:	D0: 0 or error
q_avsend	84	Input:	D1: qid; D2: msglen; A0: msg_buf
		Output:	D0: 0 or error
q_avurgent	85	Input:	D1: qid; D2: msglen; A0: msg_buf
		Output:	D0: 0 or error
q_broadcast	41	Input:	D1: qid; D2: msg_buf[0]; D3: msg_buf[1]; D4: msg_buf[2]; D5: msg_buf[3]
		Output:	D0: 0 or error; D2: count
q_create	36	Input:	D1: name; D2: count; D5: flags
		Output:	D0: 0 or error; D1: qid
q_delete	38	Input:	D1: qid
		Output:	D0: 0 or error
q_ident	37	Input:	D1: name; D2: node
		Output:	D0: 0 or error; D1: qid

System Call	Function Number	I/O List & Parameters	
q_receive	42	Input:	D1: qid; D5: flags; D6: timeout
		Output:	D0: 0 or error; D2: msg_buf[0]; D3: msg_buf[1]; D4: msg_buf[2]; D5: msg_buf[3]
q_send	39	Input:	D1: qid; D2: msg_buf[0]; D3: msg_buf[1]; D4: msg_buf[2]; D5: msg_buf[3]
		Output:	D0: 0 or error
q_urgent	40	Input:	D1: qid; D2: msg_buf[0]; D3: msg_buf[1]; D4: msg_buf[2]; D5: msg_buf[3]
		Output:	D0: 0 or error
q_vbroadcast	82	Input:	D1: qid; D2: msg_len; A0: msg_buf
		Output:	D0: 0 or error; D2: count
q_vcreate	77	Input:	D1: name; D2: maxnum; D3: maxlen; D5: flags
		Output:	D0: 0 or error; D1: qid
q_vdelete	79	Input:	D1: qid
		Output:	D0: 0 or error
q_vident	78	Input:	D1: name; D2: node
		Output:	D0: 0 or error; D1: qid
q_vreceive	83	Input:	D1: qid; D2: buf_len; D5: flags; D6: timeout A0: msg_buf;
		Output:	D3: msg_len
q_vsend	80	Input:	D1: qid; D2: msg_len; A0: msg_buf
		Output:	D0: 0 or error
q_vurgent	81	Input:	D1: qid; D2: msg_len; A0: msg_buf
		Output:	D0: 0 or error
rn_create	14	Input:	D1: name; D2: length; D3: unit_size; D5: flags; A1: saddr
		Output:	D0: 0 or error; D1: rnid; D2: asize

System Call	Function Number	I/O List & Parameters	
rn_delete	16	Input:	D1: rnid
		Output:	D0: 0 or error;
rn_getseg	17	Input:	D1: rnid; D3: size; D5: flags; D6: timeout
		Output:	D0: 0 or error; A0: seg_addr
rn_ident	15	Input:	D1: name
		Output:	D0: 0 or error; D1: rnid
rn_retseg	18	Input:	D1: rnid; A0: seg_addr
		Output:	D0: 0 or error
sm_av	73	Input:	D1: smid
		Output:	D0: 0 or error
sm_create	51	Input:	D1: name; D2: count; D5: flag
		Output:	D0: 0 or error; D1: smid
sm_delete	53	Input:	D1: smid
		Output:	D0: 0 or error
sm_ident	52	Input:	D1: name; D2: node
		Output:	D0: 0 or error; D1: smid
sm_p	54	Input:	D1: smid; D5: flags; D6: timeout
		Output:	D0: 0 or error
sm_v	55	Input:	D1: smid
		Output:	D0: 0 or error
t_create	1	Input:	D1: name; D2: prio; D3: sstack; D4: ustack; D5: flags
		Output:	D0: 0 or error; D1: tid
t_delete	5	Input:	D1: tid
		Output:	D0: 0 or error
t_getreg	10	Input:	D1: tid; D2: regnum
		Output:	D0: 0 or error; D3: reg_value
t_ident	2	Input:	D0: name; D2: node
		Output:	D0: 0 or error; D1: tid

