pSOSystem Product Family

pSOSystem Programmer’s Reference

PowerPC Processors

integrated systems

000-5437-001

Copyright © 1999 Integrated Systems, Inc. All rights reserved. Printed in U.S.A.
Document Title: **pSOSystem Programmer's Reference, PowerPC Processors**
Part Number: **000-5437-001**
Revision Date: **January 1999**

Integrated Systems, Inc. • 201 Moffett Park Drive • Sunnyvale, CA 94089-1322

	Corporate	pSOS or pRISM+ Support	MATRIX$_X$ Support
Phone	408-542-1500	1-800-458-7767, 408-542-1925	1-800-958-8885, 408-542-1930
Fax	408-542-1950	408-542-1966	408-542-1951
E-mail	ideas@isi.com	psos_support@isi.com	mx_support@isi.com
Home Page	http://www.isi.com		

Contents

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 includes *pSOSystem System Concepts*, *pSOSystem Advanced Topics*, *pSOSystem Application Examples*, *pSOSystem System Calls*, *pROBE+ User's Guide* as well as installation guides and the Getting Started manuals.

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:

- Chapter 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.

- Chapter 2, *Interfaces*, describes the pSOSystem Network Interface and Kernel Interface.

- Chapter 3, *Standard pSOSystem Block I/O Interface*, describes block I/O interface concepts, pHILE+ and the SCSI driver.
- Chapter 4, *Standard pSOSystem Character I/O Interface*, discusses the character I/O interface concepts and document the character I/O driver components of pSOS.
- Chapter 5, *pSOSystem Configuration File*, gives an overview of the configuration table associated with the corresponding pSOSystem component, tells how to start up each component, contains formulas for changing the starting address of software components, and defines the trap vectors.
- Chapter 6, *Configuration Tables*, describes each software component's configuration table, which contains parameters that characterize the hardware and application environment.
- Chapter 7, *Memory Usage*, describes formulas used to calculate the amount of RAM required by each pSOSystem software component.
- Chapter 8, *pNET: Ethernet Debugging Without Using pNA*, provides the target debug solution based on the UDP/IP protocol.

Related Documentation

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

- *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 Guide:* describes how to use the pROBE+ System Debugger/Analyzer.
- User's Guide*:* contains an introduction to the pSOSystem within the development environment, tutorials, and information on files and directories.
- *pSOSystem Advanced Topics:* contains information on how to customize your usage of your pSOSystem. It contains sections on using and creating BSPs and Assembly Language information.
- *pSOSystem Application Examples:* describes the application examples that are provided for you and tutorials on how to use these examples.

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.
- *SNMP User's Guide*: 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.
- *LAP Driver User's Guide* describes the interfaces provided by the LAP (Link Access Protocol) drivers for OpEN product, including the LAPB and LABD frame-level products.
- OpEN *User's Guide:* describes how to install and use the pSOSystem OpEN (OpEN Protocol Embedded Networking) product.
- *TCP/IP for OpEN User's Guide:* describes how to use the pSOSystem Streams based TCP/IP for OpEN (OpEN Protocol Embedded Networking) product.

Notation Conventions

This section describes the conventions used in this document.

Font Conventions

This sentence is set in the default text font, Bookman Light. Bookman Light is used for general text, menu selections, window names, and program names. Fonts other than the standard text default have the following significance:

`Courier:`	`Courier` is used for command and function names, file names, directory paths, environment variables, messages and other system output, code and program examples, system calls, prompt responses, and syntax examples.
`bold Courier:`	**`bold Courier`** is used for user input (anything you are expected to type in).

italic:	*Italics* are used in conjunction with the default font for emphasis, first instances of terms defined in the glossary, and publication titles. Italics are also used in conjunction with *`Courier`* or **`bold Courier`** to denote placeholders in syntax examples or generic examples.
Bold Helvetica narrow:	**Bold Helvetica narrow** font is used for buttons, fields, and icons in a graphical user interface. Keyboard keys are also set in this font.

Sample Input/Output

In the following example, user input is shown in **`bold Courier`**, and system response is shown in `Courier`.

```
commstats

Number of total packets sent                                160
Number of acknowledgment timeouts                             0
Number of response timeouts                                   0
Number of retries                                             0
Number of corrupted packets received                          0
Number of duplicate packets received                          0
Number of communication breaks with target                    0
```

Symbol Conventions

This section describes symbol conventions used in this document.

[]	Brackets indicate that the enclosed information is optional. The brackets are generally not typed when the information is entered.
|	A vertical bar separating two text items indicates that either item can be entered as a value.
˘	The breve symbol indicates a required space (for example, in user input).
%	The percent sign indicates the UNIX operating system prompt for C shell.
$	The dollar sign indicates the UNIX operating system prompt for Bourne and Korn shells.

68K The symbol of a processor located to the left of text identifies processor-specific information (the example identifies 68K-specific information).

XXXXX Host tool-specific information is identified by a host tools icon (in this example, the text would be specific to the XXXXX host tools chain).

Support

Customers in the United States can contact Integrated Systems Technical Support as described below.

International customers can contact:

- The local Integrated Systems branch office.
- The local pSOSystem distributor.
- Integrated Systems Technical Support as described below.

Before contacting Integrated Systems Technical Support, please gather the following information available:

- Your customer ID and complete company address.
- Your phone and fax numbers and e-mail address.
- Your product name, including components, and the following information:
  - The version number of the product.
  - The host and target systems.
  - The type of communication used (Ethernet, serial).
- Your problem (a brief description) and the impact to you.

In addition, please gather the following information:

- The procedure you followed to build the code. Include components used by the application.
- A complete record of any error messages as seen on the screen (useful for tracking problems by error code).
- A complete test case, if applicable. Attach all include or startup files, as well as a sequence of commands that will reproduce the problem.

Contacting Integrated Systems Support

To contact Integrated Systems Technical Support, use one of the following methods:

- Call 408-542-1925 (U.S. and international countries).
- Call 1-800-458-7767 (1-800-458-pSOS) (U.S. and international countries with 1-800 support).
- Send a FAX to 408-542-1966.
- Send e-mail to `psos_support@isi.com`.
- Access our web site: `http://customer.isi.com`.

Integrated Systems actively seeks suggestions and comments about our software, documentation, customer support, and training. Please send your comments by e-mail to `ideas@isi.com` or submit a Problem Report form via the internet (http://customer.isi.com/report.shtml).

1 System Services

This section describes the pSOSystem system services.

NOTE: The Network Utilities product is referred to as Internet Applications.

Function	Description
`Bootp`	Pseudo driver that requests BOOTSTRAP protocol information. (See page 1-3.)
`Bootpd`†	Implementation of the BOOTSTRAP protocol server. (See page 1-6.)
`Client API Support`†	Provides API support for accessing the client protocols, which are Telnet, FTP, and TFTP. Also, it provides interfaces common to all the Internet Applications modules. (See page 1-13.)
`DHCP Client`†	Provides framework for passing configuration information to TCP/IP hosts on the network. (See page 1-33.)
`DNS and Static Name Resolver`†	Allows pSOSystem target to resolve host names to IP addresses. (See page 1-42.)
`FTP Client`†	Transfers files to and from a remote system. (See page 1-50.)
`FTP Server`†	Allows remote systems running FTP to transfer files to and from a pHILE+ device. (See page 1-59.)
`Loader`	Allows run-time target loading and unloading of application programs. (See page 1-62.)
`pLM+`	Provides information on the Shared Library Manager. (See page 1-80.)

Function	Description
`MMU Library`	Provides mapping tables for the Memory Management Unit. (See page 1-101.)
`NFS Server`†	Allows systems to share files in a network environment. (See page 1-111.)
`pSH+`†	Interactive command line shell. (See page 1-115.)
`pSH Loader`†	Provides a simple interface to the loader library in pSOSystem. (See page 1-142.)
`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). (See page 1-144.)
`Routed`†	Implementation of the Routing Information Protocol, or RIP. (See page 1-145.)
`Telnet Client`†	Supports communication with a remote system running a Telnet Server. (See page 1-148.)
`Telnet Server`†	Allows remote systems running the Telnet protocol to log into pSH+. (See page 1-156.)
`TFTP Client`†	Provides a simple command-line user interface for using the TFTP Client protocol. (See page 1-159.)
`TFTP Server`†	Allows TFTP clients to read and write files interactively on pHILE+ managed disks. (See page 1-162.)

†. This feature is part of the Internet Applications product.

bootp

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 bootp client is provided as source in the pSOSystem PSS_ROOT/drivers directory.

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) defined 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 one second, the second retry interval of two seconds, and so on. If this parameter is set to zero, 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. If the COOK flag is set, all the parameters related to an IP address (gateway IP address, bootp server address, netname, etc.) are returned in networking byte order. You are required to use ntohl() to convert them to host byte order. If RAW flag is set, the whole bootp packet is returned so all fields will be in network byte order.

BOOTP Client Code Example

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);

...
```

Bootp client is released in source form, in the drivers/bclient.c file. To include this in a pSOSystem application, add the following lines in the makefile of the application. Add bclient.o in the list of PSS_DRVOBJS and the following lines in the appropriate section of the makefile:

```
bclient.o : $(PSS_ROOT)/drivers/bclient.c\
            $(PSS_ROOT)/bsp.h\
makefile
$(CC) $(COPTS) -o bclient.o - c $(PSS_ROOT)/drivers/bclient.c
```

bootpd

Description

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

NOTE: The bootpd is provided in the Internet Applications 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:

- Four Kbytes of task 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 BOOTP Daemon

To use bootpd in an application, you need to link the pSOSystem Internet Applications library.

bootpd is started with bootpd_start(bootpdcfg_t*). If bootpd is started successfully, bootpd_start() returns zero; otherwise, it returns a non-zero value on failure. The error value can be any pSOS+ error.

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 1-9.

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

Client API Support

Description

Client Application Programming Interface (API) Support is used to provide a programmatic interface to the client protocols, which are FTP, Telnet, and TFTP. Client API Support is contained in pSOSystem's Internet Applications product. This API support provides the following advantages:

- Isolate client functions from pSH+. You are able to run the client functions from your own custom-made shells.
- Use client functions from your own application programs (for example, HTTP servers). You do not need to run the client functions from an interactive shell.

Client API support is designed to be similar for all the client applications; however, details differ depending on the application.

The Client API Support requires the same resources as required by the corresponding FTP, TFTP, or Telnet clients for each of their sessions. See *FTP Client*, *TFTP Client*, or *Telnet Client* sections for additional details.

Generic Client API Functions

Before you use a client application, a task will need to create a handle for it. This handle is only used from the task that created it and can not be shared across multiple tasks. The following is an example of how to create a handle:

```
xxx_handle_t
xxx_create (int indev, int outdev,.... )
```

where `xxx` can be one of the following `tftp`, `ftp`, or `telnet`.

This API call is passed an input and output device that will be used as `stdin` and `stdout` for this client session. The call will remap the task's `stdio`, which is mapped to the console port by default to the device that you passed. Subsequently, all the `stdio` operations executed by the client program (for example, `gets()` for input and `printf()` for output) are redirected to your supplied device. In addition, this call allocates memory for a session structure from Region 0, which is used to store the parameters related to this session. When the task needs to end the session, the following API call is used to destroy the session structure and free the associated memory:

```
xxx_destroy(xxx_handle_t xxx)
```

After the handle is created, the task can use other client API calls to perform various operations. All of the API calls require you to pass a handle as one of the parameters. The specific API calls are dependent on the particular client applications. See *TFTP API Functions*, *FTP API Functions*, and *Telnet API Functions* sections for a complete list of all the API calls.

The DEV_NULL sample driver is provided with the Internet Applications library. While creating an application handle xxx_create() call, the device number of this device can be used as indev and outdev parameters. This provides the functions similar to /dev/null on UNIX systems — all the output written to this will be lost and input from it will always be end-of-file (EOF). It is used as default stdio device by the application that needs to run the client API calls in non-interactive mode without the output displayed anywhere and no input needed. For example, while creating a FTP client API handle, the following call is used:

```
ftp_handle_t hand= ftp_create(DEV_NULL, DEV_NULL, envp);
```

For all of the subsequent API calls using this handle, the output is not displayed anywhere and if any user input is needed by any of the calls, it will always be returned EOF.

Several API calls require the output not to be displayed anywhere, but they may need to parse the data. For example, if an application task creates a FTP client API handle and calls ftp_dir() and lists the files in a remote directory. The application will need to examine the list of output files returned by the API call to select a file to transfer. This is achieved by using another generic API call, which is used for all of the client APIs. The following is how it can be accomplished:

- The application creates a handle using DEV_NULL as indev and outdev parameters. The following API call is called to install your own input and output routines for the DEV_NULL driver:

  ```
  nuapi_installio(DEV_NULL, read_fn, write_fn);
  ```

  If any of the client API calls prints any output on its stdout, the data gets directed to DEV_NULL device that invokes write_fn(), which you provide and it passes the buffer and buffer length. By using this, you can capture the output of the client application APIs. nuapi_installio() is used as a generic API function by all the client APIs.

The following is the prototype definition for this function:

```
int nuapi_installio(int device, int(*write_fn)(char *buf, int
len), int(*read_fn)(char *buf, int len))
```

NOTE: The device driver DEV_NULL is a regular pSOS+ driver and each read/write function to be installed using nuapi_installio() should be done on a separate minor device (for example, DEV_NULL+1, DEV_NULL+2, and so on).

Note that all API calls except create and destroy return CLIENT_SUCCESS on success or an error code on failure. See *Error Codes for Client APIs* section for a list of all the error codes.

NOTE: All of the IP addresses and port number parameters in this section should be in network byte order.

TFTP API Functions

The prototypes for the API functions are included in the header file nu_api.h.

The following are descriptions of the TFTP API functions:

```
#include <nu_api.h>
tftp_handle_t tftp_create(int indev, int outdev, char *cvol, char
                          *cdir)
```

Create a TFTP handle with the given input/output devices.

indev	Specifies the device used as standard input (stdin).
outdev	Specifies the device used as standard output (stdout).
cvol	Specifies the volume for the local volume as a NULL terminated string up to 32 characters (for example, "3.0").
cdir	Specifies the pathname to the local directory as a NULL terminated string up to 32 characters (for example, "/tftpboot").

This function returns a TFTP client API handle if successful; otherwise, NULL if fails to create. All subsequent TFTP operations on file will be done at the volume and directory specified as arguments.

`int tftp_destroy(tftp_handle_t hand)`

Destroy the client handler and frees the memory resources associated with this session.

`hand`	Specifies the client API handle.

```
int tftp_connect(tftp_handle_t hand, struct in_addr *ipaddr, int
                port)
```

Connect to a TFTP server at the `ipaddr` address and port `port`. If `port` is zero, use the default TFTP Server port. If this call is used. the calls `tftp_fget()` and `tftp_fput()` need not specify the remote hostname. Otherwise, they need to specify the filename in **`hostaddress:`*`filename`*** format.

`hand`	Specifies the client API handle.
`ipaddr`	Specifies the IP address of the host to connect to. This is specified in host byte order.
`port`	Specifies the port number of the TFTP server. Use the default port if this is zero.

This function returns zero on success or an error code on failure.

`int tftp_mode(tftp_handle_t hand, char *mode)`

Set the mode of the TFTP file transfer, which can be ascii or binary mode.

`hand`	Specifies the client API handle.
`mode`	Specifies "`ascii`" or "`binary`" mode for the TFTP file transfer.

This function returns zero on success or an error code on failure.

`int tftp_fget(tftp_handle_t hand, char *rem_file, char *loc_file)`

Get a remote file `rem_file` and copy it as `loc_file`. If `loc_file` is NULL, it creates a local file with the same name as the `rem_file`.

`hand`	Specifies the client API handle.

`rem_file`	Specifies the remote file to be transferred as a NULL terminated string up to 32 characters.
`loc_file`	Specifies the local file to be copied as a NULL terminated string up to 32 characters.

This function returns zero on success or an error code on failure.

```
int tftp_fput(tftp_handle_t hand, char *locfile, char *remfile)
```

Put a file `locfile` at a remote site as the `remfile`. If `remfile` is NULL, it creates a remote file with the same name as `locfile`.

`hand`	Specifies the client API handle.
`locfile`	Specifies the local file to be transferred as a NULL terminated string up to 32 characters. This can be a regular file or a pSOS+ device (for example, "13.0").
`remfile`	Specifies the remote file to be copied as a NULL terminated string up to 32 characters.

This function returns zero on success or an error code on failure.

```
int tftp_retxmits(tftp_handle_t hand, int no_retx)
```

Set the number of retransmits before timing out a transaction for this session.

`hand`	Specifies the client API handle.
`no_retx`	Specifies the number of retransmits before timeout.

This function returns zero on success or an error code on failure.

```
int tftp_timeout(tftp_handle_t, int timeout)
```

Set the timeout value for retransmission for this session.

`hand`	Specifies the client API handle.

`timeout`	Specifies timeout (in seconds) before retransmitting a packet.

This function returns zero on success or an error code on failure.

`int tftp_blksize(tftp_handle_t, int blksize)`

Set the block size for TFTP file transfers for this session. The default size is 512 bytes.

`hand`	Specifies the client API handle.
`blksize`	Specifies the blocksize for transfer.

This function returns zero on success or an error code on failure.

`int tftp_option(tftp_handle_t, char*)`

Toggle the option specified in `option` (enable or disable) for this session.

`hand`	Specifies the client API handle.
`option`	Specifies the name of the option to be toggled. The option can be either "`blksize`," "`timeout`," or "`tsize`."

This function returns zero on success or an error code on failure.

`int tftp_filesize(tftp_handle_t hand, int filesz)`

Set the maximum file size supported for TFTP transfers on this session.

`hand`	Specifies the client API handle.
`filez`	Specifies the maximum file size supported for transfer.

This function returns zero on success or an error code on failure.

```
int tftp_verbose(tftp_handle_t hand)
```

Toggle the verbose mode for the TFTP session.

`hand`	Specifies the client API handle.

This function returns zero on success or an error code on failure.

```
int tftp_status(tftp_handle_t hand)
```

Display the status information for this session.

`hand`	Specifies the client API handle.

This function returns zero on success or an error code on failure.

```
int tftp_trace(tftp_handle_t hand)
```

Toggle the packet tracing for this TFTP session.

`hand`	Specifies the API handle.

This function returns zero on success or an error code on failure.

```
int tftp_help(tftp_handle_t hand, char *cmd)
```

Display help information about the command `cmd`. If `cmd` is NULL, it displays all of the commands.

`hand`	Specifies the client API handle.
`cmd`	Specifies the name of the command

This function returns zero on success or an error code on failure.

FTP API Functions

The prototypes for the API functions are included in the header file `nu_api.h`.

The following are the descriptions for the FTP API functions:

```
#include <nu_api.h>
ftp_handle_t ftp_create(int indev, int outdev, char *cvol,
                        char *cdir)
```

Create a FTP handle with the given input/output devices.

`indev`	Specifies the device to be used as standard input (`stdin`).
`outdev`	Specifies the device to be used as standard output (`stdout`).
`cvol`	Specifies the volume for the local directory as a NULL terminated string up to 32 characters (for example, "3.0").
`cdir`	Specifies the pathname to the local directory as a NULL terminated string up to 32 characters (for example, "`/ftpdir`").

This function returns a client API handle if successful; otherwise, NULL if it fails to create. All subsequent FTP operations will be performed at the volume and directory specified as arguments.

```
int ftp_destroy(ftp_handle_t hand)
```

Destroy the client handler and free the memory resources associated with this session.

`hand`	Specifies the client API handle.

```
int ftp_connect(ftp_handle_t hand, struct in_addr *ip_addr,
                int port)
```

Connect to the remote FTP server at address `ip_addr` and port `port`. If the `port` is zero, it connects to the default FTP server port.

`hand`	Specifies the client API handle.
`ip_addr`	Specifies the IP address of the remote host to connect. This is specified in host byte order.
`port`	Specifies the port number of the FTP server to connect. This is specified in host byte order.

```
int ftp_close(ftp_handle_t hand)
```

Close the current FTP session.

`hand`	Specifies the client API handle.

```
int ftp_login(ftp_handle_t hand, char *user, char *passwd)
```

Log in at the remote site as user `user` and with a password `passwd`.

`hand`	Specifies the client API handle.
`user`	Specifies the username as a NULL terminated string up to 32 characters.
`passwd`	Specifies the password for the user as a NULL terminated string up to 32 characters.

```
int ftp_fget(ftp_handle_t hand, char *rem_file, char *loc_file)
```

Copy the remote file `rem_file` to the local file named `loc_file`. If `loc_file` is NULL, it will create a local file with name `rem_file` and copy to this file.

`hand`	Specifies the client API handle.
`rem_file`	Specifies the name of the remote file as a NULL terminated string up to 32 characters.
`loc_file`	Specifies the name of the local file to copy to as a NULL terminated string up to 32 characters. This can be a regular file or pSOS+ device (for example, “13.0”).

```
int ftp_fput(ftp_handle_t, char *loc_file, char *rem_file)
```

Copy the local file `loc_file` to the remote file named `rem_file`. If `rem_file` is NULL, it will create a remote file named `loc_file` and copy to this file.

`hand`	Specifies the client API handle.
`loc_file`	Specifies the name of the remote file as a NULL terminated string up to 32 characters.
`rem_file`	Specifies the name of the remote file as a NULL terminated string up to 32 characters.

`int ftp_account(ftp_handle_t hand, char *account)`

Send an account command to the remote server.

`hand`	Specifies the client API handle.
`account`	Specifies the account name as a NULL terminated string up to 64 characters.

`int ftp_ttype(ftp_handle_t hand, char *type)`

Set the file transfer type to `type`.

`hand`	Specifies the client API handle.
`type`	Specifies the file transfer mode. It can be either "`ascii`" or "`binary`."

`int ftp_bell(ftp_handle_t hand)`

Enable beep when the command completes.

`hand`	Specifies the client API handle.

`int ftp_cddir(ftp_handle_t hand, char *dir)`

Change the directory at the remote site to `dir`.

`hand`	Specifies the client API handle.
`dir`	Specifies the directory name at the remote site as a NULL terminated string up to 64 characters.

`int ftp_cdup(ftp_handle_t hand)`

Change the directory at the remote site to one level up.

`hand`	Specifies the client API handle.

`int ftp_dir(ftp_handle_t hand, char *opts, char *remdir)`

List the remote directory `remdir` along with the options `opts`.

`hand`	Specifies the client API handle.
`opts`	Specifies the directory listing options as a NULL terminated string up to 32 characters.
`remdir`	Specifies the remote directory as a NULL terminated string up to 32 characters.

`int ftp_delete(ftp_handle_t hand, char *filename)`

Delete the remote file named `filename`.

`hand`	Specifies the client API handle.
`filename`	Specifies the name of the remote file as a NULL terminated string up to 32 characters.

`int ftp_hash(ftp_handle_t hand)`

Toggle the printing # symbol to indicate the progress of the file transfer.

`hand`	Specifies the client API handle.

`int ftp_help(ftp_handle_t hand)`

Display all of the supported commands.

`hand`	Specifies the client API handle.

`int ftp_lcd(ftp_handle_t hand, char *dir)`

Change to the directory named `dir` at the local site.

`hand`	Specifies the client API handle.
`dir`	Specifies the name of the local directory as a NULL terminated string up to 64 characters.

`int ftp_mdelete(ftp_handle_t hand, int number, char **list)`

Delete the multiple files that are specified in `list` at the remote site.

`hand`	Specifies the client API handle.
`number`	Specifies the number of names in the list.
`list`	Specifies the list of names to be deleted. These filenames are specified as a list of NULL terminated strings.

`int ftp_mdir(ftp_handle_t hand, int number, char **list)`

List the multiple directories that are specified in `list` at the remote site.

`hand`	Specifies the client API handle.
`number`	Specifies the number of directories in the list.
`list`	Specifies the list of directory names to be displayed. These filenames are specified as a list of NULL terminated strings.

`int ftp_mget(ftp_handle_t hand, int number, char **list)`

Get the multiple files that are specified in `list` from the remote site and copy them locally.

`hand`	Specifies the API handle.
`number`	Specifies the number of names in the list.
`list`	Specifies the list of filenames to be copied locally. These filenames are specified as a list of NULL terminated strings.

`int ftp_mput(ftp_handle_t hand, int number, char **list)`

Put the multiple files that are specified in `list` from the local site at the remote site.

`hand`	Specifies the client API handle.
`number`	Specifies the number of names in the list.
`list`	Specifies the list of filenames to be copied. These filenames are specified as a list of NULL terminated strings.

`int ftp_mkdir(ftp_handle_t hand, char *dir)`

Create a directory named `dir` at the remote site.

`hand`	Specifies the client API handle.
`dir`	Specifies the name of the directory to be created as a NULL terminated string up to 64 characters.

`int ftp_prompt(ftp_handle_t hand)`

Toggle forcing interactive prompting on multiple commands.

`hand`	Specifies the client API handle.

`int ftp_sendport(ftp_handle_t hand)`

Toggle the use of PORT command for each data connection.

`hand`	Specifies the client API handle.

`int ftp_pwd(ftp_handle_t hand)`

Display the working directory path at the remote site.

`hand`	Specifies the client API handle.

`int ftp_quote(ftp_handle_t hand, char *quote)`

Send a quote to the remote site.

`hand`	Specifies the client API handle.
`quote`	Specifies the quote to be sent to the remote site as a NULL terminated string up to 64 characters.

`int ftp_rmthelp(ftp_handle_t hand, char *remote)`

Get help from the remote server. If the remote argument is NULL, help on all commands are requested from the server.

`hand`	Specifies the client API handle.
`remote`	Specifies the command on which the remote help is needed.

`int ftp_rename(ftp_handle_t hand, char *fromfile, char *tofile)`

Rename the remote file `fromfile` to `tofile`.

`hand`	Specifies the client API handle.
`fromfile`	Specifies the file name to be renamed as a NULL terminated string up to 32 characters.
`tofile`	Specifies the file name to be named as a NULL terminated string up to 32 characters.

`int ftp_reset(ftp_handle_t hand)`

Clear the queued command replies.

`hand`	Specifies the client API handle.

`int ftp_rmdir(ftp_handle_t hand, char *remdir)`

Remove the directory named `remdir` at the remote site.

`hand`	Specifies the client API handle.
`remdir`	Specifies the name of the directory to be removed as a NULL terminated string up to 64 characters.

`int ftp_runique(ftp_handle_t hand)`

Toggle the storing unique names for local files.

hand	Specifies the client API handle.

`int ftp_sunique(ftp_handle_t hand)`

Toggle storing unique filenames on remote machines.

`hand`	Specifies the client API handle.

`int ftp_status(ftp_handle_t hand)`

Show the status of this FTP session.

`hand`	Specifies the client API handle.

```
int ftp_verbose(ftp_handle_t hand)
```

Toggle the verbose mode for this FTP session.

`hand`	Specifies the client API handle.

Telnet API Functions

The prototypes for the API functions are included in the header file `nu_api.h`.

The following are the descriptions for the Telnet API functions:

```
tnp_handle_t telnet_create(int indev, int outdev, char *term_type,
                           char *tn_prompt)
```

Create a Telnet handle with the given input and output devices.

`indev`	Specifies the device to be used as standard input (`stdin`).
`outdev`	Specifies the device to be used as standard output (`stdout`).
`term_type`	Specifies the terminal type on which this Telnet session is operating as a NULL terminated string up to 32 characters.
`tn_prompt`	Specifies the prompt string for this Telnet session as a NULL terminated string up to 32 characters.

This function returns a client API handle if successful; otherwise, NULL if fails to create. If `term_type` is specified as NULL, the default `ansi` terminal type is used.

```
int telnet_destroy(tnp_handle_t hand)
```

Destroy the client handler and free the memory resources associated with this session.

`hand`	Specifies the client API handle.

```
int telnet_connect(tnp_handle_t hand, struct in_addr *ip_addr, int
                   port)
```

Connect to the remote telnet server at the ipaddress `ip_addr` and port number `port`. If `port` is zero, connect to the default telnet server port.

`hand`	Specifies the client API handle.
`ip_addr`	Specifies the IP address of the remote host to connect. This is specified in host byte order.
`port`	Specifies the port number of the telnet server to connect. This is specified in host byte order.

`void telnet_command(tnp_handle_t hand)`

Run the Telnet command mode for this session. This provides a Telnet interactive shell. This call creates another Telnet client task, which is needed to read from `stdin` and run the Telnet protocol. It returns to the caller after you exit the Telnet session.

This call uses the `stdio` provided at the creation of the handler.

`hand`	Specifies the client API handle.

pHILE+ Independence

You do not need to have pHILE+ component to use some of the Internet Applications modules. The Internet Applications components that do require you to use pHILE+ are some commands of pSH, FTP, TFTP, FTPD, TFTPD, and NFSD. However, the Internet Applications components that do not require you to use pHILE+ are some pSH commands, Telnet, routed, Domain Name System (DNS) Static Name Resolver, and Dynamic Host Configuration Protocol (DHCP).

NOTE: You do not have to use `nuapi_installfs()` call to use Internet Applications without any file system related functions. The user applications can be linked without pHILE+ component in pSOSystem to some of the Internet Applications modules, which are not dependent of pHILE+ component.

If you do use pHILE+ component and your application requires some of the Internet Applications modules that are dependent of pHILE+, you need to invoke `nuapi_installfs()` API call at startup. `nuapi_installfs()` is used as a generic API function. The following structure can be initialized and passed to the `nuapi_installfs()` function:

```
#include <nu_api.h>
typedef struct {
   ULONG(*create_f)(char *name, ULONG expand_unit, ULONG mode);
   ULONG(*remove_f)(char *name);
   ULONG(*open_f)(ULONG *fid, char *name, ULONG mode);
   ULONG(*close_f)(ULONG fid);
   ULONG(*read_f)(ULONG fid, void *buffer, ULONG bcount, ULONG
                 *tcount);
   ULONG(*write_f)(ULONG fid, void *buffer, ULONG bcount);
   ULONG(*lseek_f)(ULONG fid, ULONG position, long offset, ULONG
                  *old_ptr);
   ULONG(*fstat_f)(ULONG fid, struct stat *buf);
   ULONG(*move_f)(char *oldname, char *newname);
   ULONG(*get_fn)(char *name, ULONG *fn);
   ULONG(*open_fn)(ULONG *fid, char *device, ULONG fn, ULONG mode);
   ULONG(*stat_f)(char *file, struct stat *buf);
   ULONG(*stat_vfs)(char *file, struct statvfs *buf);
}nu_fileops_t;

typedef struct{
   ULONG(*make_dir)(char *name, ULONG mode);
   ULONG(*change_dir)(char *name);
   ULONG(*open_dir)(char *dirname, XDIR *dir);
   ULONG(*read_dir)(XDIR *dir, struct dirent *buf);
   ULONG(*close_dir)(XDIR *dir);
}nu_dirops_t;

typedef struct{
   ULONG(*sync_vol)(char *device);
}nu_volops_t;

extern void nuapi_installfs(nu_fileops_t *, nu_dirops_t *,
                            nu_volops_t *);
```

The following code example illustrates how to use `nuapi_installfs()` call to install pHILE+ function calls in the Internet Applications library:

```
/* Set the file system calls to pHILE */
fops.create_f=create_f; fops.remove_f=remove_f; fops.open_f=open_f;
fops.close_f=close_f; fops.read_f=read_f; fops.write_f=write_f;
fops.lseek_f=lseek_f; fops.fstat_f=fstat_f; fops.move_f=move_f;
fops.get_fn=get_fn; fops.open_fn=open_fn; fops.stat_f=stat_f;
fops.stat_vfs=stat_vfs;
dops.make_dir=make_dir; dops.change_dir=change_dir;
dops.open_dir=open_dir;
dops.read_dir=read_dir; dops.close_dir=close_dir;
vops.sync_vol=sync_vol;
nuapi_installfs(&fops, &dops, &vops);
```

Protocol Stack Independence

The Internet Applications library from version 3.0 onwards will work with either pNA+ or TCP/IP for OpEN. A new function is provided to initialize the library with the required socket system calls. Before any other library functions are invoked, this function is called to register the socket calls in the library by the application from the ROOT task.

The `sockcall` structure is initialized with the appropriate socket functions and passed as an argument to the `nuapi_installsock()` function.

NOTE: If you do not invoke this function, pNA+ is assumed to be the default.

The sample application shows the usage of this function.

The API for using this function is defined in the `nuapi.h` file. The following is an example of the structure that is used to initialize socket calls that is used by the Internet Applications library.

```
struct sockcall {
         int pna;
#define NULIB_PNA     1
#define NULIB_OTCP    0
         long (*accept)(int s, struct sockaddr_in *addr, int *addrlen);
         long (*add_ni)(struct ni_init *ni);
         long (*bind)(int s, struct sockaddr_in *addr, int addrlen);
         long (*close)(int s);

         long (*connect)(int s, struct sockaddr_in *addr, int addrlen);
         long (*getpeername)(int s, struct sockaddr_in *addr, int *addrlen);
         long (*getsockname)(int s, struct sockaddr_in *addr, int *addrlen);
         long (*getsockopt)(int s, int level, int optname, char *optval,
               int *optlen);
         long (*get_id)(long *userid, long *groupid, long *);

         long (*ioctl)(int s, int cmd, char *arg);
         long (*listen)(int s, int backlog);
         long (*recv)(int s, char *buf, int len, int flags);

         long (*recvfrom)(int s, char *buf, int len, int flags,
               struct sockaddr_in *from, int *fromlen);
         long (*recvmsg)(int s, struct msghdr *msg, int flags);
         long (*select)(int width, struct fd_set *readset, struct fd_set
               *writeset, struct fd_set *exceptset, struct timeval *timeout);
         long (*send)(int s, char *buf, int len, int flags);
         long (*sendmsg)(int s, struct msghdr *msg, int flags);

         long (*sendto)(int s, char *buf, int len, int flags, struct sockaddr_in
               *to, int tolen);
         long (*setsockopt)(int s, int level, int optname, char *optval,
                int optlen);
         long (*set_id)(long userid, long groupid, long *);
         int (*shr_socket)(int s, unsigned long tid);
         long (*shutdown)(int s, int how);

         int (*socket)(int domain, int type, int protocol);
         mblk_t *(*allocb)(int size, int pri);
         mblk_t *(*esballoc)(unsigned char *base, int size, int pri, frtn_t
                  *frtn);
         void (*freeb)(mblk_t *bp);
         void (*freemsg)(mblk_t *mp);
};

extern void nuapi_installsock(struct sockcall *);
```

Memory Management

The Internet Applications library provides an API to configure memory management functions. You need to configure these functions before using any of the other library functions. The following API is used to configure memory management functions in the Internet Applications library:

```
#include <nu_api.h>

typedef struct {
    void *(*nu_malloc)(size_t );
    void (*nu_free)(void *);
    void *(*nu_realloc)(void *,size_t );
    void *(*nu_calloc)(size_t ,size_t );
} nu_memmgmt_t;

extern void nuapi_memmgmt(nu_memmgmt_t *);
```

The Internet Applications sample applications uses standard pREPC+ functions `malloc()`, `free()`, `realloc()`, and `calloc()` for this purpose. You can use the pREPC+ functions or devise your own memory management, which can provide the same functionality.

NOTE: The functions that you choose must behave similar as pREPC+ functions.

The following is an example of how the memory management API can be used:

```
mcall.nu_malloc = malloc;
mcall.nu_free = free;
mcall.nu_realloc = realloc;
mcall.nu_calloc = calloc;

nuapi_memmgmt(&mcall);
```

NOTE: If you do not invoke `nuapi_memmgmt()`, pREPC+ functions are used as the default. If you are using a C++ package for pSOSystem v2.2, use `lc_malloc()`, `lc_free()`, `lc_realloc()`, and `calloc()` functions.

Error Codes for Client APIs

The error values are defined in `netutils.h`. The following error codes are returned by the Client API calls:

`CLIENT_SUCCESS`	Client API successful.
`CLIENT_INVALID`	One or more arguments that are passed to the client API call are invalid or out of range.
`CLIENT_PROTOERR`	Due to some invalid condition, the client API call resulted in a client protocol error. For example, the remote server rejected a FTP port command.
`CLIENT_SYSERR`	The client API call failed due to some system resource error, such as not able to allocate memory or a data structure.
`CLIENT_FILEERR`	This is due to a file system call failure. Check `errno` for details.
`CLIENT_PNAERR`	This is due to pNA+ system call failure. Check `errno` for details.
`CLIENT_GENERR`	General error. These errors are due to some invalid states which should not occur during normal execution.

DHCP Client

Description

The Dynamic Host Configuration Protocol (DHCP) provides a framework for passing configuration information to the TCP/IP hosts on the network. DHCP is contained in pSOSystem’s Internet Applications product.

DHCP is based on the extension to the BOOTP protocol, which provides framework for dynamic usage of IP addresses on a network. With dynamic addressing, a device can have a different IP address every time it connects to the network.

DHCP supports the following three mechanisms for allocating IP addresses:

- Automatic allocation — Assigns a permanent IP address to a host.
- Dynamic allocation — Assigns an IP address to a host for a limited period of time or until the host relinquishes the address.
- Manual allocation — Assigns an IP address configured by the administrator. DHCP is used to simply convey the address.

The DHCP clients can request for an IP address and other configuration parameters similar to BOOTP from a DHCP server. The configuration parameters will have an associated lease-time, after which the client should either release the configuration (for example, so it can be reassigned to another client by the server) or request the server to renew the lease. See RFC1541 for additional information about the DHCP protocol.

NOTE: The DHCP Client is provided in the Internet Applications library.

The DHCP function is implemented as part of two tasks: main DHCP task and DHCP receiver task. The main task is responsible for processing user events, which include requests to start to process DHCP on any interface or to stop DHCP. This task also handles timer events for any retransmits or lease renewals in the DHCP protocol.

The DHCP receiver task is responsible for receiving DHCP responses and passing them to the main task.

System/Resource Requirements

DHCP Client requires the following:

- pSOS+ Real-Time Kernel.
- pNA+ TCP/IP Network Manager.
- pREPC+ Run-Time C Library.

In addition, DHCP requires the following system resources:

- One queue is used to communicate between DHCP main task and other tasks using DHCP services.
- One User Datagram Protocol (UDP) socket for the DHCP task and another UDP socket for the DHCP receiver task. The DHCP framework also opens another UDP socket whenever there is any IP address change on an interface, and closes it immediately after changing the IP address.
- Dynamic memory is used in DHCP to allocate 48 bytes of memory on each interface on which it is supported. Also, for each DHCP request that is started on an interface, it allocates another 100 bytes of dynamic memory. It is freed when DHCP is stopped on that interface. Dynamic memory is allocated from Region 0. DHCP also allocates memory for constructing packets and frees it after sending them on the network.
- Each of the two DHCP tasks run in supervisor mode. Each task requires 4Kbytes of supervisor stack.

Configuration and User Interface

As part of the Internet Applications library, the following API is provided for using DHCP client services:

```
#include <dhcpcfg.h>
struct dhcp {
   unsigned long task_prio; /* DHCP task priority */
   unsigned long max_ifs;   /* max interfaces needing
                               DHCP support */
   unsigned long log_msgs;  /* to enable debugging info */
};
```

`task_prio`	Specifies the priority of the DHCP task. This should be less than the priority of any other task that will call DHCP API functions.
`max_ifs`	Specifies the maximum number of network interfaces that will be supported by DHCP.
`log_msgs`	Indicates the progress of DHCP operation to enable debugging messages. The setting is zero to disable logging.

The interface parameter in the following APIs refer to the pNA+ interface number.

The following API functions are provided for using DHCP client protocol:

```
int DHCP_start(dhcpcfg_t *dhcpcfg)
```

This routine is called by an user application (for example, preferably from a ROOT task) to initialize and start DHCP operations. In addition, it starts a DHCP task that is responsible for all of the DHCP operations and another DHCP task when it needs to start processing DHCP responses.

`dhcpcfg`	Specifies the DHCP configuration.

This routine returns zero on success or non-zero value if DHCP fails to start. This call fails if it is unable to start a DHCP task or create a pSOS+ queue.

```
int DHCP_halt(void)
```

When it no longer needs to have DHCP services, this routine is called by an user application to stop DHCP. It also deletes the DHCP tasks and frees up the memory resources.

This routine returns zero on success or non-zero value on failure.

`int DHCP_coldint(char ifno, dhcp_handler_t *hand)`

This routine will start a DHCP operation on one of the network interfaces numbered `ifno`. This will initiate the DHCP state machine for that network interface in INIT state. See RFC1541 for an explanation of the DHCP states.

Before calling this function, the IP address for this network interface is 0.0.0.0. This routine is responsible for allocating an IP address by requesting the DHCP server on the network, and configuring the interface with this address. You need to provide a handle function, which is invoked whenever the DHCP state machine needs to notify you for any event. Once this is invoked on an interface, the DHCP library is also responsible for renewing the lease from the server.

`ifno`	Specifies the interface number on which to start DHCP.
`hand`	Specifies the handler that you provide to be invoked whenever the library needs to have user intervention.

The `dhcp_handler_t` handler is the following type:

```
typedef void (*dhcp_handler_t)(void       *app_cookie,
                               dhcp_hdr_t *msg_cookie,
                               void *state_cookie,
                               dhcp_event_t event,
                               struct in_addr *server)
```

The parameter `app_cookie` is for future extensions and should not be used.

The parameter `msg_cookie` is the pointer to the DHCP packet header.

The parameter `state_cookie` should be stored by the application and they should be used to invoke `DHCP_request()` to accept a DHCP offer.

The parameter `server` is the IP address of the server that made the DHCP offer. This should be used by the application to make a decision whether to accept the offer or wait for more offers. This address is in network byte order.

This routine allows you to be notified of certain events. You may not need to take any action for all of the events. The parameter event indicates the type of event.

The following are the list of events that are defined in `dhcpcfg.h`:

`DHCP_EVT_GOT_RESPONSE`	Indicates the received packet is an OFFER.
`DHCP_EVT_TIMEOUT_CHOOSE_OFFER`	Indicates the timeout is up. Pick an OFFER.
`DHCP_EVT_TIMEOUT_NO_RESPONSE`	Indicates we never got an OFFER.
`DHCP_EVT_LEASE_RENEWING`	Starts address requisition.
`DHCP_EVT_LEASE_REBINDING`	Broadcasts address requisition.
`DHCP_EVT_ADDRESS_LOST`	Indicates the lease is up or extension was NAKd.
`DHCP_EVT_CANT_ADD_ADDR_MASK`	Indicates the `add addr` failed — bad netmask.
`DHCP_EVT_CANT_ADD_ADDR_BRD`	Indicates the `add addr` failed — bad `bcast addr`.
`DHCP_EVT_CANT_ADD_ADDR_MEM`	Indicates the `add addr` failed — `malloc` failure.
`DHCP_EVT_CANT_ADD_ADDR_UNK`	Indicates the `add addr` failed — unknown reason.
`DHCP_EVT_BOUND`	Indicates the `add addr` succeeded.
`DHCP_EVT_MSG_TO_GO`	Indicates the DHCP message is about to be sent.
`DHCP_EVT_DECLINE`	Indicates the DHCPDECLINE is about to be sent.
`DHCP_EVT_BOOTP_MSG`	Indicates the packet has no message type option.
`DHCP_EVT_QUITTING`	Quits DHCP operation.

This function returns zero on success and an error code on failure. The error code can be any pSOS+ error.

```
int DHCP_warminit(char ifno, struct in_addr *ipaddr, dhcp_handler_t
                  *hand)
```

This function is similar to `DHCP_coldinit()` except that the DHCP state machine is started in INITREBOOT state rather than INIT state. It is used to renew a previously allocated IP address, and is called on an interface with a preconfigured IP address.

`ifno`	Specifies the interface number on which to start DHCP.
`ipaddr`	Specifies the IP address which the client is requesting to renew (in network byte order). This address should be passed in network byte order.
`hand`	Specifies the handler that you provide to be invoked whenever the library needs to have user intervention. For the list of events, see `DHCP_coldinit()` API function.

This routine allows you to be notified of certain events. You may not need to take any action for a few of the events. These events are defined in `dhcpcfg.h`. See `DHCP_coldint()` API function for the list of the events.

This function returns zero on success and an error code on failure. The error code can be any pSOS+ error.

```
dhcp_err_t dhcp_add_option(dhcp_hdr_t * msg_cookie,
                           bits8_t      opt_tag,
                           bits8_t  opt_len,/*
ignored if tag known */                                bits8_t
* opt_value);
```

This function adds an option to the outgoing packet.

`msg_cookie`	Specifies the pointer to the DHCP packet header.
`opt_tag`	Specifies the DHCP option tag that needs to be added to the outgoing packet. The possible values for the DHCP option tags can be found in the `dhcpcfg.h` include file.
`opt_len`	Indicates the length of the DHCP option.

`opt_value`	Indicates the value of the DHCP option.

This function returns zero on success or any of the `dhcp_err_t` error codes on failure. These error codes are described in the `dhcpcfg.h` file.

```
dhcp_err_t dhcp_get_option(dhcp_hdr_t  *msg_cookie,
                           bits8_t     tag,
                           int         * buflen,
                           bits8_t     * buffer);
```

This function gets an option out of a received DHCP packet.

`msg_cookie`	Specifies the pointer to the DHCP packet header.
`opt_tag`	Specifies the DHCP option tag that needs to be read from the DHCP packet. The possible values for the DHCP option tags can be found in the `dhcpcfg.h` include file.
`buflen`	Points to the maximum length of the buffer and upon return from this function, it contains the size of the option copied in the buffer.
`buffer`	Specifies the pointer to the value of the DHCP option once the function returns. This buffer should be passed by you.

`int DHCP_request(void *state_cookie,dhcp_hdr_t *offer)`

This routine needs to be called to send a DHCP REQUEST packet by responding to one or more DHCP OFFER messages received by the client. The decision on which the DHCP client accepts the DHCP OFFER is left to the user. You need to call `DHCP_request()` with the OFFER that needs to be accepted.

`state_cookie`	Specifies the `state_cookie` passed by the library to the DHCP handler.
`offer`	Specifies the pointer to the DHCP packet header passed by the library to the DHCP handler.

`int DHCP_decline(char ifno)`

When the client does not like some of the options in DHCP ACK packet, this routine needs to be called to send DHCP DECLINE packet to the server.

`ifno`	Specifies the interface number on which DHCP decline is sent.

This function returns zero on success and an error code on failure. The error code can be any pSOS+ error.

`int DHCP_release(char ifno)`

This routine needs to be called to send DHCP RELEASE packet to release an IP address that is no longer needed.

`ifno`	Specifies the interface number on which the DHCP release should be sent.

This function returns zero on success and an error code on failure. The error code can be any pSOS+ error.