System Call	Function Number	I/O List & Parameters	
t_mode	9	Input:	D1: mask; D2: new_mode
		Output:	D0: 0 or error; D2: old_mode
t_restart	4	Input:	D1: tid; D3: targs[0]; D4: targs[1]; D5: targs[2]; D6: targs[3]
		Output:	D0: 0 or error
t_resume	7	Input:	D1: tid
		Output:	D0: 0 or error
t_setpri	8	Input:	D1: tid; D2: new_prio
		Output:	D0: 0 or error; D2: old_prio
t_setreg	11	Input:	D1: tid; D2: regnum; D3: reg_value
		Output:	D0: 0 or error
t_start	3	Input:	D1: tid; D2: mode; D3: targs[0]; D4: targs[1]; D5: targs[2]; D6: targs[3] A0: start_addr
		Output:	D0: 0 or error
t_suspend	6	Input:	D1: tid
		Output:	D0: 0 or error
tm_cancel	64	Input:	D1: tmid
		Output:	D0: 0 or error
tm_evafter	62	Input:	D4: events; D6: ticks
		Output:	D0: 0 or error; EBX: tmid
tm_evevery	65	Input:	ESI: events; EDX: ticks
		Output:	EAX: 0 or error; D1: tmid
tm_evwhen	63	Input:	D1: date; D2: time; D3: ticks; D4: events
		Output:	D0: 0 or error; D1: tmid
tm_get	59	Input:	N/A
		Output:	D1: date; D2: time; D3: ticks
tm_set	58	Input:	D1: date; D2: time; D3: ticks
		Output:	D0: 0 or error
tm_tick	57	Input:	N/A
		Output:	0

System Call	Function Number	I/O List & Parameters	
tm_wkafter	60	Input:	D6: ticks
		Output:	D0: 0 or error
tm_wkwhen	61	Input:	D1: date; D2: time; D3: ticks
		Output:	D0: 0 or error

Shutdown Register Usage

When the fatal error handler performs node shutdown, pSOS+ either passes control to the user provided fatal error handler, passes control to pROBE+ (if present), or forces a divide by zero exception. In all cases, pSOS+ makes information describing the cause of the shutdown available to the processing entity.

On entry to the processing entity. the stack will contain a short exception frame (refer to the processor manual for the format of the short exception frame). If an application invokes **k_fatal()**, then the program counter (PC) in the exception frame will be the address of the TRAP instruction in the user code or interface library which invoked **k_fatal()**. If pSOSystem internally generated the fatal error or the shutdown occurred as a result of a shutdown packet from the master node, then the PC in the exception frame will not be meaningful.

In addition, pSOS+ loads the CPU registers as follows:

D1.L = The failure code.

D2.L = In multiprocessor systems, if the shutdown occurred due to a **k_terminate()** or a GLOBAL **k_fatal()** call, then D2.L will contain the node where the call was made. If the shut down occurred due to a LOCAL **k_fatal()** call or internally detected error condition, then D2.L will contain the node's own node number. In single processor systems D2.L is undefined.

NAME

I/O -- 68K Assembly Language I/O Interface

DESCRIPTION

This section describes the assembly language interface for I/O. The following topics are discussed:

- Calling Convention
- Backward Compatibility for Older Drivers
- Driver Register Usage

Calling Convention

I/O services are called in much the same way as other pSOS+ system calls. The conventions are as follows:

- Upon entry, register D0.L must contain a function number. The function number specifies the I/O operation, as follows:

  1 = **de_init()**

  2 = **de_open()**

  3 = **de_close()**

  4 = **de_read()**

  5 = **de_write()**

  6 = **de_cntrl()**

  D1.L must contain the device number (both major and minor), and A0.L must contain the address of an I/O Parameter Block (a user-defined structure).

- Entry into pSOS+ must happen through a Trap #12 instruction. The Trap #12 vector must have been loaded with the address of the pSOS+ I/O service entry.

- Upon return to the caller:

  D0.L contains an error code (0 = success)

  D1.L contains a quick-reference return value

When the I/O Supervisor is called from assembly language, pSOS+ returns the following additional error code if an invalid function code is loaded into D0.L:

HEX	MNEMONIC	DESCRIPTION
0103	ERR_IOOP	Illegal I/O function number.

Backward Compatibility for Older Drivers

pSOS+ actually supports two interfaces between pSOS+ and a device driver. These are known as the *old* and *new* interfaces. The preceding section (Calling I/O Services) describes the new interface. This section describes how to write a driver by using the old interface. Early versions of pSOS+ contain only the old interface. These versions pass parameters to and from a device driver through processor registers and therefore require that at least part of a device driver be in assembly language.

Newer versions of pSOS+ support both the old and the new interface. With the new interface, parameters pass to and from the device driver through the **ioparms** structure and therefore allow a device driver to be written entirely in C or a C-compatible language.

A driver written in assembly language can use either the new or old interface. (Specifically, the driver uses the values in **ioparms** in the same way a C language program would and returns to pSOS+ by using an RTS instruction.) Although the old interface is obsolete, it is supported by pSOS+ and documented here because many older pSOS+ drivers still use it. The remainder of this section describes the old interface.