```
int DHCP_rmintf(char ifno)
```

This routine is called to disable DHCP on a particular interface. By using this routine, it might lead to a protocol violation (for example, DHCP lease expires); therefore, this should be used with caution.

`ifno`	Specifies the interface number on which DHCP should be shutdown.

This function returns zero on success and an error code on failure. The error code can be any pSOS+ error.

DNS and Static Name Resolver

Description

DNS and Static Name resolver is contained in pSOSystem's Internet Applications product.

Name resolution provides name-to-address mapping service to the application programs. Currently, only host name resolution is supported. The resolution is done by looking up either the statically configured tables, which you initialize at startup and modify, or by using the dynamic name resolution protocol called Domain Name Service (DNS).

The Resolver service provides API's for configuring static host and network tables, and also the APIs for name resolution by using either static tables or the DNS protocol. The Resolver functions are executed as part of the user's calling tasks. One Resolver task RESt is created to maintain DNS cache entries.

NOTE: The Resolver is provided in the Internet Applications library.

System/Resource Requirements

The following pSOSystem resources are required to implement the resolver:

- pSOS+ Real-Time Kernel.
- pNA+ TCP/IP Network Manager.
- pREPC+ Run-Time C Library.

In addition, the resolver requires the following system resources:

- The Resolver allocates 28 bytes of dynamic memory from Region 0 for each entry of the static host or network table. It also allocates 100 bytes for each entry of the search list and 600 bytes for each DNS cache entry. Therefore, the total memory depends on the number of each of these parameters that you configure at the Resolver startup.
- The Resolver dynamically opens one UDP socket for sending DNS requests, and closes this socket after receiving a response or timeout.
- One semaphore RESs is used by the Resolver for protecting critical regions.
- The Resolver task RESt requires 2Kbytes of task stack.

Name resolution is done using either user-configured static tables or by using DNS. The system can be configured so that either of the methods or both are used to do the name resolution.

Before using name resolution APIs, you need to initialize the Resolver by calling `res_start()` by passing `rescfg_t` structure. This structure includes parameters such as the search priority. All the other subsequent resolution requests are handled by using these parameters. Subsequently, you can change these parameters by using `res_modify()`.

Configuration and User Interface

The name resolver is started by invoking the `res_start()` function with the specific application parameters. Example 1-1 illustrates how the parameters are passed.

EXAMPLE 1-1: Name Resolver Structure

```
#include <rescfg.h>
struct rescfg_t{
   long     res_priority;           /* priorities for resolution (encoded) */
#define     RES_STATIC 0x01
#define     RES_DNS    0x02
   long     res_dns_task_prio;      /* priority for resolver timer
                                       task */
   int      res_dns_query_mode;     /* Query mode */
#define     DNS_MODE_NOCHG            0x0000
#define     DNS_MODE_NOCACHE          0x0001
#define     DNS_MODE_CACHE_NET        0x0002
#define     DNS_MODE_CACHE_NONET      0x0003
   long     res_dns_max_ttl;        /* maximum TTL for DNS RRs */
   long     res_dns_res_timeout;    /* wait period for resolver
                                       timeouts */
   long     res_dns_res_retxmits;   /* number of times resolver
                                       retransmits */
   long     res_dns_max_cache;      /* maximum Cache entries */
   long     res_dns_no_servers;     /* No of DNS servers */
   long     *res_dns_servers;       /* List of DNS servers to probe */
   char     **res_dns_search_path;  /* DNS search path */
   long     res_max_hosttab;        /* Maximum entries in static host
                                       table */
   long     res_max_nettab;         /* Maximum entries in static net table */
   long     res_dns_marktime;       /* Mark time in seconds for DNS cache */
   long     res_dns_sweeptime       /* Sweep time in seconds for DNS cache */
};
typedef struct rescfg_t rescfg_t;
```

The following describes the configuration parameters used for name resolvers.

`res_priority`	Defines the priority at which the name resolution works. The possible values are encoded from `RES_DNS` (DNS resolver) and `RES_STATIC` (static resolver). The least significant byte (LSB) signifies the first preferred method. The most significant byte determines the least preferred method of name resolution. Currently, the two methods that are supported are DNS and static resolution (`RES_DNS` and `RES_STATIC`). Therefore, the following are the four possible values: `RES_STATIC` — Searches only static tables. `RES_DNS` — Searches only DNS. `RES_STATIC|(RES_DNS<<8)` — Searches static tables first then DNS next. `RES_DNS|(RES_STATIC<<8)` — Searches DNS first then static tables next.
`res_dns_task_prio`	Defines the task priority of the DNS daemon task `RESt`.
`res_dns_query_mode`	Defines the query mode for the DNS name resolver. The possible values are `DNS_MODE_NOCACHE` and `DNS_MODE_CACHE_NET` that disable the cache and enable the cache respectively, and `DNS_MODE_NONET` that disables requests going to the network.
`res_dns_max_ttl`	Defines the maximum time-to-live (TTL) for the DNS cache entries (in seconds).
`res_dns_timeout`	Defines the time-out value in milliseconds for DNS queries. The minimum value is 5000 (5 seconds). Values set to less than the minimum are reset to 5000.
`res_dns_max_rexmits`	Defines the maximum number of retransmits for DNS queries before timing out. For each retransmit, the time out value increases.
`res_dns_max_cache`	Defines the maximum entries in the DNS cache. See *System/Resource Requirements* section for details on memory requirements for cache entries.
`res_dns_no_servers`	Defines the number of DNS servers configured. A maximum of ten servers can be specified.
`res_dns_servers`	Lists the DNS servers configured. These servers are specified using their IP address in the network byte order.

`res_dns_search_path`	Defines the list of DNS names that are concatenated to the host name strings for searching. The list configured here determines the stack space requirement of the calling task for DNS queries. If this entry is NULL, only fully qualified names (for example, chief.isi.com) are supported. If more entries are listed in the search path, the DNS queries may take more time to respond and also the calling task needs more stack space. A maximum of six entries can be given in a search path.
`res_max_hosttab`	Defines the maximum entries in the static host table. See *System/Resource Requirements* section for information about static host tables.
`res_max_nettab`	Defines the maximum entries in the static network table. This is for future extensions and should be set to zero.
`res_dns_marktime`	Specifies the time in seconds after which DNS task RESt wakes up and searches the DNS cache entries for any timeout elements. They will be marked for subsequent deletion.
`res_dns_sweeptime`	Specifies the time in seconds after which the DNS task wakes up and deletes the timed out cache entries.

The resolver is initialized by calling `res_start(rescfg_t *cfg)` function, which starts the DNS daemon task, allocates required memory for static tables, and initializes the resolver parameters.

If the resolver is started successfully, `res_start()` returns zero; otherwise, it returns a non-zero value on failure. The error value can be E_RESTASK_STARTUP if the DNS task is unable to start or receive any pSOS+ error. All error codes specific to Internet Applications reside in <`netutils.h`>.

Subsequently, some of the resolver parameters can be changed by calling `res_modify(rescfg_t *cfg)`. The parameters that can be changed using `res_modify()` are `res_priority`, `res_dns_query_mode`, and `res_dns_servers`.

The following example shows the usage of both `res_start(rescfg_t *cfg)` and `res_modify(rescfg_t *cfg)` functions:

```
rescfg_t cfg;
cfg.res_priority = RES_STATIC<<8|RES_DNS;
cfg.res_xxx = yyy; /* other parameters related to resolver
                      configuration */


res_start(&cfg);
```

The host entry structure, which is found in rescfg.h, is shown in the following example:

```
#include <rescfg.h>
struct hostent
{
   char *h_name;    /* official name of host */
   char **h_aliases /* alias list */
   int h_addrtype;  /* host address type */
   int h_length;    /* length of address */
#define MAX_ADDR_LIST 10
   char *h_addr_list[MAX_ADDR_LIST]; /* list of addresses from name
                                        server */
#define h_addr h_addr_list[0] /* address, for backward
                                 compatibility */
};
```

The following API routines are provided for name-to-address mapping:

```
int gethostbyname(char *name, struct hostent *hostp)
```

Given the hostname, this routine finds the corresponding IP address and returns it.

name	Specifies the name of the host whose IP address is to be resolved.
hostp	Specifies the pointer to a structure that is filled with the resolved address. Note that h_addr_list[] array needs to be initialized. See the example in the following sections.

This routine returns zero on success or -1 on failure if it is not able to resolve the name.

```
int gethostbyaddr(char *addr, int len, int type, struct hostent
                  *hostp)
```

Given the address, this routine finds the corresponding host name and returns it.

addr	Specifies the pointer to the address to be resolved to a name.
len	Specifies the length of the address (for example, 4 for IPv4 addresses).
type	Specifies the type of the address format. This is only for the internet format that is supported and is set to AF_INET.

`hostp`	Specifies the pointer to a structure that is filled with the resolved name. You should preallocate the buffer to store the resolved name. The resolved name can be up to 255 characters long (See Example 1-2).

This routine returns zero on success or -1 on failure if it is not able to resolve the address.

The following API routines are provided to configure and search the static host table, which provides mapping between the hostname and the ipaddresses.

`set_hostentry(unsigned long ipadd, char *hostname)`

Adds an entry for the `ipadd` and `hostname`pairs in the static host table.

`ipadd`	Specifies the IP address of the host.
`hostname`	Specifies the name of the host as a NULL terminated string up to 16 characters.

This routine returns zero on success; otherwise, it returns a non-zero value on failure. The possible error values are:

`E_RESTASK_INVALID`	Determines that one or more parameters are invalid.
`E_RESHOST_FULL`	Determines if the table is full.
pSOS+ Error Codes	See *pSOSystem System Calls*, Appendix B, *Error Codes*, *pSOS+ Error Codes* section for all the possible error codes.

`del_hostentry(unsigned long ipadd)`

Removes the entry whose IP address is `ipadd`.

`ipadd`	Specifies IP address of the host.

This routine returns zero on success; otherwise, it returns a non-zero value on failure. The possible error values are:

`E_RESTASK_INVALID`	Determines that one or more parameters are invalid.
`E_RESHOST_FULL`	Determines if the table is full.
pSOS+ Error Codes	See *pSOSystem System Calls*, Appendix B, *Error Codes*, *pSOS+ Error Codes* section for all the possible error codes.

`get_hentbyname(char *hostname, unsigned long *addr)`

Returns the host IP address for the given name. Only the static table is searched, and you are required to use a more generic interface `gethostbyname()` if DNS resolution is required.

`hostname`	Specifies the name of the host.
`addr`	Specifies the pointer to the address to be retrieved.

This routine returns zero on success; otherwise, it returns a non-zero value on failure. The possible error values are:

`E_RESTASK_INVALID`	Determines that one or more parameters are invalid.
`E_RESHOST_FULL`	Determines if the table is full.
pSOS+ Error Codes	See *pSOSystem System Calls*, Appendix B, *Error Codes*, *pSOS+ Error Codes* section for all the possible error codes.

`get_hentbyaddr(unsigned long addr, char *hostname)`

Returns the host name given its IP address. Only the static table is searched, and you are required to use a more generic interface `gethostbyaddress()` if DNS resolution is required.

`addr`	Specifies the address of the host.
`hostname`	Specifies the pointer to the hostname to be retrieved.

This routine returns zero on success; otherwise, it returns a non-zero value on failure. The possible error values are:

`E_RESTASK_INVALID`	Determines that one or more parameters are invalid.
`E_RESHOST_FULL`	Determines if the table is full.
pSOS+ Error Codes	See *pSOSystem System Calls*, Appendix B, *Error Codes*, *pSOS+ Error Codes* section for all the possible error codes.

The usage of the functions differ from their counterparts on UNIX systems. Example 1-2 on page 1-49 illustrates the code segments for the usage of `gethost-byname()` and `gethostbyaddr()` functions.

NOTE: None of the Resolver API functions can be called from Interrupt Service Routines (ISRs).

EXAMPLE 1-2: Resolver Example

```
#include <rescfg.h>

/* To resolve name to address */
name_to_address(char *name) /* name to be resolved */
{
   unsigned long ipaddr;
   struct hostent hostp;
   ....

   memset((char *)&hostp, 0, sizeof(struct hostent));
   hostp.h_addr_list[0] = (char *)&ipaddr;
   if (gethostbyname(name, &hostp))
           printf("Host address not found\n");
   else
           printf("Hostname: %s, Hostaddress: 0x%x\n", name,
ipaddr);

           ...

}

/* To resolve address to name */
address_to_name(unsigned long ipaddr) /* address to be resolved */
{
   struct hostent hostp;
   char hostname[256];
   ....

   hostp.h_name = hostname;
   hostp.h_addr_list[0] = (char *)&ipaddr;
   if (gethostbyaddr((char *)&ipaddr, sizeof(long), AF_INET,
&hostp))
           printf("Host name not found\n");
   else
           printf("Hostname: %s, Hostaddress: 0x%x\n", hostname,
ipaddr);

   ....
}
```

NOTE: All of the IP addresses mentioned in this section will either expect or pass the addresses in network byte order.

FTP Client

Description

The FTP (File Transfer Protocol) Client contained in the Internet Applications 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 or a hostname if Name Resolver is configured.

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 1-51.

If the command contains arguments, FTP executes an `open` command with those arguments. See *FTP Commands* on page 1-51 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 Internet Applications 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.

In addition, each session of the FTP requires the following system resources:

- 8978 bytes of dynamic memory allocated from Region 0. This is allocated to store session information and freed when the session exits.
- Two TCP sockets. One is used for FTP control connection. The other is used for data connection. Both sockets are closed when the session exits.
- The stack space needs to be configured by the user. If the user stack size is set to -1, the task is started in supervisor mode.

pSH+ starts FTP Client by calling `ftp_main()`. The Internet Applications library 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},
      {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. The `local_file` parameters can be also a device number (for example, "13.0").

`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`). `host` can be the IP address of the FTP server or hostname if Name Resolver is configured.
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 Limitations

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.

FTP Server

Description

FTP Server is contained in pSOSystem's Internet Applications product.

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 Internet Applications 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 task 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 table should be statically allocated by the application and passed to the library. 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;          /* ptr list of trusted clients */
   ulist_t *ulist;        /* ptr list of trusted users */
   int  abort_cmd;        /* To support ABORT while data xfer */
   long reserved[1];      /* reserved for future use, 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`	Allows you to put a list of IP addresses in dot notation with a NULL at the end. If this field is zero, 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 zero, 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}
   }
```

`abort_cmd`	Supports ABORT command if it is set to one while conducting a file transfer. This enables you to abort a file transfer (from a client) in a graceful manner. However, enabling this option will slow down the FTP transfer rate.
`reserved`	Reserved for future use, and each must be zero.

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 zero; otherwise, it returns a non-zero value on failure. The error value can be any pSOS+ error.

Configuration Table 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.

EXAMPLE 1-3: apps/netutils/root.c file example

```
#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");
   }
```

NOTE: The FTP server can be used to transfer files to or from regular files or pSOS+ devices (for example, “13.0”).

Loader

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 1-79. 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 ELF format. 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.
- ELF object files containing absolute code.
- ELF 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 host 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 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/libloadr.a`	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/beginapi.s`	Application startup file for use by position- independent loadable applications (similar to the `begina.s` file provided with pSOSystem).
`configs/std/loadable.lnk`	is a linker command file for linking relocatable and position independent applications.

The loader contains two user-configurable modules. One supports the loading of Motorola S-records and the other supports the loading of ELF 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_ELF_MODULE YES     /* ELF file 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_ELF_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/sunos) # if csh is the working shell
```

or

```
PATH=$PATH:$PSS_ROOT/bin/sunos # 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, libloadr.a.

The `load()` function is provided for loading *loadable applications*. These applications may remain resident on the target forever, or only temporarily. L*oadable application*s 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 pHILE+ `read_f()` function 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 `ELF`) 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 ELF, which correspond to Motorola S-record or ELF formats, respectively. This is a read-only element.
code_type	Can take one of the values: LD_ABSOLUTE, LD_PIC, 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. LD_PIC implies that the object file is position independent. 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 or LD_PIC.

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

The load() function 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.

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 zero; 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_INVALID	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 ELF 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_BADOP	A bad I/O operation, like seeking back on an I/O device, was attempted internally by the loader.
ERR_UNDEFSYM	The object file contains unresolved symbols.
ERR_NOSYMTAB	The symbol table needed by the relocatable loader is missing in the object file.
ERR_RELOC	Relocation error due to an unknown relocation type or due to incorrect relocatable value for a relocation entry.
ERR_UNKWNSZ	Missing or incomplete section in the object file.

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

The unload() function 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 or LD_PIC, 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 zero; 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

The `release()` function 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 zero; 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 ELF 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 zero.

`-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 <u>global</u> 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 <u>floating point units</u>. 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 zero.
`-s` *`supv_stack_size`*	Specifies the size of the task's supervisor stack in bytes, and must be greater than 512. If unspecified, it is set to zero.
`-u` *`user_stack_size`*	Specifies the size of the task's user stack in bytes, and may be zero if the task executes only in supervisor mode. If unspecified, it is set to zero.
`-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 zero. When detecting the zero 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 zero upon successful execution; otherwise, it exits with exit status one 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 workstations, Hewlett Packard series 700 workstations, and machines running Windows 95 or NT.

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 ELF 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 ELF 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 ELF 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.elf` (the ELF 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.elf`) 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 Chapter 2. 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.

pLM+

Description

The process of building a shared library and an application that calls it in the host is shown in Figure 1-1.

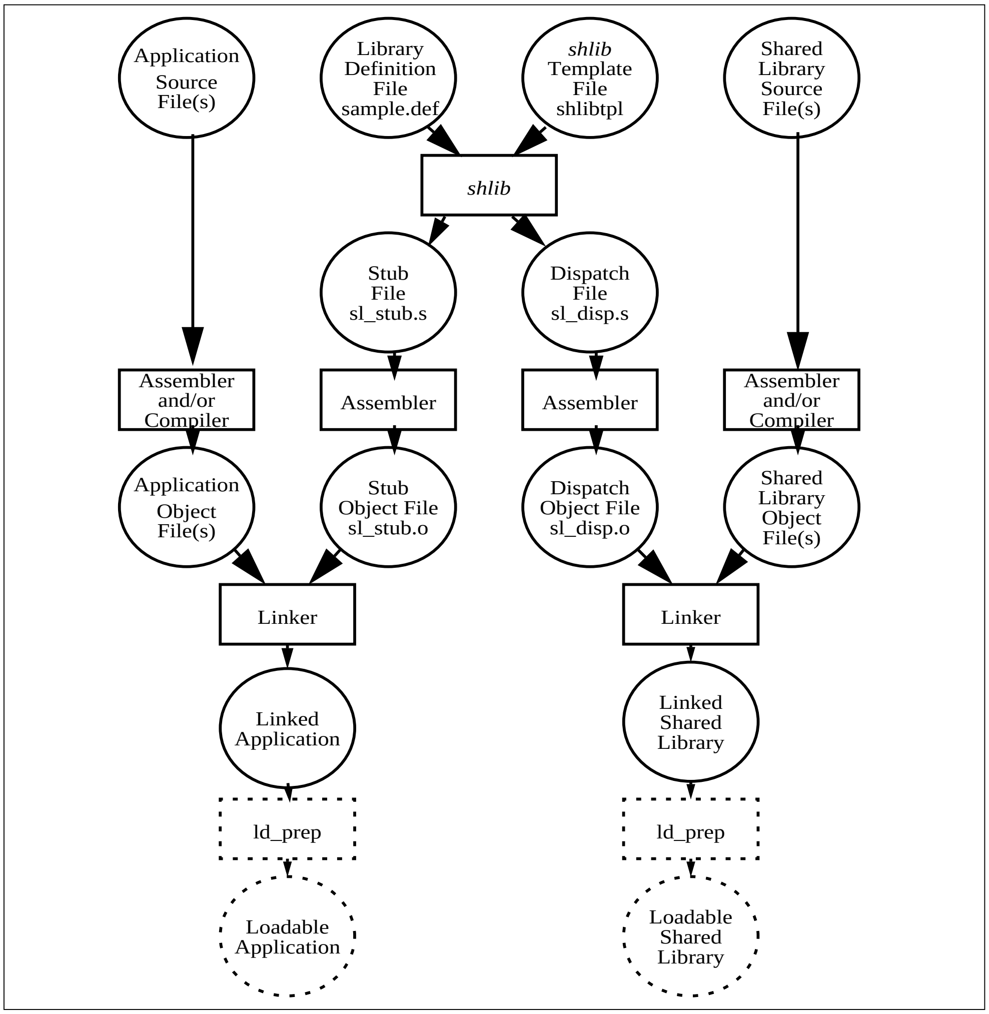

Application Source File(s)
Library Definition File sample.def
shlib Template File shlibtpl
Shared Library Source File(s)
shlib
Stub File sl_stub.s
Dispatch File sl_disp.s
Assembler and/or Compiler
Assembler
Assembler
Assembler and/or Compiler
Application Object File(s)
Stub Object File sl_stub.o
Dispatch Object File sl_disp.o
Shared Library Object File(s)
Linker
Linker
Linked Application
Linked Shared Library
ld_prep
ld_prep
Loadable Application
Loadable Shared Library

FIGURE 1-1 Building a Shared Library and an Application That Calls It

The step that makes a loadable application or a loadable shared library is needed only if the *pSOSystem* loader would be used to load either one of them. A host utility, `shlib`, is used in building a shared library. It takes as input a library definition file, and a shared library template file. It outputs two files: a stub file (in assembly), and a dispatch header file (in assembly).

The syntax of the `shlib` command is below.

```
shlib [-s stub-file] [-d dispatch-file] [-notarget-symtbl]
 [-t shlib-template-file] library-definition-file
```

`shlib` interprets its command line as follows. The stub file is written to the file specified by the `-s` option, otherwise to `sl_stub.s` in the current directory. The dispatch header file is written to the file specified by the `-d` option, otherwise to `sl_disp.s` in the current directory. The `shlib` template file is specified by the `-t` option. If `-t` option is present, the named file is used as the `shlib` template file. Otherwise, as a default, file `shlibtpl` in the current directory is used as the `shlib` template file.

The library definition file describes the shared library by listing the functions that are exported by the library. The library definition file is written by the programmer of the shared library. The `shlib` template file customizes the output assembly files (stub and dispatch header) for a specific target processor and tool chain. An `shlib` template file will be provided for supported processors and tool chains.

The stub file is responsible for locating and calling the correct function in the shared library. It is assembled and linked with the code images that call the shared library. Note that only the stub object is linked with the executables and not the actual code of the shared library functions. The stub file contains one label and associated code, called the binding, for each function exported by the shared library. The label matches the function name exactly so that the linker will resolve all the application code references to that function using the label in the binding.

Application and library programmers using implicit calling method must decide which shared libraries that the program will be using at the program build time and link in the appropriate stubs of the libraries. Note that a code image can not link with two different shared libraries that happen to export the same symbol. Also, it is not possible to interpose on the shared library symbols. Neither the finally linked application nor the finally linked shared libraries can have unresolved symbols. If a program makefile is modified to use the shared libraries instead of static libraries, the program may not link the same way. This is due to the fact that the search order may not be the same and also there may be more or less number of exported symbols brought into the name space from the stubs of the shared libraries. If a shared library uses other shared libraries by implicit calling method, its library definition file must contain the names and versions of the dependent libraries (first level) and the library must be appropriately linked with their stub objects as explained above. Independent shared library vendors may ship their definition file and object files of the shared library to the customers, if they want to leave some symbols unresolved. Customers can complete the build of the shared library at their installation by re-solving the unresolved symbols through the linker, either from other static libraries or shared libraries.

The dispatch header file is assembled and linked with the object files containing the executable code for the library such that the dispatch header object is placed at the start of the code segment of the final shared library. This header object contains the C structure `sl_header` shown below:

```
typedef USHORT index_t;

typedef ULONG offest_t;

typedef struct sl_dependent {
   offset_t deplibname;  /* Dependent library: Name
                           (Offset of string) */
   offset_t deplibinfo;  /* Dependent lib: LM_LOADCO call-out
                            info (Offset of function) */
   ULONG deplibscope;    /* Dependent library: Scope */
   ULONG deplibver;      /* Dependent library: Version */
} sl_dependent_t;

typedef struct sl_header {
   UCHAR magic[4];   /* ISI shared library magic number "pLM+" */
   USHORT type;      /* ISI Shared library type */
   USHORT machine;   /* CPU: 0-68K, 1-PPC, 2-MIPS, 3-x86 */
   offset_t name;    /* Library name (Offset of string) */
   ULONG key;        /* Library key */
   ULONG version;    /* Library version number */
   offset_t libinfo; /* Library LM_BINDCO callout info.
                        (Offset of strubg) */
   ULONG ndeplibs;   /* Number of dependent libraries */
   offset_t deplibs; /* Dependent libraries
                        (Offset of array [ndeplibs]
                        of sl_dependent_t */
   UINT   max_id;    /* Maximum symbol ID (Size of sym. tbl.)*/
   offset_t addr_tbl;/* Symbol address table
                        (Offset of array [max_id] of offset */
   offset_ sym_tbl;  /* Symbol name table
                        (Offset of array [max_id] of offset of
                         string) */
   USHORT nbuckets;  /* # of buckets in hash table */
   USHORT ncells;    /* # of hash table cells
                        (max_id+1+nbuckets) */
```

```
    offset_t buckets; /* Hash table buckets
                         (Offset of array [nbuckets] of index_t
                          in cells) */
    offset_t cells;   /* Hash table cells
                         (Offset of array [ncells] of index_t
                          in sym_tbl and addr_tbl) */
    offset_t sl_entry;/* Lib. entry fn ptr
                         (Offset of function) */
    ULONG reserved;   /* Reserved for future use */
} sl_header_t;
```

This structure contains (or points to) information such as the library magic number, type, name, key, version, bind callout info, the names, scopes, versions and load callout infos of the other shared libraries used (directly linked with), the names and addresses of exported functions (or symbols) in the library, hash table and other library attributes. Note that to ensure position independence, all the pointers or addresses are stored as offsets from the beginning of this dispatch header structure. Hence, at run time one must add the address of this structure in memory to any pointer fields' offsets to compute the absolute addresses. During registration, address of this structure is passed to pLM+ which stores it in its table of pointers to library headers at an entry pointed to by the index assigned to the library.

The hash table is generated using an internal hash function shown below for indexing the addresses of the exported symbol efficiently using the symbol names. The `sl_hash` field in the header is set to `0` by `shlib` to denote that pLM+ should use the internal hash function listed below to access the hash table to implement `sl_getsymaddr()` service. If this entry is set by the user to point to a nonzero value, pLM+ will access the hash table using the code pointed by `sl_hash`. However, the user is responsible to hand code the hash table in the library dispatch header in this case.

```
ULONG sl_hash(const char *symname) {
    ULONG key =0;       /* Hash key */
    UCHAR character;    /* One character of symname */

    while ((character = *symname++) != '\0') {
       /* Rotate left 8 bits */
       key = (key << 8) | (key >> (8 * (sizeof(key) - 1)))

       /* Exclusive or in the next character of symname */
       key ^= character;
    }

    /* Key can not be KEY_UNKNOWN (0). */
    if (key == KEY_UNKNOWN)  key = KEY_UNKNOWN + 1;
```

```
   return h;
}
```

The buckets array stores indices that point to the cells array to the beginning of the list of indices of the symbols that have the same hash value. To denote a empty list or the end of a list, the value -1 is used. The symbol indices in the cells array can be used to index into both sym_tbl and addr_tbl. The hash function accepts a symbol name and returns a value that can be used to compute a buckets index. Consequently, if the hashing function returns the value x for some name, buckets[x%nbuckets] gives an index y, into the cells table. If the cells[y] is -1 then the symbol is not in the hash table. Otherwise, if sym_tbl[cells[y]] is not the desired name, then cells[y+1] gives the next symbol with the same hash value. sl_getsymaddr() follows the cells links until either the selected sym_tbl entry is of the desired name or the cells entry contains the value -1. If sym_tbl[cells[y]] is the desired name, then addr_tbl[cells[y]] gives the address (offset) of the symbol of given name.

By default, the symbol table of all exported functions and the hash table will be generated in the dispatch header file. The symbol and hash tables generation can be disabled by using the -notarget-symtbl option. If disabled, the sym_tbl field would point to 0 and the hash table entries nbuckets, ncells, buckets, and cells would all be 0.

The sl_entry field is set to the address (offset) of the library entry function, if defined. Otherwise, it is set to 0. The library entry function is described in the next section.

Library Entry Function

Each shared library can optionally define an entry function in its dispatch header which performs initialization and cleanup. This entry function if defined, as determined by a nonzero pointer in the dispatch header of the library, is called by pLM+ whenever the library is registered, attached, detached and unregistered. The syntax of the entry function is shown below.

```
ULONG <libname>_entry(ULONG index, ULONG event) {
   switch(event) {
       case SL_REGEV:
          /* Do system initialization */
       break;
       case SL_UNREGEV:
          /* Do system cleanup */
       break;
       case SL_ATTACHEV:
          /* Do per process initialization */
```

```
         break;
         case SL_DETACHEV:
            /* Do per process cleanup */
         break;
      }
      return 0;
   }
```

The entry function is called with its event parameter set to the reason for which it is being called, so that the library can perform the appropriate system or per process initialization and cleanups. When a shared library is registered, pLM+ will call the entry function with SL_REGEV as the event parameter. When the library is unregistered pLM+ will call the entry function with SL_UNREGEV as the event parameter. This allows the shared library to perform system initialization and cleanup, if necessary. The entry function is called in the context of the process that is registering or unregistering the library.

Similarly, when a process attaches to the library by calling sl_acquire(), pLM+ will call the entry function with SL_ATTACHEV as the event parameter. When the process detaches from the library by calling sl_release(), pLM+ will call the entry function with SL_DETACHEV as the event parameter. The entry function is called in the context of the process that is attaching to or detaching from the library. This allows the shared library to perform per process initialization and cleanup, if necessary. Since acquiring and releasing libraries may also result in registering and unregistering them, their entry functions are also called with SL_REGEV and SL_UNREGEV event parameters respectively, when that happens.

The index parameter represents the library index assigned by pLM+. This may be used by the library to identify itself. It should not be used within the entry function during SL_REGEV and SL_UNREGEV events. The entry function returns 0 on success and a nonzero value on failure. If a nonzero value is returned, registration and process attach will fail. Process detach and unregistration ignore the return value but they do report an information code, denoting that the entry function returned a nonzero value.

The specifications of library definition file and the *shlib* template file are explained next.

Library Definition File

The library definition file defines the name and version of the library, the names and versions of the first level dependent shared libraries, if any, the names, number of parameters, and IDs of the functions exported by the library, and, optionally, the library's entry function. The syntax for the library definition file is in Figure 1-2 on page 1-87.

Every exported function is assigned a function ID which is used only within the stub file to call the function. If the optional parameter function ID of the <exported function> is not specified by the user, the `function-id` is calculated by `shlib` by adding one to the highest function ID so far or is `0` for the first function. Otherwise the function ID is `function-id`.

The <dependent library list> lists the libraries (direct dependence only) that must be registered in the system before this library can be used. This list will be embedded in the dispatch header in the dispatch file.

The <entry function> can be used to set the entry function pointer in the dispatch header. If set, this function will be called when the library is registered, attached, detached and finally, when unregistered.

The shared library key is a nonzero 32-bit unsigned integer. It is used to uniquely identify a library in the table of pLM+. It is specified by the operational parameter `library-key` of the `LIBRARY` command. If not supplied, it will be computed by `shlib`.

The shared library version number is a nonzero 32-bit unsigned integer. It is written in the library definition file as `major.minor`. The value is constructed by using major as the most significant 16 bits, and minor as the least significant 16 bits. Therefore the range of major and minor is 0 to 65,535 and the range of the version number is 0 to 4,294,967,295. It is specified by the `library-version-number` parameter of the `LIBRARY` command.

The shared library version scope is used to check if the version available is the version desired. It can be one of exact, compatible or any. In exact scope, exact match of the major and minor version numbers is required. In compatible scope, exact major version and equal or higher minor version (than the desired minor version) is required. In any scope, any major and minor version is accepted. It is specified by the `<version scope>` parameter of the `USE` command.

- <library definition file> ::= <library> [<library commands>] [<dependent library list>] <exported function list>
- <library> ::= *LIBRARY* library-name library-version-number <prefix_enable> [library-key] \n
- <prefix_enable> ::= on | off
- <library commands> ::= <library command> [<library commands>]
- <library command> ::= <bindco libinfo> | <entry function>
- <bindco libinfo> ::= *BINDCO_LIBINFO* bindco-libinfo \n
- <entry function> ::= *ENTRY_FUNCTION* function-name \n
- <dependent-library-list> ::= <dependent libary> [<dependent library list>]
- <dependent library> ::= *USE* library-name <version scope> version-number [loadco-libinfo] \n
- <version scope> ::= sl_*exact* | *sl_comp* | sl_*any*
- <exported function list> ::= <exported function> [<exported function list>]
- <exported function> ::= *EXPORT* function-name function-parameters [function-id] \n
- Blank lines are allowed anywhere in the file.
- Any line that starts with # is a comment. Comment lines can be anywhere in the file.
- Any command line in the file can contain a comment on the same line starting with a # after all required parts of the command.

FIGURE 1-2 shlib Library Definition File: Syntax in BNF

To supply information to the callouts in order to locate the libraries at run time, the binding callout (LM_BINDCO) information for the library and the load callout (LM_LOADCO) information for the dependents may optionally be provided. Typically, this parameter can be used to denote the search pathname for locating the library in the system. The library BINDCO information is specified by the optional BINDCO_LIBINFO command. The dependent library LOADCO information is specified by the USE command optional parameter, loadco-libinfo.

shlib Template File

The template file defines the contents of the stub file and the dispatch file. The syntax for the shlib template file is shown in Figure 1-3 on page 1-89. The stub and dispatch output files are computed using <stub file definition> and <dispatch file definition> respectively. The *_ONCE commands are used to output file prologue, starting a table, and file epilogue. The *_FOREACH commands are used to output stubs, and table entries. In order, for each of these commands in the template file, the text lines are copied to the appropriate output file with the pattern substitutions shown in Table 1-1 on page 1-90. They are copied once for *_ONCE commands, and zero or more times for *_FOREACH commands. The *_FOREACH commands are copied once for each occurrence of their unit parameter: hash table bucket, hash table cell, dependent library, or exported function for units BUCKET, CELL, DEPENDENT, and FUNCTION, respectively. STUB_FOREACH allows only FUNCTION. DISPATCH_FOREACH allows any of them.

Any DISPATCH_ONCE and DISPATCH_FOREACH command can be made conditional by adding two optional parameters: <condition type> and <condition value>. The four combinations are ENTRY 1, ENTRY 0, SYMBOL 1, and SYMBOL 0 which output only if an entry function was specified, an entry function was not specified, a target symbol table is produced, and a target symbol table is not produced, respectively.

- <shlib-template-file> ::= <tool chain> <id increment> <function prefix> <stub file definition> <dispatch file definition>
- <tool chain> ::= *TOOL_CHAIN* tool-chain-name \n
- <id increment> ::= *ID_INCREMENT* id-increment \n
- <function prefix> ::= FUNCTION_PREFIX function-prefix \n
- <stub file definition> ::= <stub command> [<stub file definition>]
- <stub command> ::= <stub once> | <stub foreach>
- <stub once> ::= *STUB_ONCE* \n text-lines
- <stub foreach> ::= *STUB_FOREACH* <stub unit> \n text-lines
- <stub unit> ::= *FUNCTION*
- <dispatch file definition> ::= <dispatch command> [<dispatch file definition>]
- <dispatch command> ::= <dispatch once> | <dispatch foreach>
- <dispatch once> ::= *DISPATCH_ONCE* [<condition>] \n text-lines
- <dispatch foreach> ::= *DISPATCH_FOREACH* <dispatch unit> [<condition>] \n text-lines
- <dispatch unit> ::= *BUCKET* | *CELL* | *DEPENDENT* | *FUNCTION*
- <condition> ::= <condition type> <condition value>
- <condition type> ::= *ENTRY* | *SYMBOL*
- <condition value> ::= *0* | *1*
- Blank lines are allowed anywhere before the first <stub file definition> line.
- Any line that starts with # is a comment. Comment lines are allowed anywhere before the first <stub file definition> line.
- Any command line in the file can contain a comment on the same line starting with a # after all required parts of the command. This is not allowed in text-lines fields.

FIGURE 1-3 `shlib` Template File Commands: Syntax in BNF

TABLE 1-1 shlib Pattern Substitutions

Pattern	Replacement
%L	Library: Name
%H	Library: Key Substituted as a decimal number
%V	Library: Version Number Substituted as a hexadecimal number
%I	Library: BINDCO libinfo
%E	Library: Entry function
%P	Library: Function prefix
%M	Maximum valid function ID Substituted as a decimal number
%F	Current function: Name
%D	Current function: ID Substituted as a decimal number
%C	Current function: Index in cell table Substituted as a decimal number
%S	Current function Index in symbol table Substituted as a decimal number
%u	Number of dependent libraries Substituted as a decimal number
%l	Dependent library: Name
%s	Dependent library: Scope Substituted as a decimal number
%v	Dependent library: Version Substituted as a hexadecimal number
%i	Dependent library: LOADCO libinfo
%b	Number of buckets Substituted as a decimal number
%%	%

The TOOL_CHAIN command is required to document the tool chain that is described by the file. It is not used to calculate the output files.

The ID_INCREMENT command is used to scale the function code values used in the stub file. This can avoid scaling them in the code in the stub file that computes a function address. If scaling is not required specify an id-increment of 1.

Some checks are made to partially verify the template file. An error is reported if certain commands are out of order, missing, or repeated. An error is reported if an invalid pattern is found. In any text line field patterns with any of the following characters are valid: "LHVIEMub%." In only *_FOREACH commands with unit DEPENDENT the following patterns are valid: "lsvi." In only *_FOREACH commands with all units except DEPENDENT the following patterns are valid: "FDCS." An error is reported if a required pattern is not found. Table 1-2 shows the required patterns.

TABLE 1-2 shlib Template File: Required Patterns

Command	Required Pattern(s)
DISPATCH_FOREACH BUCKET	C
DISPATCH_FOREACH CELL	S
DISPATCH_FOREACH DEPENDENT	l
DISPATCH_FOREACH FUNCTION	F
DISPATCH_ONCE ENTRY 1	E
STUB_FOREACH_FUNCTION	FD

The contents of a template file is not completely specified by the BNF. Twenty-five commands are needed to produce a stub file, and a dispatch file with all stubs, headers, and tables needed by pLM+. Table 1-3 on page 1-92 lists these twenty-five commands in the order that they appear in the template file. The details are in the next two sections which explain how to write the two main parts of a template file: the <stub file definition> and the <dispatch file definition>.

TABLE 1-3 shlib Template File: Contents

Order	Command	Purpose	Patterns	
			Code or Labels (Necessary)	**Comments or Labels (Optional)**
1	`TOOL_CHAIN`	Documents processor and tool chain.	n/a	
2	`ID_INCREMENT`	Prescales function ID.	n/a	
3	`STUB_ONCE`	Stub file prologue Start assembly file Structure definitions Data section Start of code section		LV
4	`STUB_FOREACH FUNCTION`	Stub definition	`FD`	
5	`STUB_ONCE`	Stub file epilogue Common code used by all stubs Shared library `LM_BINDCO libinfo` End assembly language file	`HILV`	
6	`DISPATCH_ONCE`	Dispatch file prologue Start assembly file Most of dispatch header	`HVuM`	L
7	`DISPATCH_ONCE SYMBOL 1`	Symbol name table and hash table pointers if a target symbol table is produced	`Mb`	L
8	`DISPATCH_ONCE SYMBOL 0`	Null symbol name table and hash table pointers if a target symbol table is not produced		L

TABLE 1-3 shlib Template File: Contents (Continued)

Order	Command	Purpose	Patterns	
			Code or Labels (Necessary)	Comments or Labels (Optional)
9	`DISPATCH_ONCE ENTRY 1`	Entry function pointer if there is an entry function	E	L
10	`DISPATCH_ONCE ENTRY 0`	Null entry function pointer if there is not an entry function		L
11	`DISPATCH_ONCE`	End of dispatch header Start of dependent library table		L
12	`DISPATCH_FOREACH DEPENDENT`	Dependent library entry	`lsv`	L
13	`DISPATCH_ONCE`	Start of symbol address table		L
14	`DISPATCH_FOREACH FUNCTION`	Symbol address table entry	F	L
15	`DISPATCH_ONCE SYMBOL 1`	Start of symbol name table if a target symbol table is produced		L
16	`DISPATCH_FOREACH FUNCTION SYMBOL 1`	Symbol name table entry if a target symbol table is produced	F	LS
17	`DISPATCH_ONCE SYMBOL 1`	Start of cell table if a target symbol table is produced		L
18	`DISPATCH_FOREACH CELL  SYMBOL 1`	Cell table entry if a target symbol table is produced	S	CF

TABLE 1-3 shlib Template File: Contents (Continued)

Order	Command	Purpose	Patterns	
			Code or Labels (Necessary)	Comments or Labels (Optional)
19	DISPATCH_ONCE SYMBOL 1	Start of bucket table if a target symbol table is produced		L
20	DISPATCH_FOREACH BUCKET SYMBOL 1	Bucket table entry if a target symbol table is produced	C	
21	DISPATCH_ONCE	Start of dependent library string table		L
22	DISPATCH_FOREACH DEPENDENT	Dependent library string table entry	li	
23	DISPATCH_ONCE SYMBOL 1	Start of symbol string table if a target symbol table is produced		L
24	DISPATCH_FOREACH FUNCTION SYMBOL 1	Symbol string table entry if a target symbol table is produced	F	S
25	DISPATCH_ONCE	Dispatch file epilogue Shared library name Shared library BINDCO libinfo End assembly file	LI	L

<stub file definition>

A <stub file definition> contains commands 3 through 5 in Table 1-3 on page 1-92. The first STUB_ONCE specifies the stub file prologue which contains assembler pseudo operations necessary to begin an assembler file, for example, selection of the code section. The STUB_FOREACH FUNCTION specifies the stub definition. There are many ways to write a stub definition. The stub definition contains a public data symbol through which the exported function is called (implicitly with a call to sl_bindindex()) and supplies the function ID used to compute the address of the function. The second STUB_ONCE specifies the stub file epilogue. The stub file epilogue contains any common code used by all stub definitions, and assembler pseudo operations necessary to end an assembler file, for example END. One must follow the stub mechanism explained by the shared library implicit calling mechanism to write the stub definition and the stub file epilogue. See, *Shared Library Implicit Calling Mechanism* on page 1-96.

<dispatch file definition>

A <dispatch file definition> contains commands 6 through 25 as shown in Table 1-3 on page 1-92. The dispatch file prologue contains assembler pseudo operations necessary to begin an assembler file, for example, selection of the code section, and a dispatch file header. Table 1-4 shows the contents of the dispatch file header as defined by the C structure `sl_header` and which commands output them. Some of the fields are offsets. These are written as the difference between the target and the label that starts the dispatch file header. Some of the header fields are offsets to additional tables. Table 1-4 also shows the commands that are used to output the start, and each entry of these tables. The commands that output the table start are needed since the header refers to a label that starts the table. Two of these tables contain offsets to entries in two string tables. Table 1-3 on page 1-92 also shows the commands that are used to output the string tables. These start commands could be omitted or just contain comments without labels since the string table start labels are not used. However, the string table entry commands must contain labels since these are used. The dispatch file epilogue contains strings referred to directly by the dispatch file header, and assembler pseudo operations necessary to end an assembler file, END for example.

TABLE 1-4 Dispatch File Header

Offset	Field	Command Order in Table 1-3	
		Output By	**Target Output By**
0	Magic number - pLM+	6	
4	Library Type	6	
6	Processor architecture	6	
8	Offset of Library name	6	25
12	Library key	6	
16	Library version number	6	
20	Offset of Library binding callout info	6	25

TABLE 1-4 Dispatch File Header (Continued)

Offset	Field	Command Order in Table 1-3	
		Output By	Target Output By
24	Number of dependent libraries	6	
28	Office of Dependent library table	6	11 (Start) and 12 (Entry) then their targets 21 (Start) and 22 (Entry)
32	Maximum symbol ID	6	
36	Offsaet of Symbol address table	6	13 (Start) and 14 (Entry)
40	Offset of symbol name table	7 (Included) or 8 (Excluded)	15 (Start) and 16 (Entry) then their targets 23 (Start) and 24 (Entry)
44	# of buckets in hash table	7 (Included) or 8 (Excluded)	
46	# of cells in hash table	7 (Included) or 8 (Excluded)	
48	Offset of bucket table of hash table	7 (Included) or 8 (Excluded)	19 (Start) and 20 (Entry)
52	Offset of cells table of hash table	7 (Included) or 8 (Excluded)	17 (Start) and 18 (Entry)
56	Offset of the entry function	9 (Exists) or 10 (Does not exist)	
60	Reserved	11	

Shared Library Implicit Calling Mechanism

A code image that implicitly calls a shared library must be linked with the stub of the library generated by `shlib`. The stub code is a assembly language file that contains a binding for each function exported by the shared library. Binding is essentially a label and associated code. The label matches the function name exactly so that the linker will resolve application code references to a library function using the label in the binding. For example, if a library contains a function named `foo`, then the binding will contain a label `foo`. When a call to `foo()` is executed from the code image, control comes to the binding for `foo` in the stub. Each binding is responsible for locating and passing control to the appropriate function within the target library.

Each stub file has a single data section variable which, once initialized, holds the index of the library. The first time any function in the library is called, the contents of this variable is indeterminate. This case is detected and `sl_bindindex()` is called to get the index of the library. This index is then stored in the variable for future use. Subsequent calls to the same library use the cached copy of the index stored in the variable.

Using the index of the shared library, the stub code first locates the address of the desired shared library by accessing pLM+'s table directly. Once the address of the library is known, the stub code makes sure that the library at that address is compatible. If it is compatible, then the address of a specific function is located by examining a function address table in the shared library header. For this purpose, a unique function ID (or index) associated with each shared library function is used by the binding for each function. Thus, each binding in the stub file uses a unique function ID. Otherwise, the code for each binding is identical.

Except for on the first call, this entire process may be performed very efficiently. Depending upon the processor, the overhead is about 20 to 100 instructions. On the first call, a single call to `sl_bindindex()` is required.

Note that the validity of the cached index is verified on every implicit function call so that if the library is unregistered or replaced by another library, any subsequent calls will fail and the wrong library or bad address will not be called. To provide fast and reliable validation of the cached index, the name of the associated library is hashed to produce a 32-bit key. The hash function is carefully chosen to minimize or avoid any key collisions. This key is calculated by `shlib` and stored both in the stub file and the library header file. The library key may also be optionally provided by the user in the library definition file which is used by `shlib` instead of the above calculation. When the library is registered, the address of the shared library header is stored in the pLM+ table of pointers to library headers at the corresponding entry.

Every time the library is implicitly called, the cached index is first compared with the maximum allowable value of the pLM+ table indices. If it is too large, it cannot be valid. This comparison avoids wild memory references the first time the stub code is executed. If the index is in an allowable range, the address stored at the entry corresponding to the index in the pLM+ table of library header pointers is checked if valid (nonzero). If the header address is valid, then the key stored in the stub is compared with the library key stored in the header. If the index is out of range or if the header address is invalid (zero) or if the keys do not match, the stub assumes that the index is invalid and calls sl_bindindex() to get the correct index. Note that the *pLM+* table is cleared whenever pSOS+ is initialized so that following a warm restart, all the pLM+ table entries will have address of 0 and appear invalid. If the keys match, the version in the library header is checked for compatibility against the version stored in the stub. If the version in the library header does not have the same major version and the same or higher minor version as the stub version, the stub decides that the library corresponding to the cached index is incompatible and calls sl_bindindex() to get an index that is compatible. If compatible, then the stub code computes the address of the required function using the offsets within the library header and then jumps to that address. It is also possible to ignore the version compatibility checking in the stub code by customizing shlib.