When pSOS+ calls a device driver, in addition to putting a pointer to **ioparms** on the stack, it loads the CPU registers as follows:

- D0.L contains the caller's task id.
- D1.L contains **dev** as provided by the calling task.
- D2.L contains the calling task's processor status word in the lower 16 bits.
- A0.L contains **iopb** (a pointer to the I/O parameter block), which is provided by the calling task.

On exit from the driver, pSOS+ can use the following registers to pass parameters back to the calling task:

- D0.L is returned to the application as the I/O function call return value.
- D1.L is copied to the variable pointed to by the service call input parameter **retval**.

Device drivers return two scalar values to pSOS+. These are a quick-reference return value and an error code. When the C interface is used, they are returned to pSOS+ through the **err** and **out_retval** parameters in the **ioparms** structure, and when the old interface is used, they are returned through registers D0.L and D1.L. To determine which interface is being used and, hence, where to find the return values, pSOS+ examines a parameter in **ioparms** called **used**. If **used** is 0 (its value upon entry to the driver), pSOS+ assumes the old interface is being used. Otherwise, pSOS+ assumes the new interface is being used.

A driver must restore all register contents (except those used for output), restore any execution mode that it changed, and use a subroutine exit (RTS) to return control to pSOS+.

Driver Register Usage

This section provides additional information on how register values are managed while control passes from the application code to a device driver and back again.

When control passes from a calling task to a device driver, most register values pass through unchanged. The exceptions are D2.L, which will contain the calling task's ID, and A1.L, which pSOS+ uses internally. In addition, the C language bindings (**de_init()**, **de_open()**, and so on) alter D0.L, D1.L, and A0.L before pSOS+ gains control.

When control passes from the device driver back to the calling task, all register values are passed through unchanged. Thus, if the driver has altered a register value, the altered value remains when control returns to the task.

Finally, earlier pSOS+ manuals document a convention by which the address of a driver's data area is passed between pSOS+ and the driver through register A1. Although this convention is obsolete, pSOS+ still supports it for backward compatibility. If an explanation of how to use A1 in this way is needed, Integrated Systems Technical Support will provide it.

NAME

pHILE+ -- 68K Assembly Language Interface for pHILE+

DESCRIPTION

This section describes the assembly language interface for pHILE+.

Although it is now possible to write almost an entire pSOSystem application in C or a C-compatible language, various elements may optionally be coded in assembly language. This section provides useful information for assembly language users of pHILE+.

Calling Convention

All pHILE+ system calls are invoked in the following manner:

1. All call parameters are pushed onto the stack in a right-to-left order. Each parameter occupies four bytes.
2. Register D0.L is loaded with a system call function number that is unique for each call.
3. A Trap #11 instruction is executed.

Execution of the Trap #11 puts the processor into the supervisor state and passes control to pSOS+ through the previously installed *Trap #11 vector*. By examining the function number in D0, pSOS+ determines the call is a pHILE+ call and passes control to pHILE+.

Control returns to the caller at the instruction following the Trap #11 instruction. At this time, register D0.L contains the error code. (A 0 indicates a successful call, and any other value indicates an error.) All other registers (including the stack pointers) are fully restored to their original content upon return to the caller.

The following example shows how a system call can be programmed using assembly language:

The C language call

```
lseek_f(fid, position, offset, &old_ptr)
```

could be written in assembly language as:

```
PEA.L       old_ptr           ;PUSH ADDRESS TO GET OLD L_PTR
MOVE.L      offset,-(SP)      ;PUSH THE OFFSET
MOVE.L      position,-(SP)    ;PUSH POSITIONING METHOD
MOVE.L      fid,-(SP)         ;PUSH THE FILE DESCRIPTOR
MOVE.L      #20EH,D0          ;LOAD FUNCTION CODE
TRAP        #11               ;AND CALL PHILE+
TST.L       D0                ;ANY ERRORS?
```

```
BNE          ERROR            ;YES, PROCESS IT
```

Table of System Calls

The following table shows the function codes for each pHILE+ service:

System Call	Function Number	System Call	Function Number
access_f	0x213	**nfsmount_vol**	0x224
annex_f	0x211	**open_dir**	0x216
change_dir	0x20C	**open_f**	0x20D
chmod_f	0x225	**open_fn**	0x21A
chown_f	0x227	**pcinit_vol**	0x21B
close_dir	0x218	**pcmount_vol**	0x21C
close_f	0x214	**read_dir**	0x217
create_f	0x208	**read_f**	0x20F
fchmod_f	0x226	**read_link**	0x22B
fstat_f	0x222	**read_vol**	0x205
fstat_vfs	0x21D	**remove_f**	0x20A
ftruncate_f	0x220	**stat_f**	0x221
get_fn	0x219	**stat_vfs**	0x215
init_vol	0x201	**symlink_f**	0x22A
link_f	0x229	**sync_vol**	0x203
lock_f	0x212	**truncate_f**	0x21F
lseek_f	0x20E	**unmount_vol**	0x204
lstat_f	0x223	**utime_f**	0x207
make_dir	0x209	**verify_vol**	0x21E
mount_vol	0x202	**write_f**	0x210
move_f	0x20B	**write_vol**	0x206

Index

A

Assembly Language, 68K A-1
- for pHILE+ A-15
- for pSOS+ A-3
- I/O interface A-11

B

bootpd 1-3, 1-5
- configuration database 1-7
- configuration table 1-7
- daemon task 1-5
- generic tag 1-8
- parent IP address 1-8
- resource requirements 1-5
- server options 1-7
- starting Routing Daemons 1-5, 1-6
- system requirements 1-5
- tag symbol 1-8

BTPD 1-5

C

Configuration Tables 3-1
- multiprocessor 3-5
- node 3-3
- pHILE+ 3-25
- pNA+ 3-31
- pREPC+ 3-29
- pROBE+ 3-15
- pRPC+ 3-39
- pSOS+ 3-9

D

Device Driver
- guidelines for writing 1-37

DISI 2-31
- callback functions 2-33
- data structures 2-51
- error codes 2-54
- features 2-34
- function calls 2-32
- multiplex driver mapping 2-55
- SerialClose function 2-44
- SerialInit function 2-35
- SerialIoctl commands 2-41
- SerialIoctl function 2-41
- SerialOpen function 2-35
- SerialSend function 2-38
- user callback functions 2-45

DISIplus 2-57
- callback functions 2-59
- data structures 2-82
- error codes 2-89
- function calls 2-58
- multiplex driver mapping 2-89

F

FTP Client 1-11
- commands 1-12
- configuration 1-11
- help 1-13
- startup 1-11

FTP Client Bugs 1-18

FTP Command
- file naming conventions 1-17
- file transfer parameters 1-17

FTP Server 1-19
- configuration 1-19
- configuration table 1-19
- startup 1-21

G

gateway structure 1-76
- parameter 1-76

K

Kernel Interface 2-21
- conventions 2-26
- error conditions 2-25
- packet buffer sizes 2-22
- packets
  - packet buffers 2-21
- services 2-27
- transmission requirements 2-24

L

Loader 1-23
- concepts and operations 1-25
- configuration 1-24
- functions 1-23
- load function 1-26, 1-30
- load function errors 1-31
- loader API 1-28
- object files 1-23
- procedure for compiling
  - running application on 1-36
- release function 1-26, 1-33
- release function errors 1-33
- unload function 1-32
- unload function errors 1-33
- user configurable modules 1-24

M

Memory Usage 4-1
- pHILE+ 4-7
- pNA+ 4-11
- pREPC+ 4-9
- pRPC+ 4-15
- pSOS+ 4-3

mmulib 1-39
- concepts and operations 1-40
- functions 1-42
- map template 1-41
- page attributes 1-41

N

Network Interface 2-3
- calling conventions 2-10
- MIB-II related operations 2-15
- packets
  - packet buffers 2-3
- pNA+-dependent interface 2-5
- pNA+-independent interface 2-4
- pROBE+ debug support 2-9

services 2-3, 2-10
NFS Server 1-47
configuration 1-47

P

pSH 1-51
adding applications to 1-56
adding commands to 1-55
built-in commands 1-56
command descriptions 1-58
configuration 1-51
shell 1-51
subroutines 1-56

R

RARP 1-73
routed 1-75
configuration table 1-75
starting routing daemons 1-77
system/resource requirements 1-75

S

SCSI
pSOS-to-Driver interface 2-97
upper to lower driver interface 2-98
user interface 2-93
System Calls A-4
for pHILE+ A-16
for pSOS+ A-4

T

Telnet Client 1-79
commands 1-80
configuration 1-79
Telnet Server 1-85
configuration 1-85
TFTP Server 1-87
configuration 1-87

Document Title: **pSOSystem Programmer's Reference**
Part Number: **000-5078-001**
Revision Date: **March 1996**