The following C like code demonstrates the above mechanism. However, the real stub code is in assembly. The stub contains a static global variable slib_index to cache the index of the shared library. In the code below, foo is the binding for an exported library function named foo. The binding initializes the function ID and jumps to the common code executed by all bindings in the stub. Note that the jump to the shared library function is shown as a call, but in assembly language it is a real jump and the target shared library function returns directly to the caller of the binding. The parameters a and b to the exported function may be in registers or stack and are unaltered.

```
/* SLIB_NAME - library name in the stub. */
/* SLIB_KEY  - library key in the stub. */
/* SLIB_VER  - library version in the stub. */
/* SLIB_INFO - library LM_BINDCO callout info. in the stub */

extern const NODE_CT *anchor;
static ULONG slib_index; /* Cached index of shared library */

foo:  /* Binding for foo */
   const plm_CT *plmct;      /* pLM+ config table */
   const sl_header *header; /* Shared lib header */
   ULONG err;               /* Error code */
   ULONG fnid;              /* Function id */
   FUNCPTR func_ptr;          /* Pointer to exported function */

   fnid = FOO_ID; /* Code specific to binding for foo */
   goto COMMON;
/* Common code for all bindings. Uses sl_bindindex() implicitly */
COMMON:
TRY_AGAIN:
   if((plmct = anchor->plmct) == 0) /* pLM+ not present? */
```

```
         goto TRY_AGAIN;
      /* Is slib_index within bounds? */
      if (slib_index >= plmct->lm_maxreg)
         goto INIT_BIND; /* No */
      /* Read the address at the pLM+ table entry */
      header = plmct->data->slhdrs[slib_index];
      if(header==0) /* Valid library address? */
        goto INIT_BIND;
         if(header->key != SLIB_KEY) /* Valid key? */
         goto INIT_BIND;   /* No */
      /* Is version of library at index compatible? */
      if(header->version != SLIB_VER) {
        if(MAJOR(header->version) != MAJOR(SLIB_VER))
          goto INIT_BIND;        /* Incompatible */
        /* Minor versions are not equal */
        if(MINOR(header->version) > MINOR(SLIB_VER))
          goto INIT_BIND; /* Not a later revision */
}
      /* Compatible index. Get ptr. to the exported function. */
         func_ptr = (FUNCPTR) (header +
                 ((ULONG *)(header+header->addr_tbl))[fnid]);
      /* Jump to the shared library function. */
         return (*func_ptr)(a, b);

INIT_BIND:

      /* Update slib_index */
         err = sl_bindindex(SLIB_NAME, SL_COMP, SLIB_VER, SLIB_INFO,
                             &slib_index);
      goto TRY_AGAIN;
      /* Never reached */
```

The specification of `sl_bindindex()` is as follows:

```
ULONG sl_bindindex(
      const char *libname, /* Name of the shared library */
      ULONG scope,         /* See below */
      ULONG version,       /* Version of the shared library */
      void *libinfo,       /* Info parameter for LM_BINDCO */
      ULONG *index)         /* Returns index of shared library */
```

`sl_bindindex()` searches the pLM+ table for a registered library having the same name as specified by the null terminated string pointed to by `libname` and a suitable version number specified by `version` depending on the `scope` parameter. The `scope` parameter can be SL_EXACT, SL_COMP, or SL_ANY. In SL_EXACT scope, it checks for the exact match of the major and minor version numbers. In SL_COMP scope, it checks for exact major versions and equal or higher minor versions (revisions) of the registered library. In SL_ANY scope, it ignores the version number and any library with the same name is accepted. If suitable library is found, sl_bindindex() returns its index by storing it into the variable pointed to by index. Also, it returns 0 as the return value. Note that the returned version of the library may be different than the requested version in the case of SL_COMP or SL_ANY scopes. In general, SL_COMP scope is mostly used since the stub code typically checks if the library is of a compatible version. If one needs to customize the stub code to ignore version checking then SL_ANY scope can be used. Or if one needs to customize the stub code for strict exact match, SL_EXACT scope can be used.

sl_bindindex() neither attaches the calling process to the shared library nor it requires the calling process to be attached to the shared library. If a suitable library is not found, sl_bindindex() will call a user established system callout LM_BINDCO, if there is one, with one of the following error codes as its error parameter. Also, sl_bindindex() will pass libname, scope, version and libinfo as parameters to this callout. Note that the bind callout info parameter libinfo is not interpreted by sl_bindindex() in any way, and is passed as is to the callout.

LME_NOTFOUND	Library with the given name is not found.
LME_MAJORVER	The currently registered library has a different major version if scope is SL_EXACT or SL_COMP.
LME_MINORVER	The currently registered library has a different minor version if scope is SL_EXACT. If scope is SL_COMP, the currently registered revision has a lower minor version. In both cases, the currently registered library is of the expected major version.

The callout may handle the error appropriately and return, or simply terminate the process. If the callout returns, lm_bindindex() will search the pLM+ table again for a suitable library. If a suitable library is still not found, sl_bindindex() will return any of the above error codes.

mmulib

Description

The mmulib included in the pSOSystem base package provides the following memory management services for PowerPC 603 and 604 processors:

- Creation of MMU maps (page or block tables for the Memory Management Unit)
- Control of attributes for individual pages or blocks 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*, and/or blocks. Each page or block contains 4 Kbytes and each block contains between 128K and 256M bytes. All memory locations within the same page or block have the same access characteristics. A particular map is implemented via a page or block table stored in memory and Block Address Translation (BAT) registers.

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*.

A page or block (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 or blocks are *undefined*. Any attempt to access a page or block 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 or block. Pages and blocks defined by a map have associated *attributes* that further control access to memory within that page or block. Every defined page or block may have none, some, or all of the following attributes:

`No-access`	Accessing the page or block is disabled. Any attempt to access it causes a hardware exception.
`Read-Only`	The page or block is write protected. Any attempt to write to it causes a hardware exception.
`Cache-Disabled`	The page or block is not cached.
`Copyback`	If cache is enabled, use copyback as opposed to write-through mode.
`Guarded`	The page or block is guarded. Speculative accesses are disabled.

`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. This call 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 or block defined by the map can be examined with the `map_getattr()` call. The attributes associated with any page or block defined by the map can be altered via the `map_setattr()` call. However, undefined pages or blocks 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.

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/Block Attributes

When application code exchanges page or block 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_IBAT	When set, instruction block address translation is used.
MAP_DBAT	When set, data block address translation is used, i.e. this element is a block rather than a page.
MAP_INVBIT	When set, the page or block has no access.
MAP_WPBIT	When set, the page or block is read only. This bit is ignored if MAP_INVBIT is set.
MAP_CIBIT	When set, the page or block is cache inhibited.
MAP_CBBIT	When set, copyback mode is used.
MAP_SERBIT	When set, the page/block is guarded.

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 or block boundary and len must be either a multiple of the page size or the exact block size. Blocks are restricted to sizes from 2^{17} to 2^{28}. The number of IBATs and DBATs are limited as specified by the define statements below. attr is created by OR-ing together any combination of MAP_IBAT, MAP_DBAT, MAP_INVBIT, MAP_WPBIT, MAP_CIBIT, MAP_CBBIT, and MAP_SERBIT.

PAGE_ALIGMENT	Mask that can be used to test an address to see if it is aligned to a page boundary.
PAGE_SIZE	Define that gives the size of a page of memory.
MIN_BAT_SIZE	Define statement that gives the minimum size of a block of memory.
MAX_BAT_SIZE	Define statement that gives the maximum size of a block of memory.
MAX_IBATS	Define statement that gives the maximum number of IBATs allowed.
MAX_DBATS	Define statement that gives the maximum number of DBATs allowed.

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_size	Calculate the page table size for a size of contiguous memory.
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 a page table and stores the BAT information 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 page table and BAT information. mapmemlen specifies the length, in bytes, of mapmem. If mapmem is zero, mmulib allocates memory for the page table and BAT information from Region 0. In this case, mapmemlen is ignored.

If mapmem is not zero, mapmemlen specifies the page table size plus BAT_ARRAY_SIZE. The page table size is restricted to sizes from 2^{16} to 2^{25}. mapmem must be a multiple of the page table size. If mapmem is too small or Region 0 lacks sufficient space, or mapmem is not on a page table boundary, then a fatal error occurs. An error code is returned, and the page table is not created.

BAT_ARRAY_SIZE	Define statement that gives the size of the BAT information.
MIN_PGTBL_SIZE	Define statement that gives the minimum size of a page table.
MAX_PGTBL_SIZE	Define statement that gives the maximum size of a page table.

mmulib calculates the amount of memory required to hold the page table and BAT information and returns it in the variable pointed to by tablesize. 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 zero 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	Duplicate page is referenced in the map_t array.
EMMU_ADDR_NOT_ON_PAGE	Starting address of a section is not on a page boundary.
EMMU_LEN_NOT_PAGE_MULT	Length of a section is not a multiple of the page size.
EMMU_INV_BAT_SIZE	Length of a BAT is invalid.
EMMU_ADDR_NOT_ON_BAT	Starting address of a section is not on a BAT boundary.
EMMU_EXCEED_MAX_BATS	Number of BATs defined exceeds the max number allowed.
EMMU_PGTABLE_ALIGN	Page table address is not on a page table boundary or the page table size is invalid.

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 zero 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 or block boundary, and len must be a multiple of the page size. For BATs, len is ignored. If an address in the BAT which contains addr is changed then the whole BAT is changed.

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_IBAT, MAP_DBAT, 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 zero 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.
EMMU_ADDR_NOT_ON_BAT	Starting address of a section is not on a BAT boundary.
EMMU_EXCEED_MAX_BATS	Number of BATs defined exceeds the max number allowed.
EMMU_BAT_NOT_DEFINED	BAT in the memory area is not defined.

`map_getattr()` returns the attributes of a given page or block 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 or block. `attr` points to a variable that is used both as an input and an output. The variable must specify whether to get the attributes of a page, IBAT, or DBAT. The `attr` variable must be set to zero for a page, `MAP_IBAT` for an IBAT, or `MAP_DBAT` for a DBAT. The page or BAT attributes are then 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_IBAT`, `MAP_DBAT`, `MAP_INVBIT`, `MAP_WPBIT`, `MAP_CIBIT`, `MAP_CBBIT`, `MAP_SERBIT`, `MAP_SUPBIT`

`map_getattr()` returns zero or the following error code:

EMMU_PAGE_NOT_DEFINED	A page in the memory area is not defined.
EMMU_BAT_NOT_DEFINED	BAT 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 zero or the following error code:

EMMU_TID_NOT_VALID	The task ID is not valid.

Callout Functions

mmulib includes several functions which must be called during system state changes.

NOTE: If you are using the pSOSystem Application Development Environment, calls to these functions are automatically generated at the appropriate points. You need not be concerned with the details of the calls.

Callout mmulib functions are as follows:

map_cocs	Context switch callout procedure
map_reco	pROBE+ entry callout procedure
map_rxco	pROBE+ exit callout procedure
map_start	Task start callout procedure

map_cocs() is a special-purpose procedure 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_cocs (unsigned long NewTID,
   unsigned long *NewTCBptr,
   unsigned long OldTID,
   unsigned long *OldTCBptr
   )
```

The address of map_cocs() can be entered directly into the kc_switchco entry in the pSOS+ Configuration Table.

NewTCBptr is the pointer to the task control block of the task that is gaining the CPU. NewTID 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 the pROBE+ debugger has complete access to all physical memory. map_rxco() restores the MMU to its state prior to the last map_reco() call.

map_reco() and map_rxco() should be called from a function which is put into the pROBE+ Configuration Table entry td_statechng. Code for this is given in probecfg.c in the configs/std directory. The function name is ProbeStateChng.

map_start is a special-purpose procedure that saves the startup state of the cache and the mmu. Then on a restart these saved values will be used to restore the state. The syntax for this function is as follows:

```
void map_start (void);
```

The address of map_start() can be entered directly into the kc_startco entry of the pSOS+ Configuration Table.

NFS Server

Description

NFS Server contained in the Internet Applications product allows systems to share files in a network 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 Internet Applications 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.

In addition, NFS Server requires the following system resources:

- Total of 16 Kbytes of task 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 one (TRUE), NFS Server checks a client's UNIX ID for the value zero (indicating a root client) before mounting. If unix_auth equals zero (FALSE), any client on a trusted machine can mount any of the exported directories.
error_opt	Relates to error response. If error_opt equals one (TRUE), NFS Server returns the appropriate error status on operations that attempt to modify file attributes. If error_opt equals zero (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 two 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 zero, 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 zero.

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 zero; otherwise, it returns a non-zero value on failure. The error values are:

`E_NFSD_CFGPARAM`	Determines that one or more parameters are invalid.
`E_NFSD_START`	Indicates if it is unable to start NFS tasks.

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.

EXAMPLE 1-4: Calling NFS Server

```
#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.

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.

pSH+

Description

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

- Application task(s) (`pshn`), which provide the login shell(s) on the console or other serial ports.
- 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 Internet Applications 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.
- (optional) pHILE+ File System Manager (only if none of the configured shell commands require pHILE+).
- pNA+ TCP/IP Network Manager.
- pREPC+ Run-Time C Library.

In addition, each session of pSH+ requires the following system resources:

- Two Kbytes of dynamic memory is allocated from Region 0 for each shell session and it is freed when the session exits. In addition, some of the pSH commands allocate and free memory from Region 0 during their operation.
- One TCP socket, which is used to listen for Telnet server requests, and two additional TCP sockets per shell session.

The user-supplied pSH+ Configuration Table defines application-specific parameters. The following is a template for this configuration table. This structure should be statically allocated by the application and passed to the library. 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;           /* Pointer 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 */
   unsigned long supv_stack; /* Supv stack size */
   unsigned long user_stack; /* User stack size */
   char *psh banner;         /* Banner to be used */
   };
```

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

`flag`	The interpretation of this field is different from earlier releases of Internet Applications. This flag is a bit mask used to control pSH task modes. If PSH_FPU is set, the pSH task is started with T_FPU mode. If PSH_SUPV bit is set, the pSH task is started in supervisory mode on processors supporting USER/SUPER-VISOR mode.
`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. This should be set to NULL if pHILE+ component is not configured.

`ulist` Points to a list of structures. The structures contain login information about permitted users. If this field is zero, a user callout function is invoked. This function should return zero upon a successful validation of your login, or -1 if the login sequence fails. 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 zeroes 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}
   };
```

The user callout function should be implemented as part of the user application. The prototype declaration for this user callout is as follows:

```
int psh_validate_user(char *login)
```

`login` After a successful login, the login name should be copied to this buffer. This should be a NULL terminated string of up to 20 characters. The callout function should either return zero for a successful login or -1 on failure.

NOTE: Another use for this callout function could be to run any command or application automatically as soon as the user logs in.

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. 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 zeroes 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 built-in commands must be manually added to the cmd table. The cmd entry allows users to add commands to the shell. (The subsection *pSH+ Built-In Commands* on page 1-124 describes built-in commands, and on page 1-120 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 zeroes 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 cmd-data[], for example:

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

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

xxx is any built-in command supported by Internet Applications library.

`console_dev`	Represents the device for this application. The name `console_ dev` is not accurate, but it is retained for compatibility. This device number can be any serial device supporting the Device Independent Terminal Interface (DITI) interface on which the pSH session can be started.
`pseudo_dev`	Stands for the pseudo device, used for I/O redirection. It is set to DEV_PSEUDO, and is defined in `sys_conf.h` The pseudo driver is provided as part of the Internet Applications 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+>".
`supv_stack`	Specifies the supervisor stack size to be used for the pSH task.
`user_stack`	Specifies the user stack size to be used for the pSH task.
`psh banner`	Specifies the banner string to be displayed for pSH. This is a `NULL_TERMINATED` static string pointer. This string is displayed by pSH at startup.

NOTE: On most processors, a task can execute only in supervisor mode. Thus, a task can have only a supervisor task. On these processors, user_stack is added to supv_stack to create a supervisor stack of the combined sizes.

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 zero; otherwise, it returns a non-zero value on failure. The error value can be any pSOS+ error. When this error occurs (a non-zero value) an error message will not be displayed. Check your pshcfg table entries for input errors.

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_PSEUDO,           /* 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");
}
```

The pSH daemon is started by making the pshd_start(pshcfg) system call from the application. This task is responsible for pSH sessions over telnet connections.

The parameter `pshcfg` is a pointer to the configuration table, which is the same as the pSH+ configuration table `pshd_start()`.

The commands and applications can be added to pSH+ so that they can be invoked from the shell. Both commands and applications can be passed parameters, which follow the command or application name. The following is an example:

```
pSH+> ls -1
```

or

```
pSH+> telnet <hostname>
```

For the parameters, a space character is used as token separators. If the parameter contains a space character, the entire parameter needs to be included in double-quotes. A double-quote immediately followed by another double-quote is interpreted as one double-quote character. The following is an example:

```
pSH+> ls -1 "my file"
```

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+.

If any of the user commands need to change its stdio settings (for example, turning off echo), you can do this by invoking de_cntrl() in a transparent manner. The user applications called from either local or remote pSH sessions use the Pseudo device to send IOCTL messages using a uniform interface. If the Pseudo driver detects that IOCTL is coming from a task running on a local serial port, it redirects IOCTL to the appropriate driver. However, if IOCTL comes from a task that runs on a remote pSH session, the Pseudo driver will handle it appropriately.

The following example illustrates how a command can change the stdio settings that can be used on any local or telnet pSH+ ports:

```
#include <diti.h>

change_mode()
{
        unsigned long ioretval;
        TermCtl termctl;
        struct termio tio;
        termctl.arg = (void *)&tio;

    termctl.function = TCGETA;
if (de_cntrl(DEV_PSEUDO, &termctl, &ioretval)) {
    perror("de_cntrl");
}
tio.c_lflag &= ~ECHO;
termctl.function = TCSETAF;
if (de_cntrl(DEV_PSEUDO, &termctl, &ioretval)) {
    perror("de_cntrl");
 }
}
```

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 five parameters: argc, argv, env, exit_param, and console_dev. The first three parameters are defined in the same way for applications as they are for commands (see the preceding subsection). exit_param is a parameter used in the psh_exit function when an application terminates. console_dev parameter is the device on which this application is currently running. The shell sets up the application task with the stdio pointing to this device. If this application starts other tasks, the console_dev parameter can be used by the application to redirect stdio of these tasks.

pSH+ applications inherit both the mode and flags from the pSH+ task. If pSH+ starts in supervisor mode, all the applications started from this shell are started in supervisor mode.

Similarly, if the pSH+ task starts with the FPU flag set, all the applications started from pSH+ inherit this flag. If the user stack size is set to -1, the pSH+ application is started in supervisor mode (irrespective of the mode of pSH+ task starting it).

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. To enable these commands, they must be manually added to the cmd table. 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`*`...]`

`cat` Concatenate and display**.** `cat` sequentially reads each ***file-name*** 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`*`]`

`cmp`	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 zero denotes octal). `cmp` options are as follows:

l	Silent. Print nothing for differing files.
s	Silent. Print nothing for differing files.

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

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

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

`cp` 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:

cp	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.

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.

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.

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. For 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***). For 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. For example:

```
psh> kill tftd
```

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

`ls`

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. `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.

`ls`	`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`	Shows 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. For 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. For 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`	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).

For 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: host_directory directory`*

Mount the remote file system using NFS protocol. The host ***host*** should advertise the directory, ***host_directory*** for this command to complete successfully. The `host` argument can be either an IP address or a hostname if the Name Resolver is configured.

`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. For 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).

For example:

```
psh> pcmount 5.3
```

`pwd`

Display the pathname of the current working directory. For 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***. For 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.

  Examples:

```
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
```

`ping [ -s ] `*`host`*` [`*`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. The `host` argument can be either an IP address or a hostname if the Name Resolver is configured.

When using `ping` for fault isolation, first `ping` the local host to verify that the local network interface is running. For 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. For 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. For example:

```
psh> pwd
5.5/

psh> pushd test_dir
psh> pwd
5.5/test_dir
```

`rmdir` *`directory`*`...`

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

`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.

Examples:

```
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. For 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***). For 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***). For 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. For 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). For 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
```

pSH Loader

Description

The pSH Loader is implemented as part of the user application and is provided in the loader interface source, which is located in `apps/netutils/loader` directory. This feature provides a simple interface to a loader library in pSOSystem.

The loader is implemented as a pSH command, which is invoked from the pSH user interface either on the console or Telnet session. The loader command provides a simple interface, which is similar to other pSH applications like Telnet or FTP.

System/Resource Requirements

pSH Loader requires the following components:

- Loader Library.
- pNA+ TCP/IP Network Manager.
- TFTP Driver.

pSH Loader Commands

The loader application is invoked from a pSH session by the command `loader`. At the `loader>` prompt, you can enter commands to load or unload applications.

The following is a list of the commands that are supported by the pSH loader:

`load` The `load` command is used to load an application from either the RAM disk file, SCSI disk file, or TFTP host file.

The following are examples used for the `load` command:

```
loader> load /r <local file>
loader> load <tftp file>
```

`display` The `display` command displays the loaded application on the system.

The following is an example of the `display` command:

```
loader> display
```

`unload`	The `unload` command is used to unload a loaded application. The following is an example of the `unload` command: loader> unload <*module name*>
`help`	The `help` command displays the supported commands. The following is an example of the `help` command: loader> help
`exit`	The `exit` command returns to the pSH prompt.

RARP

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).

This function returns the IP address of the given network interface. The RARP request is a link-layer broadcast with the following syntax:

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

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

RARP Dialog Example

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.

routed

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 Kbytes of task stack used by each 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 one or zero. One 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 one or zero. One 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 Internet Applications library.

routed is started with routed_start(routedcfg_t *). If routed is started successfully, routed_start() returns zero; otherwise, it returns a non-zero value on failure. The error value can be any pSOS+ error.

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");
}
```

Telnet Client

Description

The Telnet Client contained in the Internet Applications 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 1-150.

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` in the *Telnet Commands* section). 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.

In addition, each Telnet session requires the following system resources:

- 3952 bytes of dynamic memory allocated from Region 0 to store session parameters. This memory is freed when the session exits.
- Three TCP sockets are used for each Telnet session, which are closed when the session exits.
- Stack size is configured by the user. If the user stack size is given as -1 or if pSH+ from which Telnet Client is started is in supervisor mode, the telnet task is started in supervisor mode.

The Telnet Client can be started from any pSH session on any of the serial ports, or from a pSH session over the Telnet login.

pSH+ starts Telnet Client by calling `telnet_main()`. The Internet Applications library 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, 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 five parameters: `argc`, `argv`, `env`, `exit_param`, and `console_dev`. 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.
`int console_dev`	Device number of the device on which this Telnet session is operating on.

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***.

`mode` *`argument value`* Enable the Telnet Client and Server as a line mode option. The supported variables follow:

`char`	Set to character-by-character default mode.
`line`	Set to line-by-line mode.

`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* or a hostname if DNS or Static Name Resolver is configured. For example:

```
telnet> open 192.9.9.9 or
telnet> open MyHostName
```

`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.

`send` *`arguments`*	`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.

`set` *`argument value`*	`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.
`set` *`argument value`*	`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.

`toggle argument...`	`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.
	`localflow`	Toggle local XON/XOFF flow control. If disabled (for example, the default of the operation), the Telnet Client will not handle XON/XOFF characters and forwards them to the server. If enabled, the Telnet Client handles XON/XOFF flow control locally.
	`netdata`	Toggle the display of all network data (in hexadecimal format). The initial value for this toggle is FALSE.
	`?`	Display the legal toggle commands.
	`binary`	Toggle between binary and network virtual terminal (NVT) mode. If set to binary mode, telnet options DO BINARY and WILL BINARY are sent to the server. If the server accepts them, the session is configured in binary mode. The telnet escape character is not accepted. The only way to go back to NVT mode is to disable this option from the server. See the applicable Telnet RFC for an explanation on BINARY option and other telnet commands such as DO and WILL.

Limitations in the Current Version

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

Telnet Server

Description

Telnet Server contained in the Internet Applications 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.
- Eight Kbytes of task 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. This table is statically allocated by the application and passed to the library. The template exists in the include/netutils.h file.

```
struct tnpcfg_t {
   long task_prio;                  /* Priority for tnpd task */
   long max_sessions;              /* Maximum number of concurrent sessions */
   char **hlist;                   /* Ptr to the list of trusted clients */
   int  (*get_shell_port) (void);  /* To contact the shell */
   char *tnpd_banner;              /* Banner to display on telnet session */
   };
```

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 zero, Telnet Server accepts connection from any client.
`get_shell_port`	Specifies the function that is invoked by the Telnet server when it needs to connect to a shell. This function should return the TCP port number. The telnet server tries to connect to this port (over a local TCP connection). If this field is set to NULL, the telnet server uses a default function `psh_get_dport()` and connects to the pSHELL session. This field enables you to provide your own shell for remote login users instead of the default pSHELL.
`tnpd_banner`	Initializes the banner string that is printed when a remote telnet user logins to the telnet server. If this field is set to NULL, the telnet server prints a default banner. This field enables you to provide your own banner for remote login users instead of the default telnet server banner.

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 zero; otherwise, it returns a non-zero value on failure. The error value can be any pSOS+ error.

Configuration Table 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");
```

TFTP Client

Description

Trivial File Transfer Protocol (TFTP) Client in the Internet Applications library provides a simple, command-line user interface to the client side of the TFTP protocol. TFTP Client is contained in pSOSystem's Internet Applications product.

TFTP Client runs as an application under pSH+ and is invoked by the following command:

```
pSH+> tftp[remote system]
```

If no arguments are given, it enters in a command mode.

Configuration and Startup

A TFTP Client requires:

- pSOS+ Real-Time Kernel.
- pNA+ TCP/IP Network Manager.
- pREPC+ Run-Time C Library.
- pSH+ interactive shell.

In addition, each session of TFTP requires the following system resources:

- One UDP socket that is used to send and receive TFTP packets.
- 100 bytes of dynamic memory for each session allocated from Region 0. This is freed when the TFTP session exits. Additionally, TFTP allocates dynamic memory for sending packets, and frees it after the packet is sent over the network.

pSH starts TFTP Client by invoking `tftp_main()`. An entry should be made in the list of pSH+ applications, which is illustrated in the following example:

```
struct appdata_t appdata[] = {
{"tftp", "TFTP protocol", tftp_main, "tf00", 250, 4096, 4096, 1, 0}.
{0, 0, 0, 0, 0}
};
```

If the user stack size is given as -1 or if pSH+ from which TFTP Client is started is in supervisor mode, TFTP Client is started in supervisor mode.

The routine `tftp_main()`, which runs as a separate pSH+ task, allocates memory for the session and initializes the session parameters to the default values. Subsequently, the parameters can be changed using the TFTP command line interface. The session parameters include the timeout value used for TFTP retransmission, the number of times a packet is retransmitted, and so forth.

Additional Options

The TFTP client and server support RFCs 1782 – 1785. These RFCs define the option negotiation in the TFTP protocol, which is described in RFC1782. The application for this negotiation to blocksize, filesize, and timeout parameters are described in RFC1783, RFC1784, and RFC1785. The option negotiation facilitates TFTP Client and TFTP Server to agree on some of the operating parameters, which were earlier hardcoded values. The parameters include blocksize for transfer, timeout value for retransmission, and size of the file to be transferred. See RFCs 1783 – 1785 for more detailed description on these parameters.

TFTP Client Commands

The following TFTP commands are supported by the TFTP Client:

`connect` *`<host-address>[port]`*	Connect to a site. This can be an IP address or a hostname if the Resolver is configured.
`mode` *`<ascii/binary>`*	Set the mode of transfer to ASCII or BINARY.
`put` *`<loc_file> [host:rem-file]`*	Put a file at a remote site. If `rem_file` argument is not present, a remote file is created with the same name as `loc_file`.
`get` *`[host:]<rem-file> [loc-file]`*	Get a file from a remote site. The `loc_file` argument is a regular file or a pSOS+ device (for example, "13.0"). If `loc_file` is not specified, a local file is created with the same name as `rem_file`.
`verbose`	Toggle setting and disabling verbose mode.
`trace`	Toggle setting and disabling packet tracing option.

`remxit` *`<no-retransmits>`*	Set the number of retransmits.
`timeout` *`<timeout>`*	Set the value of timeout for retransmission (in seconds).
`quit`	Quit the tftp session.
`status`	Show status of tftp session.
`option` *`<blksize|timeout|trsize>`*	Enable and disable option negotiation of the parameters.
`blksize` *`<blocksize for transfer>`*	Set the blocksize to negotiate for transfer. If the negotiation succeeds, use this for transfers.
`filesize` *`<max filesize for transfer>`*	Set the maximum size of the file that can be transferred. This is useful only when the client is able to negotiate the `trsize` option with the server.
`help`	Provide help for commands.

TFTP Server

Description

TFTP Server in the Internet Applications 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.

In addition, TFTP server requires the following system resources:

- Sixteen Kbytes of task stack for `TFD$` daemon task.
- 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. This table is statically allocated by the application and passed to the library. 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 one, TFTP Server runs in verbose mode. A zero disables it.
enable_log	Determines the logging code where: 1 = yes 0 = no
reserved	Reserved for future use, and each must be zero.

TFTP Server comes as an object module in the networking utilities library. To use it, sys/libc/netutils.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 zero; otherwise, it returns a non-zero value on failure. E_TFTPD_MALLOC is the error value for memory allocation failure or it can be any pSOS+, pHILE+, or pNA+ error.

Additional Options

The TFTP client and server are enhanced to support RFCs 1782 – 1785. These RFCs define the option negotiation in the TFTP protocol, which is described in RFC1782.

The application for this negotiation to blocksize, filesize, and timeout parameters are described in RFC1783, RFC1784, and RFC1785. The option negotiation facilitates TFTP Client and TFTP Server to agree on some of the operating parameters, which were earlier hardcoded values. The parameters include blocksize for transfer, timeout value for retransmission, and size of the file to be transferred. See RFCs1783 – 1785 for more detailed description on these parameters.

Configuration Table 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 zero indicates a successful shut down. Otherwise, the return value indicates the error status.

NOTE: The TFTP Server is used to transfer files to or from a regular file or pSOS+ devices (for example, "13.0").

2 Interfaces

Introduction

pSOSystem contains a set of standard interfaces that can be used between upper level hardware independent and lower level device dependent drivers. Table 2-1 list the interfaces described in this chapter.

TABLE 2-1 pSOSystem Interfaces Documented in this Chapter

Interface	Description	Page
DISI	Device Independent Serial Interface. An interface between the device dependent and independent parts of a serial driver	2-2
DISIplus	Superset of the DISI specification that provides enhancements to its features such as additional I/O control calls and specifications for the use of HDLC (High-level Data Link Control).	2-31
KI	Kernel Interface. Provides a set of standard services that pSOS+m uses to transmit and receive packets.	2-71
NI	Network Interface. A interface to pNA+.	2-85
SLIP	Serial Line Internet Protocol. Packet framing protocol.	2-106

DISI (Device Independent Serial Interface)

Overview

The Device Independent Serial Interface (DISI) is the interface between the device dependent and 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.

Operation

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 controller that will interface with the upper level hardware independent serial protocols of the pSOSystem. This specification provides the information required on the lower level hardware dependent functions you need to write and the functionality they need.

A template of a lower level serial driver, that you can use as a starting point, is provided. This template contains skeleton functions and some common code that can help you organize the hardware 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 `#define` statements and structures discussed in this specification.

You can also use this specification if you have a new protocol or custom serial needs that you want to add on top of a lower level serial controller driver that conforms to the DISI interface. This specification informs you as to what services are provided to those drivers. Figure 2-1 on page 2-3 illustrates the DISI interface.

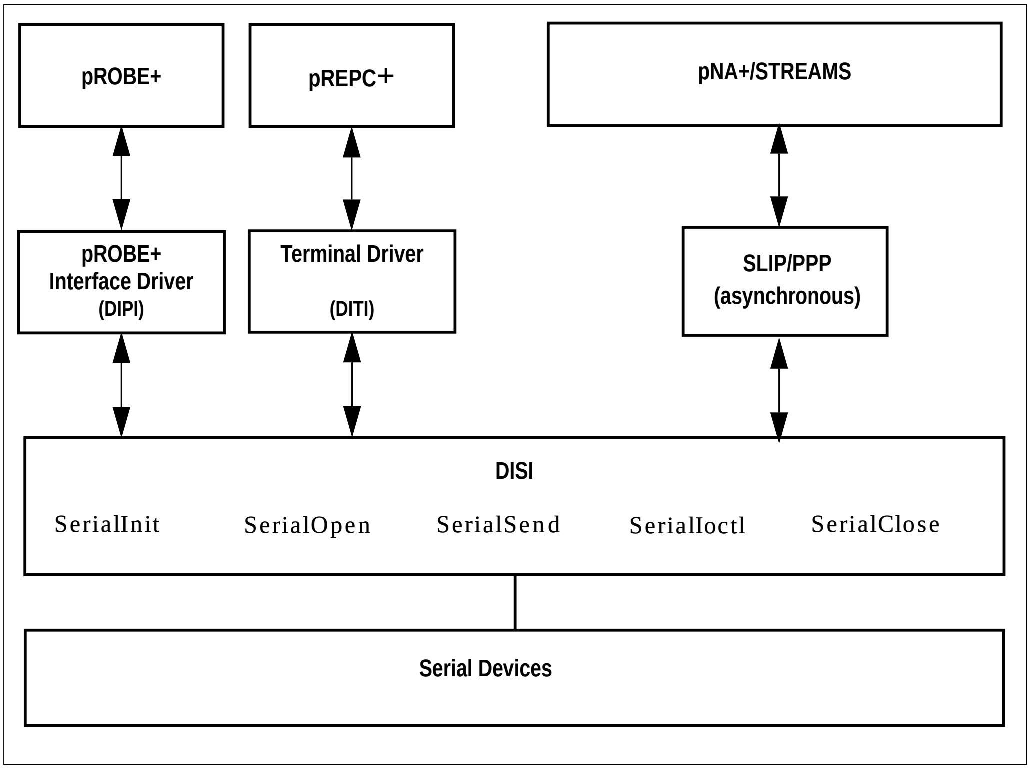


FIGURE 2-1 DISI Interface

The DISI interface consists of the following components:

- Functions that must be provided by the lower level hardware dependent device driver.
- 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 listed in Table 2-2.

TABLE 2-2 Device Dependent Lower Level Serial Code Functions

Function	Description
`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 callback functions that must be supported by the upper level serial driver are listed is Table 2-3.

TABLE 2-3 Upper Level Serial Driver Callback Functions

Callback Function	Description
`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 2-2 illustrates function calls and callback routines in the serial interface:

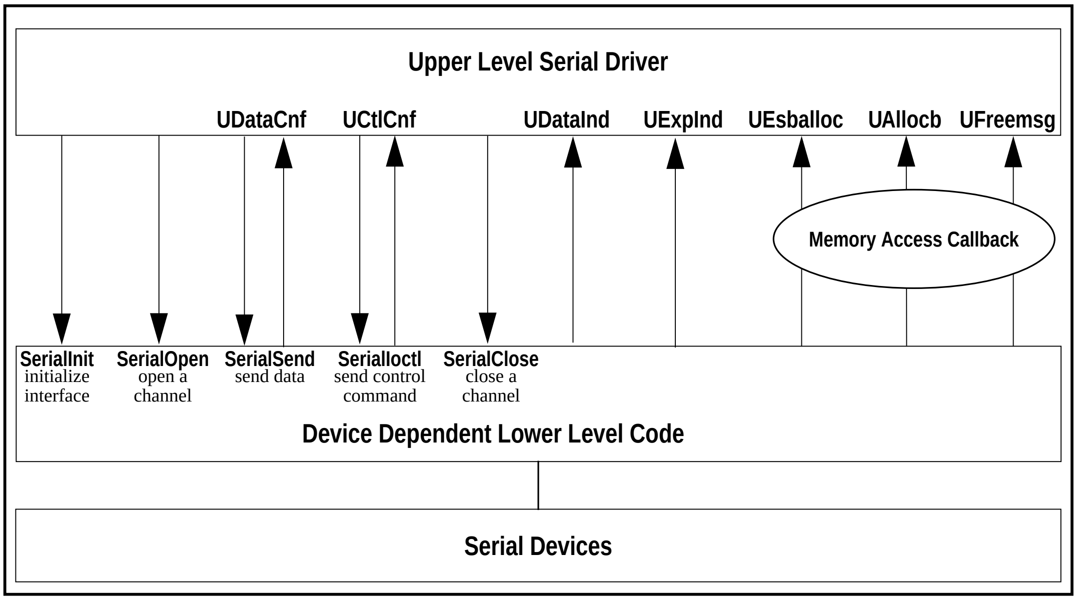


FIGURE 2-2 Function Calls and Callbacks in the Serial Interface

Data is transferred between the upper level drivers and the DISI using the `SerialSend` call to send data out a channel. The `UDataInd` callback function is used by the lower level device dependent part of the driver to inform the upper level driver that data has been received. Data is transferred using the Streams message block structure.

The DISI implements various features such as:

- Asynchronous character mode
- Asynchronous block mode
- Flow control, using 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.

DISI Functions

The following sections describe 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 at boot time before any components are initialized. It sets the driver to a default state with: all channels closed, interrupts off, and all buffer pools empty. It should set the hardware to a known state. Because it is called before pSOS+ is initialized, it cannot use any system calls.

SerialOpen Function

The `SerialOpen` function opens a channel for a particular mode of operation.

```
long SerialOpen(
   unsigned long channel,
   ChannelCfg *cfg,
   Lid *lid,
   unsigned long *hdwflags
   );
```

Input	channel	Indicates the serial channel to be opened.
	cfg	Points to the configuration table that defines various configuration parameters such as baud rate, line parameters, and the addresses of the callback functions. See *Data Structures* on page 2-27 for more details on the configuration table.
Output	lid	Set by the lower level driver and is the reference ID of this channel used by the lower level. All calls to the DISI by the upper layer pass lid except for the SerialInit command.
	hdwflags	Reserved field. Not used for DISI.

Example 2-1 shows the use of a SerialOpen function call used by an upper level serial driver, such as the DITI driver, to open a channel.

EXAMPLE 2-1: SerialOpen Function Call

```
/*****************************************************************/
/* 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);
       }
```

Error Codes

The following error codes can be returned by the `SerialOpen` function:

SIOCAOPEN	Channel already open.
SIOCBADCHANNELNUM	Channel does not exist.
SIOCCFGNOTSUPPORTED	Configuration not supported.
SIOCBADBAUD	Baud rate not supported.
SIOCNOTINIT	Driver not initialized.
SIOBADMINCHAR	MinChar greater than Rbuffsize.

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,
   mblk_t* mbp
   );
```

Input	lid	The lower level ID that was acquired during the `SerialOpen` operation for the channel to which this call is directed.
	mbp	A pointer to the message block that contains the data to be transmitted.
Return	A 0 return code indicates that the message block has been queued to send. The `UDataCnf` callback is used by the lower level driver when the data in the message block has actually been sent.	

Example 2-2 on page 2-11 shows the use of a `SerialSend` call to send data to the lower serial driver.

EXAMPLE 2-2: SerialSend Function Call

```
/*------------------------------------------------------------*/
/* 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);
}
```

Error Codes

The following error codes can be returned:

SIOCNOTOPEN	Channel not open.
SIOCOQFULL	Output queue full, send failed.

NOTE: If a SIOCOQFULL error is received, no data was sent because the transmit queue is full. The SerialSend function continues to return SIOCOQFULL until the next UDataCnf callback happens. Since UDataCnf is the confirmation of a message being sent, the transmit queue is no longer full.

SerialIoctl Function

The SerialIoctl function specifies various control operations that modify the behavior of the DISI.

```
long SerialIoctl(
   Lid lid,
   unsigned long cmd,
   void *arg            input
   )
```

Input	lid	The lower level ID that is acquired during a SerialOpen operation.
	cmd	The type of control operation. (See, Table 2-4)
	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.

Error Codes

The following error codes can be returned:

SIOCCFGNOTSUPPORTED	Configuration not supported.
SIOCNOTOPEN	Channel not open.
SIOCINVALID	Command not valid.
SIOCBADBAUD	Baud rate not supported.
SIOCWAITING	Waiting for previous command to complete.
SIOBADMINCHAR	MinChar greater than Rbuffsize

SerialIoctl Commands

Table 2-4 list the SerialIoctl commands available.

TABLE 2-4 SerialIoctl Commands (cmd)

Command	Description
SIOCPOLL	Polls the serial device for asynchronous events such as data 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.
SIOCBREAKCHK	This command checks 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.

TABLE 2-4 `SerialIoctl` Commands (`cmd`) (Continued)

Command	Description	
`SIOCPROBEEXIT`	This commands tell the driver that 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`	This call queries the lower level driver about which modem controls are supported by the channel. It stores this information into the long int variable 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, control lines 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 variable pointed to by the `arg` parameter. The `SIOCMGET` command uses the same encoding as the `SIOCMQRY` command. 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.	

TABLE 2-4 `SerialIoctl` Commands (`cmd`) (Continued)

Command	**Description**
`SIOCMPUT`	Sets the modem controls of the channel. The `arg` parameter is a pointer to a long int variable 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. The `SIOCMPUT` command uses the same encoding as the `SIOCMQRY` command. 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 takes 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 present in the hardware and stores this information into the long int variable pointed to by the `arg` parameter. This command is immediate, so no call back is made.

Example 2-3 shows the use of a `SerialIoctl` function call to get the baud rate of the channel.

EXAMPLE 2-3: SerialIoctl Function Call

```
/*****************************************************/
/* 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.
Return	If the channel is not open, `SIOCNTOPEN` is returned.	

Example 2-4 shows a SerialClose function call to close the channel.

EXAMPLE 2-4: SerialClose Function Call

```
/*************************************************************/
/* 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.*/
}
```

Error Codes

The following error codes can be returned:

SIOCNOTOPEN	Channel not open.

User Callback Functions

This section describes 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 call of the channel to the device dependent lower level code. These pointers can be changed via the SerialIoctl command SIOPUTA.

NOTE: These user callback functions must be callable from within an interrupt. Consequently, it is important that they do not block within the call and only invoke 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,
   mblk_t * mbp,
   unsigned long b_flags
   );
```

Input	uid	The ID of the upper level serial driver for the associated channel. The ID is passed to the lower lever serial driver during the SerialOpen call 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 without 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, with the SIOCHDWRXPOOL bit set, then the lower level code has a receive buffer pool. This pool needs replenishing through the use of a call to SerialIoctl function with the command, SIOCREPLENISH.

The user supplied functions in the upper level serial driver must use the SerialIoctl function to replenish the buffers. The upper level serial driver must free the message block (pointed to by mbp) when it is emptied by calling the UFreemsg callback function.

Example 2-5 shows a UDataInd function call to send data and status to a task.

EXAMPLE 2-5: UDataInd Function Call

```
/****************************************************************/
/* 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,
   unsigned long exp
   );
```

Input	`uid`	The ID of upper level serial driver for the associated channel which is passed to the lower lever serial driver during the `SerialOpen` function call 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 the SerialSend call has been transmitted.

```
static void UDataCnf(
   Uid uid,
   mblk_t * mbp,
   unsigned long b_flags
   );
```

Input	uid	The ID of the upper level serial driver for the associ-ated channel which is passed to the lower lever serial driver during the SerialOpen function call of the channel on which the data was sent.	
	mbp	Points to the message block sent using the 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	Transmit underrun (HDLC).
		SIOCABORT	Transmit 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 2-6 shows a UDataCnf function call to confirm that data has been sent.

EXAMPLE 2-6: UDataCnf Function Call

```
/****************************************************************/
/* 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,
   unsigned long cmd
   );
```

Input	uid	The ID of the upper level serial driver for the associated channel which is passed to the lower lever serial driver during the SerialOpen function call of the channel on which the I/O control call was made.
	cmd	The command being confirmed.

Example 2-7 shows a UCtlCnf function call to confirm the completion of a SerialIoctl control command.

EXAMPLE 2-7: UCtlCnF Function Call

```
/*******************************************************/
/* 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 (Example 2-10 on page 2-43) under the SerialSend function for an example of this call.

```
static mblk_t * UEsballoc(
   char *bp,
   long len,
   long pri,
   frtn_t *frtn
   );
```

Input	bp	Points to the user 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 controller.

NOTE: This function corresponds to the `gs_esballoc` function supplied by pSOSystem in the file `drivers/gsblk.c` source code file. 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 (Example 2-2 on page 2-11) for an example of this call.

```
static mblk_t * UAllocb(
   long size;
   long pri
   );
```

Input	size	Specifies the size of the buffer.
	pri	Specifies the priority for the 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 the gs_allocb function for the UAllocb callback function.

UFreemsg Callback Function

The UFreemsg callback function is used to free a message block. See Example 2-7 on page 2-24 the term_ctlcnf example under the UctlCnf 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 the gs_freemsg function for the UFreemsg callback function.

Data Structures

Following are templates of data structures. They can be found in `include/disi.h`.

ChannelCfg

```
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. If not set, interrupt mode is used.
`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, the `SerialSend` function 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, the SerialOpen or SerialIoctl functions fail, an error is returned.
`LineMode`	Line mode, which 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.
`alloc`	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. 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.
	PARODD	Odd parity, else even.
	HWFC	Hardware flow control on. When HWFC is set, the channel uses CTS/RTS flow control. If the channel does not support hardware flow control, this bit is ignored.
	SWFC	Software flow control on. When SWFC bit is set, XON/XOFF flow control is enabled.
	SWDCD	Software data carrier detect. 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.
	LECHO	Enable local echo.
	BRKINT	Interrupt on reception of a break character. When BRKINT is set, the channel issues an UExpInd exception callback function if a break character is received.
	DCDINT	Interrupt on loss of DCD. 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.
`ParityErrs`	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 Implementation

See Chapter 5, *Multiplexor Implementations* of *pSOSystem Advanced Topics* for information about multiplexor implementations.

DISIplus (Device Independent Serial Interface)

Overview

DISIplus is a superset of the DISI specification that provides enhancements to its features. In addition to the features provided for by the DISI specification, DISIplus provides several additional I/O control calls and specifications for the use of HDLC (High-level Data Link Control).

Operation

DISIplus is the interface between the device dependent and the device independent parts of a serial driver. The DISIplus interface is used by pSOSystem Terminal, SLIP, PPP (Asynchronous) and pROBE+ upper level drivers to interface with the hardware 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 DISIplus 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 controller that will interface with the upper level hardware independent serial protocols of the pSOSystem. This specification provides the information required on the lower level hardware dependent functions you need to write and the functionality they need.

A template of a lower level serial driver, that you can use as a starting point, is provided. This template contains skeleton functions and some common code that can help you organize the hardware 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 `#define` statements and structures discussed in this specification.

You can also use this specification if you have a new protocol or custom serial interface requirements, that you want to add on top of a lower level serial controller driver that conforms to the DISI interface. This specification informs you as to what services are provided to those drivers. Figure 2-3 on page 2-32 illustrates the interface.

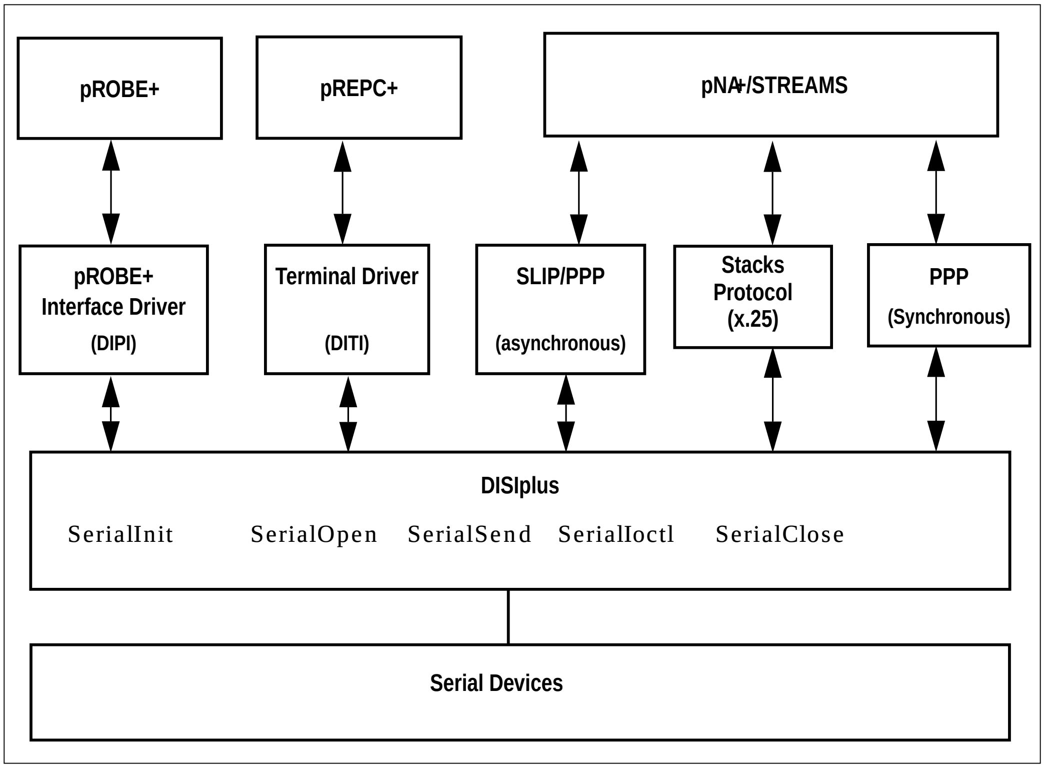


FIGURE 2-3 DISIplus Interface

The DISIplus specification consists of the following components:

- Functions that must be provided by the lower level hardware dependent device driver.
- Callback functions that must be provided by the upper level hardware independent device driver.

Function Calls

The DISIplus 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 listed in Table 2-5.

TABLE 2-5 Device Dependent Lower Level Serial Code Functions

Function	Description
`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 Table 2-6.

TABLE 2-6 Upper Level Serial Driver Callback Functions

Callback Function	Description
`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 2-4 illustrates function calls and callbacks in the serial interface.

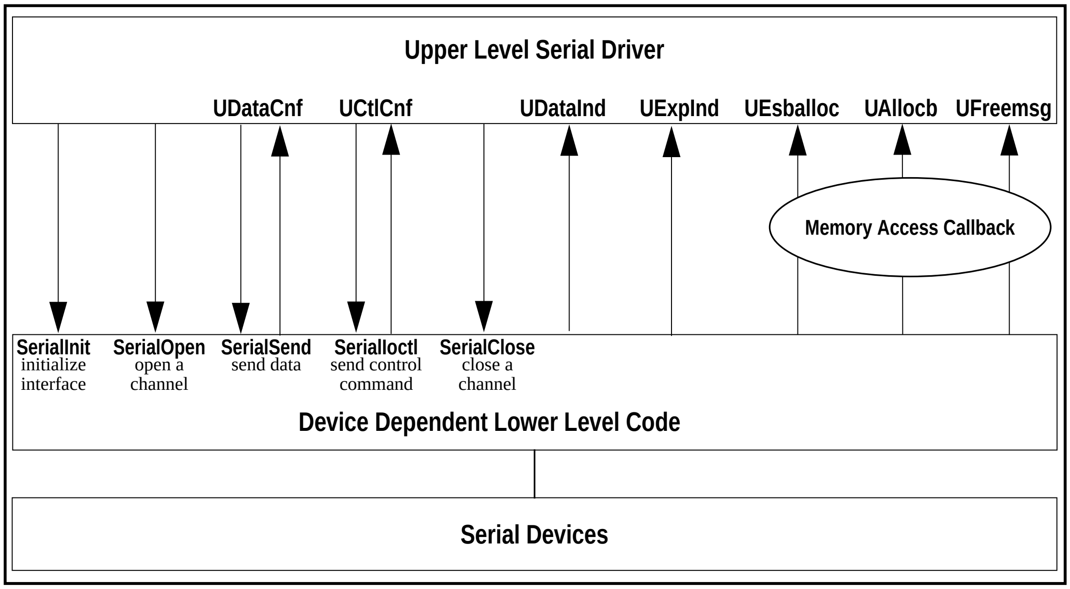


FIGURE 2-4 Function Calls and Callbacks in the Serial Interface

Data is transferred between the upper level drivers and the DISI using the `SerialSend` call to send data out a channel. The `UDataInd` callback function is used by the lower level device dependent part of the driver to inform the upper level driver that data has been received. Data is transferred using the Streams message block structure.

The DISIplus implements various features such as:

- Character mode asynchronous
- Block mode asynchronous and block mode synchronous
- Flow control, using 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.

DISIplus Functions

The following sections describe the functions that must be implemented in the device dependent layer of the DISIplus.

SerialInit Function

The `SerialInit` function initializes the device dependent lower level code.

```
void SerialInit (void);
```

`SerialInit` is called at boot time before any components are initialized. It sets the driver to a default state with: all channels closed, interrupts off, and all buffer pools empty. It should set the hardware to a known state. Because it is called before pSOS+ is initialized, it cannot use any system calls.

SerialOpen Function

The SerialOpen function opens a channel for a particular mode of operation.

```
long SerialOpen(
   unsigned long channel,
   ChannelCfg *cfg,
   Lid *lid,
   unsigned long *hdwflags
   );
```

Input	channel	Indicates the serial channel to be opened.	
	cfg	Points to the configuration table that defines various configuration parameters such as baud rate, line parameters, and the addresses of the callback functions. See Data Structures for more details on the configuration table.	
Output	lid	Set by the lower level driver and is the reference ID of this channel used by the lower level. All calls to the DISI by the upper layer pass lid except for the SerialInit command.	
	hdwflags	Returned by the DISIplus to indicate the capabilities of the lower level serial code. The hdwflags flags can be:	
		SIOCHDWHDL	HDLC supported.
		SIOCHDWRXPOOL	Has receive buffer pool. If SIOCHDWRXPOOL is set, the lower level contains a buffer pool to receive characters and, as they are sent up through the DISIplus, these buffers need to be replenished. (See the SerialIoctl command, SIOCREPLENISH and UDataInd call for more information.)
		SIOCHDMAXTIM	Can do intercharacter timing.
		SIOCAUTOBAUD	Can do autobaud (sync only).

Example 2-8 shows the use of a `SerialOpen` function call used by an upper level serial driver, such as the DITI driver, to open a channel.

EXAMPLE 2-8: SerialOpen Function Call

```
/*******************************************************/
/* 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,
   mblk_t* mbp
   );
```

Input	`lid`	The lower level ID that was acquired during the `SerialOpen` operation for the channel to which this call is directed.
	`mbp`	A pointer to the message block that contains the data to be transmitted.
Return	A 0 return code indicates that the message block has been queued to send. The `UDataCnf` callback is used by the lower level driver when the data in the message block has actually been sent.	

Example 2-9 on page 2-41 shows the use of a `SerialSend` call to send data to the lower serial driver.

EXAMPLE 2-9: SerialSend Function Call

```
/*-------------------------------------------------------------*/
/* 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);
}
```

Example 2-10 on page 2-43 shows the use of the SerialSend function to take a list of data buffers and attaches them to an mblock structure then chains the mblock structures together so they are all part of one HDLC frame.

EXAMPLE 2-10: SerialSend Function Call

```
#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);
```

Error Codes

The following error codes can be returned:

SIOCNOTOPEN	Channel not open.
SIOCOQFULL	Output queue full, send failed.

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 is no longer full.

SerialIoctl Function

The `SerialIoctl` function specifies various control operations that modify the behavior of the DISI.

```
long SerialIoctl(
   Lid lid,
   unsigned long cmd,
   void *arg            input
   )
```

Input	`lid`	The lower level ID that is acquired during a `SerialOpen` operation.
	`cmd`	The type of control operation. (See, Table 2-4)
	`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.

Error Codes

The following error codes can be returned:

`SIOCCFGNOTSUPPORTED`	Configuration not supported.
`SIOCNOTOPEN`	Channel not open.
`SIOCINVALID`	Command not valid.
`SIOCBADBAUD`	Baud rate not supported.
`SIOCWAITING`	Waiting for previous command to complete.
`SIOBADMINCHAR`	MinChar greater than Rbuffsize

SerialIoctl Commands

Table 2-7 lists the `SerialIoctl` commands available.

TABLE 2-7 `SerialIoctl` Commands

Command	Description
SIOCPOLL	Polls the serial device for asynchronous events such as data 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 checks 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 that 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.

TABLE 2-7 `SerialIoctl` Commands (Continued)

Command	Description	
`SIOCBREAK`	Sends a break character out the channel. Any argument passed is ignored. This command is immediate, no callback is made.	
`SIOCMQRY`	This call queries the lower level driver about which modem controls are supported by the channel. It stores this information into the long int variable 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, control lines 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 variable pointed to by the `arg` parameter. The `SIOCMGET` command uses the same encoding as the `SIOCMQRY` command. 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.	

TABLE 2-7 `SerialIoctl` Commands (Continued)

Command	Description
SIOCMPUT	Sets the modem controls of the channel. The arg parameter is a pointer to a long int variable 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. The SIOCMPUT command uses the same encoding as the SIOCMQRY command. 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.
SIOCFLGET	Gets the current state of the flags (defined by the UartCfg structure) and stores this information into an unsigned long int variable 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 variable 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 variable pointed to by the arg parameter. This command is immediate, so no call back is made.
SIOCXFPUT	Sets the new **XOFF** character using the long int variable 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 variable pointed to by the arg parameter. This command is immediate, so no call back is made.
SIOCXNPUT	Sets the new **XON** character using the long int variable pointed to by the arg parameter. The effect is immediate, so no call back is made.

TABLE 2-7 `SerialIoctl` Commands (Continued)

Command	Description
SIOCREPLENISH	Causes the receive buffer pool (if any) to be replenished with new buffers. In some cases, the lower level drivers use a ring of buffers to receive data. As a buffer in the ring is used, it is attached to a mblk structure and sent to the upper level driver by way of a UDataInd call. For more efficient operation and to keep the interrupt latency down, the upper level driver must use the SIOCREPLENISH command so the lower driver replenishes those buffers. The upper level 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 level driver code should keep track of the number of buffers used (one used each time the UDataInd function 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 level driver does not run out of buffers. Of course, the upper level driver can also use the SIOCREPLENISH command every time the UDataInd function is called. Since the UDataInd function is called as part of the interrupt routine, using the SIOCREPLENISH command causes the interrupt to take longer. The SIOCREPLENISH command is necessary only if the hdwflags structure passed in the SerialOpen call had the SIOCHDWRXPOOL bit set. If the SIOCHDWRXPOOL bit is not set, the lower level 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 variable pointed to by the arg parameter. This command is immediate, so no call back is made.

TABLE 2-7 SerialIoctl Commands (Continued)

Command	Description
SIOCSBAUD	Sets the new baud rate for the channel using the information stored in the long int variable 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 variable 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 variable 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; 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.
SIOCRXSTART	Indicates that the upper level serial driver wants to continue to receive characters. The lower level serial code takes 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.

TABLE 2-7 `SerialIoctl` Commands (Continued)

Command	Description
`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.
`SIOCRXFLUSH`	Closes the current receive buffer. This causes the `UDataInd` function to be called for the current `mblock` structure. 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.
`SIOCNUMBER`	Gets the total number of serial channels present in the hardware and stores this information into the long int variable 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 2-11 on page 2-53 shows the use of a `SerialIoctl` function call to get the baud rate of the channel.

EXAMPLE 2-11: SerialIoctl Function Call

```
/*******************************************************/
/* 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.
Return	If the channel is not open, `SIOCNTOPEN` is returned.	

Example 2-12 on page 2-54 shows a `SerialClose` function call to close the channel.

EXAMPLE 2-12: SerialClose Function Call

```
/*****************************************************************/
/* 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 using the SerialIoctl command, SIOPUTA.

NOTE: The user callback functions 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,
   mblk_t * mbp,
   unsigned long b_flags
   );
```

Input	uid	The ID of the upper level serial driver for the associated channel. The ID is passed to the lower lever serial driver during the SerialOpen call 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 without 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 needs replenishing through the use of a call to SerialIoctl with the command, SIOCREPLENISH.

The user supplied functions in the upper level serial driver must use SerialIoctl function to replenish the buffers. The upper level serial driver must free the message block (pointed to by mbp) when it is emptied by calling the UFreemsg function.

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 the UDataInd function is called when the control character is received with the current receive buffer. The last character in the buffer is the character received just before the control character was received.

Example 2-13 shows a `UDataInd` function call to send data and status to a task.

EXAMPLE 2-13: UDataInd Function Call

```
/****************************************************************/
/* 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,
   unsigned long exp
   );
```

Input	uid	The ID of upper level serial driver for the associated channel which is passed to the lower lever serial driver during the SerialOpen function call 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,
   mblk_t * mbp,
   unsigned long b_flags
   );
```

Input	`uid`	The ID of the upper level serial driver for the associated channel which is passed to the lower lever serial driver during the `SerialOpen` function call of the channel on which the data was sent.	
	`mbp`	Points to the message block sent using the `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`	Transmit underrun (HDLC).
		`SIOCABORT`	Transmit 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 2-14 on page 2-59 shows a `UDataCnf` function call to confirm that data has been sent.

EXAMPLE 2-14: UDataCnf Function Call

```
/*************************************************************/
/* 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,
   unsigned long cmd
   );
```

Input	uid	The ID of the upper level serial driver for the associated channel which is passed to the lower lever serial driver during the SerialOpen function call of the channel on which the I/O control call was made.
	cmd	The command being confirmed.

Example 2-15 on page 2-60 shows a UCtlCnf function call to confirm the completion of a SerialIoctl control command.

EXAMPLE 2-15: UCtlCnf Function Call

```
/*********************************************************/
/* 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 (Example 2-10 on page 2-43) under the `SerialSend` function for an example of this call.

```
static mblk_t * UEsballoc(
   char *bp,
   long len,
   long pri,
   frtn_t *frtn
   );
```

Input	`bp`	Points to the user 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 controller.

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 (Example 2-9 on page 2-41) for an example of this call.

```
static mblk_t * UAllocb(
   long size;
   long pri
   );
```

Input	size	Specifies the size of the buffer.
	pri	Specifies the priority for the 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 the `gs_allocb` function 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` (Example 2-14 on page 2-59) 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 the `gs_freemsg` function for the `UFreemsg` callback function.

Data Structures

Following are templates of data structures. They can be found in `include/disi.h`.

ChannelCfg

```
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, asynchronous mode if not set.
SIOCPOLLED	Poll mode, interrupt mode if not set.
SIOCLOOPBACK	Local loop back mode.

The `Mode` parameters decides which structure to use. If `SIOCSYNC` is set the `HdlcCfg` structure is used, otherwise the `UartCfg` structure 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, the `SerialSend` function 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, the SerialOpen or SerialIoctl functions fail, an error is returned.

`LineMode`	Line mode, which 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 (transmit 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_DMANCHESTE
R

Flags	Number of inter-frame 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 causes 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 can be:

CS5	5 bits per character
CS6	6 bits per character
CS7	7 bits per character
CS8	8 bits per character

Flags can be:

CANON	Canonical mode. When the CANON flag 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 the CANON flag is set. If the CANON flag is not set, the delimiters in LineD are ignored and the values of the MinChar and MaxTime arguments are used to determine when to call the UDataInd callback function.
C2STOPB	Send two stop bits, else one.
PARENB	Parity enable. 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.
PARODD	Odd parity, else even.
HWFC	Hardware flow control on. When HWFC is set, the channel uses CTS/RTS flow control. If the channel does not support hardware flow control, this bit is ignored.
SWFC	Software flow control on. When the SWFC bit is set, XON/XOFF flow control is enabled.
SWDCD	Software data carrier detect. 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.

LECHO	Enable local echo.
BRKINT	When BRKINT is set, the channel issues an UExpInd exception callback function if a break character is received.
DCDINT	Interrupt on loss of DCD. When DCDINT is set, the channel issues an UExpInd exception call-back function upon loss of the DCD signal.
AUTOBAUDENB	Enable autobaud. 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 which invokes the UDataInd function.

LFlags A bit field that controls the characters use as follows:

ENDOFTABLE	Invalid (last entry in table). If table has two entries neither entry has 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 the `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 is not placed in an endless loop waiting for 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 is 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. The `UDataInd` function 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. The `UDataInd` function 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 the `UDataInd` function.

`ParityErrs`	Keeps track of the parity errors that occur on the channel. This information is used by MIB.

`FramingErrs`	Keeps track of the framing errors that occur on the channel. This information is used by MIB.
`OverrunErrs`	Keeps track of the overrun errors that occur on the channel. This information is used by MIB.
`Reserved`	Reserved.

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 Implementation

See Chapter 5, *Multiplexor Implementations* of *pSOSystem Advanced Topics* for information about multiplexor implementations.

KI (Kernel Interface)

Overview

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 use shared memory by way of 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 Advanced Topics* manual for more information on the generic shared memory driver.

Operation

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.

The KI of a node must provide the services called by pSOS+m (see Table 2-8).

TABLE 2-8 Required KI Services

Service	Function Code	Description
`KI_INIT`	1	Initialize the KI of a node.
`KI_GETPKB`	2	Get a packet buffer from the KI.
`KI_RETPKB`	3	Return a packet buffer to the KI.
`KI_SEND`	4	Send a packet to another node.
`KI_RECEIVE`	6	Receive an incoming packet.
`KI_SEND`	7	Allow the KI to perform its own timing. For example, to time transmission retries.
`KI_ROSTER`	9	Provide roster information to the KI.

pSOS+m calls the above KI operations as simple subroutines. The KI performs the requested service and simply returns to pSOS+m using a subroutine return. 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:

- Systems with `mc_nnode` > 576 nodes. In such systems, whenever a node joins, the master node requests the 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 is required whenever a node joins.

- Systems that transmit variable length messages larger than 28 bytes. Whenever such a message is sent or requested, pSOS+m requests the 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 requests 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 sends and receives messages of lengths 256 and 512 bytes, then the KI could create three pools of buffers having sizes 100, 328 and 584 bytes. 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 is not allowed to join if its `mc_kimaxbuf` is different from that of the master node. pSOS+ uses `mc_kimaxbuf` in the following two ways:

- 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.
- Any attempt to create a global variable length message queue fails 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 bytes. The KI may allocate a packet buffer of size 256 bytes. Subsequently, pSOS+m calls `ki_send` to send the packet. At this time, pSOS+m will pass a packet size of 80 bytes, not 256 bytes. If the KI has multiple packet buffer pools, then certain KI services, most notably `ki_retpkb`, needs 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 KI of the target node 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:

- The interrupt vectors control to a packet ISR, which should be part of the KI, and the packet ISR receives the packet into a packet buffer.
- The ISR calls the pSOS+m `Announce_Packet` entry (see page 2-75) to inform pSOS+m that one or more packets are pending in the KI.
- The ISR exits using the pSOS+m `i_return` system call.
- At the next dispatch (normally part of `i_return`), pSOS+m calls `ki_receive` to obtain the received packet.
- pSOS+m processes the packet.

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 pSOS+m processes a packet, pSOS+m actually returns to the action of calling ki_receive to obtain queued packets. 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 the first two actions listed above.

NOTE: 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` from supervisor mode. This pSOS+m subroutine neither accepts nor returns any parameters.

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 real-time clock/timer ISR of the node.

Transmission Requirements

pSOS+m assumes that the KI implements the following requisites:

- Reliable Transmission
- Order Preservation
- No Duplication

To achieve compliance with these requisites the KI must adhere to the following rules:

- Rule No.1: *Packets must be delivered correctly to the destination node or an error code must be returned to pSOS+m*. The KI must be responsible for delivery of packets. Failure detection, retransmission (if necessary) and reporting must be done in the KI.
- Rule No. 2: *Between any node pair, packet order must be strictly preserved.* 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. 3: *Packets must be delivered without duplicates*. pSOS+m cannot handle duplicates of the same packet.

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 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 do 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 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.

Table 2-9 describes actions, which are based on three types of node to node communication cases, that pSOS+m follows when `ki_getpkb` or `ki_send` returns an error.

TABLE 2-9 pSOS+m Actions Following `ki_getpkb` or `ki_send` Error Return

Case	Action
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 returns 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 simply shuts down the node. Even in systems that tolerate node failure, these KI calls should never return an error.

KI Conventions and Restrictions

The psos+m kernel calls KI services by way of the KI entry routine specified in the psos+m multiprocessor configuration table. It passes the KI service function code and parameters using the C language calling conventions. The syntax of the KI entry routine is:

```
ULONG ki_call(UNLONG fcode, param1, param2, ..)
```

The KI entry routine executes appropriate KI services depending on the function code, refer to Table 2-8 on page 2-71. A return value of zero indicates successful operation. All other return error codes must be greater than 0x10000. When psos+m invokes the function, `ki_call()`, the processor is in the supervisor state. In the KI services, interrupts may be disabled but must be restored to their original state prior to returning to the kernel.

The KI is logically an extension to pSOS+ kernel. 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 the KI. In general, the 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. The call does not generate a recursive request to the KI (an RSC, for example). This is normally not a problem, since the pSOS+m system calls needed by the KI are unlikely to require remote service.

2. The 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

The `ki_getpkb` service obtains a packet buffer from the KI. The `ki_getpkb` service is called by way of the KI entry routine, `ki_call`, with the following parameters:

```
unsigned long ki_call (
         unsigned long fcode,
         unsigned long dest,
         void **pkt_ptr,
         unsigned long size
);
```

Input	`fcode`	Function code. Must be 2.
	`dest`	Destination node number. 0 means packet will be broadcast.
	`pkt_ptr`	Pointer to the variable that will hold the packet buffer address.
	`size`	Size of the packet requested.
Return	0, or KI-specific error code.	
Output	Packet buffer address is returned in the variable pointed to by `pkt_ptr` if return code is 0.	

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 is sent, 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

The `ki_init` service initializes the KI of a node. The `ki_init` service is called by way of the KI entry routine, `ki_call`, with the following parameters:

```
unsigned long ki_call (
          unsigned long fcode,
         void (*notify_fn)()
);
```

Input	`fcode`	Function code. Must be 1.
	`notify_fn`	Address of pSOS+ `Announce_Packet` Entry.
Return	0, or KI-specific error code.	

The `ki_init` service is called during pSOS+ startup to initialize the KI. It is only called once for each system startup. This `ki_init` service should do the following:

- Initialize the communication hardware.
- Initialize all KI data structures.
- Create a pool of packet buffers. If enough buffers are not created, a system failure can result.
- Save the pSOS+ `Announce_Packet` entry address.

The `ki_init` service is called after all local pSOS+m facilities (including creation of the ROOT and IDLE tasks) have been initialized, and are thus usable. This service is subject to the same restrictions as all other KI services (see *KI Error Conditions* on page 2-76) with the following exceptions:

- It is always called when the interrupts is disabled
- `ki_init` can enable the interrupt, provided that the necessary steps have been taken (for example, setting up ISRs) to handle any possible interrupt sources, and the interrupt is disabled again before returning to the pSOS+m kernel.

ki_receive

ki_receive obtains a received packet. The ki_receive service is called by way of the KI entry routine, ki_call, with the following parameters:

```
unsigned long ki_call (
          unsigned long fcode,
         void **pkt_ptr
);
```

Input	fcode	Function code. Must be 6.
	pkt_ptr	Pointer to the variable that will hold the packet buffer address.
Return	0, or KI-specific error code.	

pSOS+ calls ki_receive only after a call has been made by the KI to the special pSOS+ Announce_Packet entry.

ki_retpkb

The ki_retpkb service returns a packet buffer to the KI. The ki_retpkb service is called by way of the KI entry routine, ki_call, with the following parameters:

```
unsigned long ki_call (
          unsigned long fcode,
          void *pkt
);
```

Input	fcode	Function code. Must be 3.
	pkt	Pointer to packet buffer.
Return	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 can embed the size in the packet buffer envelope.

ki_roster

The ki_roster service provides roster information to the KI. The ki_roster service is called by way of the KI entry routine, ki_call, with the following parameters:

```
unsigned long ki_call (
        unsigned long fcode,
        unsigned long change,
        void **roster,
        unsigned long parm1,
        unsigned long parm2,
        unsigned long parm3
);
```

Input	fcode	Function code. Must be 9.
	change	Specifies the type of change as listed below.
	Value	**Change Description**
	0	Initial roster. The roster parameter points to the internal pSOS+m roster.
	1	A node joined. The parm1 and parm2 parameters contain, respectively, the node number and sequence number of the new node.
	2	A node has failed. The parm1, parm2, and parm3 parameters 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).
Return	0, or KI-specific error code.	

ki_send

The ki_send service sends a packet to another node. The ki_send service is called by way of the KI entry routine, ki_call, with the following parameters:

```
unsigned long ki_call (
         unsigned long fcode,
         unsigned long size,
         unsigned long dest,
         void *pkt
);
```

Input	fcode	Function code. Must be 4.
	size	Packet size in bytes.
	dest	Destination node number.
	pkt	Pointer to packet buffer.
Return	0, or KI-specific error code.	

The ki_send service 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.

The ki_send service 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 can embed the size in the packet buffer envelope.

ki_time

The `ki_time` service allows the KI to implement its own timing and timing dependent operations, if necessary. The `ki_time` service is called by way of the KI entry routine, `ki_call`, with the following parameters:

```
unsigned long ki_call (
         unsigned long fcode
 );
```

Input	`fcode`	Function code. Must be 7.
Return	0, or KI-specific error code.	

The `ki_time` service 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.

NI (Network Interface)

Overview

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 physical media or topology of the network. This interface isolates pNA+ from the physical characteristics of the network.

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 section when describing the Network Interface.

Operation

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.

A network interface to the pSOSystem must include the pNA+ service calls listed in Table 2-10:

TABLE 2-10 Network Interface Service Calls

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

These services are defined by `#define` statements within the `include/ni.h` header file. 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:

- pNA+ Independent Packet Interface (Nonzero-Copy)
- pNA+ Dependent Packet Interface (Zero-Copy).

pNA+ determines which type of packet interface the supporting device driver uses, by checking the `flag` element in the `ni_init` structure of the interface table entry for each driver. Refer to the `include/ni.h` header file 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 the 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 that holds 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. 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+ performs the following sequence of events:

- Uses NI_GETPKB to obtain a packet buffer.
- Stores data in the packet buffer.
- 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 sequence of actions occur on the receiving system:

- The interrupt transfers control to a packet ISR. This ISR is part of the NI, and receives the packet into a packet buffer.
- For each pending packet (several may arrive nearly simultaneously), the packet ISR calls the pNA+ Announce_Packet entry function (see *The pNA+ Announce_Packet Entry* on page 2-89) to transfer the packet to pNA+. pNA+ queues the packet and returns to the ISR.
- After all packets have been transferred to pNA+, the ISR exits using the pSOS+ kernel i_return system call (see *pROBE+ Debug Support* on page 2-93 for one exception to this rule).
- pNA+ processes the packets just received.
- 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 using polling by setting the POLL flag of the NI. If this flag is set, the NI_POLL function 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*.

The dependent 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 by way of 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 must use these routines to allocate and deallocate message block triplets.

A pointer to an interface callback function is passed to the NI during an `NI_INIT` call. 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 link list.

pNA+-Dependent Packet Reception

Upon receipt of a packet, typically by way of an ISR, the driver performs the following sequence of actions:

- The interrupt routine transfers control to a packet ISR. The packet ISR is part of the NI, and receives the packet into a packet buffer.
- The driver attaches the packet buffer to a message block using the `pna_esballoc` service call. The driver then calls the `Announce_Packet` entry function and passes the message block pointer to pNA+. pNA+ queues the

packet and returns to the ISR. The driver repeats this process for each pending packet.

- The ISR then exits using the pSOS `i_return` system call. (If you are using the pROBE+ debugger, there is an exception to this action. See section titled, *pROBE+ Debug Support* on page 2-93 for further information.)
- pNA+ processes the packet and then calls the free buffer routine (passed by way of the `pna_esballoc` function) to free the buffer.

The pNA+ Announce_Packet Entry

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

The C syntax for the `announce_packet` function is:

```
*announce_packet (
   unsigned long   type,
   char           *buff_addr,
   unsigned long   count,
   unsigned long   if_num,
   char          * src_addr,
   char          * dest_addr,
   )
```

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

`Announce_Packet` takes six input parameters. Each parameter is 32 bits long. The parameters are:

`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+ and 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`	Network interface 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` service call.
`src_addr`	Pointer to the source hardware address of the packet.
`dest_addr`	Pointer to the destination hardware address of the packet.

To summarize, upon receiving control by way of the `Announce_Packet` function, pNA+ expects contents in registers r3 through r8 to look like the following:

Registers	Contents
r3	type
r4	buff_addr
r5	count
r6	if_num
r7	src_addr
r8	dest_addr

The `Announce_Packet` entry returns a pSOS+ kernel status flag, which is used by the ISR, if pNA+ is providing communication facilities for pROBE+. (See section *pROBE+ Debug Support* on page 2-93.)

pNA+ Interface Callback

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

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 function is only meant to be used for setting interface parameters and not for retrieving them. For instance, PPP may use this callback function to notify pNA about the new IP address after a negotiation is complete and that the interface is now active.

The `Interface_Callback` function takes four parameters. Each parameter is 32 bits long. The parameters are:

`cmd`	The command 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. Currently, only the `IFF_UP` flag can be set.
`argbuf`	Pointer to the command data. This must be filled with the `ifreq` structure which is defined in the `net/if.h` header file.	
`size`	Size of the `argbuf` parameter. Must be `sizeof(struct ifreq)`.	
`if_num`	Network interface number of the NI making the request. It is the number returned by the `NI_INIT` call.	

Zero-Copy NI Driver

The zero-copy NI driver transfers ownership of the data buffer occupied by the received packet to pNA+. This section describes some design considerations for developing a zero-copy NI driver.

The zero-copy NI driver cannot receive an IP packet if the number of fragments for the packets exceeded the receive buffers configured in NI. The receive buffer is unavailable to the NI driver until pNA+ has finished processing the packet. pNA+ IP reassembly cannot be completed until the last fragment of the IP packet is received. If the fragments for a single IP packet exceed the number of receive buffers in the NI driver, the packet can never be received because NI will exhaust all of its receive buffers before all the fragments of the packets can be received. Even though pNA+ has allocated sufficient memory buffers, the LAN driver cannot utilize those memory blocks.

NOTE: This is not an issue with a nonzero copy NI driver since pNA+ makes a copy of the received buffer and relinquishes its ownership back to the driver.

To address this problem, the zero-copy NI driver can maintain a receive buffer threshold called a *low water mark*. The driver keeps count of the remaining receive buffers and whenever the number of receive buffers reaches a minimum threshold value, the driver issues an indication to pNA+. This is done by logical ORing the `NI_RX_LOW_WATERMARK_REACHED` flag to the `type` parameter to announce the packet. The `NI_LOW_WATERMARK_REACHED` flag uses the upper 16 bits of the `type` parameter (which were not used in versions of pNA+ prior to 4.0).

pNA+ copies the packet into its internal buffer and returns the receive buffers to the driver by calling the free buffers function of the driver.

If the number of receive buffers in the driver goes above the receive low water mark, the driver again indicates to pNA+, in a similar manner, by using the flag `NI_RX_ABOVE_LOW_WATERMARK`.

This return value indicates to pNA+ that the receive threshold in the driver for a specific `IF_NUM` no longer exists, and pNA+ resumes the zero-copy program.

The `NI_RX_LOW_WATERMARK_REACHED` and `NI_RX_ABOVE_LOW_WATERMARK` flags are defined in the `include/ni.h` header file as follows:

```
#define NI_RX_LOW_WATERMARK_REACHED    0x00010000
#define NI_RX_ABOVE_LOW_WATERMARK    0x01000000
```

When pNA+ receives a low water mark indication from the zero-copy NI driver, it temporarily switches to nonzero-copy mode and frees the receive buffer immediately; thus avoiding the conditions described above.

Promiscuous Mode Operation

When communication takes place between end stations that traverse through a gateway, the packet contains the hardware address of the gateway at the MAC or link layer, and the network address of the final destination at the network layer. A host, capable of forwarding packets, generates ICMP redirect packets to the source if it receives a packet which cannot be forwarded directly. This happens if the receiving host determines the received packets must be again forwarded to a gateway, which resides on the same physical network as the source of the received packets. Thus, the route is redirected.

The LAN driver receives all traffic on the LAN while operating in promiscuous mode. Every packet is sent to pNA+ for processing. Without knowing the mode of operation of the LAN driver for certain packets, this can result in packet forwarding by pNA+. In addition, pNA+ generates unnecessary ICMP redirect packets to the sources of those packets. This results in excess traffic being introduced into the network incorrectly and valuable network bandwidth is wasted.

To avoid this, pNA+ queries the LAN driver for its mode of operation at runtime, and the promiscuous mode of operation for the LAN driver is enabled or disabled through pNA+. This ensures that pNA+ correctly performs the packet forwarding function.

pROBE+ Debug Support

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

The first requirement arises because pNA+ (and the NI) may be operating before pSOS+ kernel is initialized. Normally, every ISR should exit by calling the pSOS+ kernel `i_return` system call. Obviously, this is not possible if pSOS+ kernel is not running. Therefore, the NI ISR must be coded so that is only calls the pSOS+ kernel I_ENTER and I_EXIT entries after pSOS has initialized. Before pSOS has initialized, the NI ISR should not call into pSOS at all. This requirement can be met either by installing a new ISR once pSOS has initialized, or by coding the ISR such that it checks a flag to see whether pSOS is initialized. The flag can be cleared by the system initialization code and set by the 'ROOT' task.

The variable `PSOS_FLAG` must be zero prior to pSOS+ initialization, and nonzero 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 the pROBE+ debugger sometimes polls for incoming packets. The NI must provide an additional NI service called `NI_POLL`, which is described in the section *NI Services* below.

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 are explained in this section. 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 defined in the header file `include/ni.h`. The syntax to the entry point of the NI driver is as follows:

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

Following is a description of the arguments used by the `NiLan` function call:

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

Function Code	Description
`NI_INIT`	NI initialization call.
`NI_GETPKB`	NI get 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 I/O control call.

These codes definitions are located in the `include/ni.h` header 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 are explained in the specific section that covers the function code.

NI_BROADCAST

The NI_BROADCAST function is called by pNA+ to transmit a packet to all nodes in the network.

The C structure in the nientry union for the NI_BROADCAST service routine 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;
```

The parameter block for this service is as follows:

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

Following is a field description of the nibrdcast structure:

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.

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

```
args->nibrdcast.count
```

Returns

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

Notes

- NI_BROADCAST is responsible for returning the packet buffer whether or not the packet is successfully broadcast.
- 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, for example) permits, this can be accomplished by a single transmission. Otherwise, the packet must be individually sent to each node.
- 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 C structure in the nientry union for the NI_GETPKB service routine 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;
```

The parameter block for this service is as follows:

pblock + 0	count
+ 4	hwa_ptr
+ 8	if_num

Following is a field description of the `nigetpkb` structure:

`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.

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

```
args->nigetpkb.count
```

Returns

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

Notes

- 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 destination of the packet.

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.

The C structure in the `nientry` union for the `NI_INIT` service routine 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;
```

The parameter block for this service is as follows:

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

Following is a field description of the `niinit` structure:

`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` header file.

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

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

Returns

The NI must return a pointer to the hardware address of the network interface. A return value of -1 signifies that the network interface is not functional.

Notes

- If the interface is not using ARP, the hardware address returned by NI_INIT is not used.
- 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.
- 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 (see Table 2-11) in the parameter block passed to the function.

The C structure in the nientry union for the NI_IOCTL service routine is as follows:

```
struct niioctl
   {
   long cmd;            /* ioctl command */
   long *arg;           /* Pointer to ioctl argument */
   long if_num;         /* NI interface IP address */
   } niioctl;
```

The parameter block for the NI_IOCTL service is as follows:

pblock + 0	command
+ 4	arg
+ 8	if_num

Following is a field description of the `niioctl` structure:

`command`	Operation to be performed by the NI. The operations that can be called by pNA+ are defined in header file `pna.h`. Valid `command` values are listed in Table 2-11.
`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`, which is defined in the C header file `pna_mib.h`.
`if_num`	Network interface number to which the call is made.

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

```
    args->niioctl.if_num
```

The `command` element is specified by a constant. The allowable constants are listed in Table 2-11, and are defined in the include files `pna.h` and `pna_mib.h`.

TABLE 2-11 `NI_IOCTL` Operation Commands

Command	**Operation Description**
Interface Related Operations:	
`SIOCSIFADDR`	Inform the NI of setting of its IP address.
`SIOCSIFDSTADDR`	Inform a Point-to-Point NI of the destination IP address.
`SIOCPSOSINIT`	Inform NI that pSOS is initialized.
Multicasting Related Operations:	
`SIOCADDMCAST`	Add multicast hardware address for packet reception. The `arg` parameter is a pointer to the data type structure `mib_ifreq`, defined in the C header file `pna_mib.h`.
`SIOCDELMCAST`	Delete multicast hardware address for packet reception. The `arg` parameter is a pointer to the data type structure `mib_ifreq`, which is defined in the C header file `pna_mib.h`.

TABLE 2-11 NI_IOCTL Operation Commands (Continued)

Command	Operation Description
SIOCMAPMCAST	Map a protocol multicast address to a hardware address. The arg parameter is a pointer to ni_map_mcast structure which is defined in pna.h. The type field in the structure defines the type of protocol. For the 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.

TABLE 2-11 NI_IOCTL Operation Commands (Continued)

Command	Operation Description
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.

Returns

The NI returns a 0 if successful, or an error value if an error condition exists.

Notes

- MIB-II operations might not be implemented if the application does not require MIB-II support. A call to the NI to retrieve or set the MIB object is made when the application makes an `ioctl()` call on the NI MIB object.
- The operations `SIOCSIFADDR` and `SIOCSIFDSTADDR` are called by pNA+ when an application changes the IP address of the NI or the IP address of the destination (Point-to-Point links) by using the `ioctl()` function call.
- NI can implement a private operation, and the call can be made available to the `ioctl()` call.
- The operation `SIOCPSOSINIT` is called when pNA+ is initialized by pSOS+ kernel. This call is useful when pNA+ is used by the pROBE+ debugger. Since the memory of pNA+ 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 the pROBE+ debugger 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 C structure in the nientry union for the NI_POLL service routine is as follows:

```
struct nipoll
   {
   long if_num;      /* NI interface number */
   } nipoll;
```

The parameter block is as follows:

pblock + 0	if_num

Following is a field description of the nipoll structure:

if_num	Network interface number assigned to this NI by pNA+.

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

```
   args->nipoll.if_num
```

Returns

NI_POLL should always return 0.

Notes

- 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 C structure in the nientry union for the NI_RETPKB service routine is as follows:

```
struct niretpkb
   {
   char *buff_addr;  /* Address of the buffer */
   long if_num       /* NI interface number */
   } niretpkb;
```

The parameter block is as follows:

pblock + 0	buff_addr
+ 4	if_num

Following is a field description of the niretpkb structure:

buff_addr	Address of the packet buffer being returned.
if_num	Network interface number assigned to this NI by pNA+.

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

```
    args->niretpkb.if_num
```

Returns

This service should always return 0.

NI_SEND

NI_SEND is called by pNA+ to send a packet.

The C structure in the nientry union for the NI_SEND service routine 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;
```

The parameter block is as follows:

pblock + 0	`hwa_ptr`
+ 4	`buff_addr`
+ 8	`count`
+ 12	`type`
+ 16	`if_num`

Following is a field description of the nisend structure:

`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.

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

```
args->nisend.if_num
```

Returns

This service returns 0 if the packet is successfully sent. Otherwise, it returns an error code.

Notes

- The `NI_SEND` function 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.

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 provide addressing, packet type identification, error detection or 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. For example: 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 MTU of the peer. 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` file is shown in Example 2-16.

EXAMPLE 2-16: slip_conf.h file

```
/***********************************************************************/
/*                                                                     */
/*   MODULE: slip_conf.h                                               */
/*   PRODUCT: pNA+                                                     */
/*   PURPOSE: User configurations for SLIP driver                      */
/*   DATE: 93/11/15                                                    */
/*                                                                     */
/********************************************************************* */
/*                                                                     */
/*              Copyright 1998, 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_CHANNEL       3          /* which channel to use as SLIP */
#define SLIP_MTU          1006        /* also used for buffer size    */
#define CSLIP              1          /* define to 0 for plain SLIP!  */
#define SLIP_LOCAL_IP   0xc1000002    /* 193.0.0.2                    */
#define SLIP_PEER_IP    0xc1000004
#define SLIPBUFFERS         32        /* Number of slip buffers       */
```

```
/*==================================================================*/
/* End of user configurations                                       */
/*==================================================================*/

#endif /* __SLIP_CONF_H__                                            */
```

3 Standard pSOSystem Block I/O Interface

Introduction

This chapter discusses the standard pSOSystem Block I/O Interface. It provides:

- A general overview of the block I/O interface.
- Detail information regarding the use of pHILE+ block I/O devices. (See page 3-3.)
- The specification for the SCSI device driver; a block I/O type interface. (See page 3-23.)

Overview

The standard pSOSystem Block I/O Interface, defines the interface through which application programs interact with block oriented I/O devices. It also serves as a reference to developers working on writing device drivers for block oriented devices to work under the pSOSystem environment.

The block device driver interface document is an extension to the standard pSOS+ device driver interface which is documented in the *pSOSystem System Concepts* manual. The block device driver interface document merely defines the structure of the I/O Parameter Block (IOPB) passed to the various driver entry points. The following discussion assumes that you are already familiar with the standard pSOS+ device driver interface.

There are two types of block oriented I/O devices: sequential, such as tape; and random access, such as disk. This discussion covers only random access block oriented devices.

Who must understand and follow this specification?

As an application developer, you need to understand this specification if your application needs to directly interact with the device through the de_* services provided by the pSOS+ kernel. A random access block oriented I/O device, such as a disk, is usually not accessed that way. Instead the random access device contains one or more volumes which each contains a file system. The application accesses the file system using pHILE+ by calling standard pREPC+ ANSI C library functions, the C++ I/O streams package, or pHILE+. Therefore it is not necessary to have a detailed understanding of the device driver interface to access a file system. However, you may still need to understand a few details, like how to address the device (such as the device naming convention) and how to use de_cntrl() to initialize the disk (low level physical format of floppy disks or creating primary disk partitions on hard disks). pHILE+ itself, not disk drivers, handles the next step after disk initialization, namely volume initialization (writing a boot record or root block, and creating an empty file system).

Note that all the standard pSOSystem drivers that perform block I/O follow this specification. Many custom drivers for block oriented I/O devices are expected to follow this specification. As a driver developer, pSOSystem does not require you to follow any specific I/O driver interface. However, if you want your driver to work with other pSOSystem software (like pHILE+) that perform block I/O, you must write the driver to follow this specification.

What is a Block Oriented I/O Device?

A block oriented I/O device stores data in blocks. All I/O starts and stops on block boundaries. It is not possible to read or write part of a block. Blocks can be either all the same size, or variable sized. A disk drive has fixed size blocks with 512 bytes, the most common size today. With fixed sized blocks, the I/O size and starting block number, for example BUFFER_HEADER members, b_bcount and b_blockno, respectively, are in units of blocks.

When reading from or writing to a block oriented device the data read or written can be read again. A random access block oriented I/O device requires only de_read(), either directly or from pHILE+.

pHILE+ Devices

This section initially discusses the aspects of pHILE+ devices that are common to all driver entries, as well as introducing the aspects that only apply to particular driver entries. The remaining portions of this section discuss each driver entry and the aspects that apply to that entry.

Except for NFS volumes, the pHILE+ file system manager accesses a volume by calling a device driver via the pSOS+ I/O switch table. When needed, the pHILE+ file system manager calls the driver corresponding to the major and minor device number specified when the volume was mounted.

Device drivers are not initialized by the pHILE+ file system manager. They must be initialized by your application before mounting a volume on the device. Refer to the *pSOSystem System Concepts* manual for more information about initializing of device drivers.

The pHILE+ file system manager uses driver services for the following three purposes:

- I/O operations
- First time initialization of disk partitions and magneto-optical disks
- Media change

Of these three purposes, only I/O operations, is required. Therefore, the corresponding driver entries `de_read()` and `de_write()` are required to be supplied by every driver. The other two purposes are optional.

I/O operations

The `de_read()` and `de_write()` entries are used for I/O operations. They are required in all drivers used with pHILE+.

First time initialization of disk partitions and magneto-optical disks

The `de_cntrl()` driver entry is called with `cmd` set to `DISK_GET_VOLGEOM` to get the geometry of the partition or unpartitioned disk. Without this, `pcinit_vol()` supports re-initialization but not first-time initialization of disk partitions and arbitrary unpartitioned disk geometries since the geometry is not known. (`pcinit_vol()` always supports first time initialization of six built-in floppy disk formats and one built-in magneto-optical disk format.)

Media change

The `de_cntrl()` entry cmd `DISK_SET_REMOVED_CALLBACK` is required for a driver to report a media change to pHILE+. It could be provided by drivers for devices with removable media such as floppy disks and magneto-optical disks. It could also be provided by drivers for removable devices, for example, a PCMCIA SCSI card. Here, the media might not be removable but the whole device is.

A driver can implement driver services not used by the pHILE+ file system manager for additional functions; for example, physical I/O, error sensing, formatting, and so forth. A driver can implement the `de_init()`, `de_open()`, and `de_close()` driver entries even though they are not called by the pHILE+ file system manager. A driver can add additional `cmd` values to the `de_cntrl()` entry.

Before a driver exits, it must store an error code indicating the success or failure of the call in `ioparms.err`. A value of zero indicates the call was successful. Any other value indicates an error condition. In this case, the pHILE+ file system manager aborts the current operation and returns the error code back to the calling application. Error code values are driver defined. Check the error code appendix of *pSOSystem System Calls* for the error code values available to drivers.

Disk Partitions

This section discusses aspects of disk partitions that apply to more than one device driver entry point. Disk partitions are discussed again underneath the driver entry points to present the aspects that apply to only that device driver entry point.

Disks can be either partitioned or unpartitioned. Normally, hard disks are partitioned, and other disks are not. A partitioned disk is divided by a partition table, into one or more partitions each of which contains one volume. An unpartitioned disk is not divided. The entire disk contains one volume. Note, an unpartitioned disk and a partitioned disk with only one partition are not the same.

The partitions on a partitioned disk do not all have to contain the same format volumes. For example, a disk with four partitions could have two partitions containing pHILE+ format volumes, one containing an MS-DOS FAT12 format volume, and one containing an MS-DOS FAT16 format volume, or any other combination.

Our supported disk partitioning is compatible with MS-DOS. The device drivers supplied by Integrated Systems use the same disk partition table format as MS-DOS. This is entirely separate from the file system format used inside a partition. Since the disk table format is the same as MS-DOS, if a partition contains an

MS-DOS format volume, that partition can be accessed by MS-DOS or Windows if the disk is connected to a computer running those operating systems.

The partition number and drive number are encoded within the 16-bit minor device number field. The standard mapping is as follows:

- The upper eight bits are the partition number.
- The first partition is one. Zero is used if the disk is unpartitioned.
- The lower eight bits are the drive number.

Table 3-1 shows the mapping of minor device number to drive number and partition number for drive number zero. All disk drivers supplied by Integrated System, Inc. implement this standard mapping.

TABLE 3-1 Minor Number to Drive/Partition Mapping

Minor Number		Drive	Partition
0	(0x0)	0	Unpartitioned
256	(0x100)	0	1
512	(0x200)	0	2
768	(0x300)	0	3
1024	(0x400)	0	4
1280	(0x500)	0	5
1536	(0x600)	0	6
...			

In custom device drivers, a nonstandard mapping of 16-bit minor number to partition number and drive number is possible. Disk partitions are implemented by disk drivers, not by pHILE+ itself. pHILE+ passes the 32-bit device number including the 16-bit minor device number to the device driver without interpretation. There is only one place internally that pHILE+ divides the 16-bit minor device number into a partition number and a drive number: parsing volume names when they are specified as major.minor.partition. With a non-standard mapping, an application must use major.minor and encode the drive number and partition number into the 16-bit minor number. For example, if custom device driver 3 uses the bottom 12 bits for the drive number and the top 4 bits for the partition number, an application must use 3.0x1002 or 3.4098, not 3.2.1, for driver 3, drive 2, partition 1.

The Buffer Header

A buffer header is used to encapsulate the parameters of a disk read or write. In the `de_read()` and `de_write()` entries of a device driver, the IOPB parameter block pointed to by `ioparms.in_iopb` is a *buffer header*. One parameter of the application I/O error callback is a buffer header to describe the operation that failed.

A buffer header has the following structure, which is defined in pSOSystem `include/phile.h`:

```
typedef struct buffer_header
{
   unsigned long b_device;   /* device major/minor number */
   unsigned long b_blockno;  /* starting block number */
   unsigned short b_flags;   /* block_type: data or control */
   unsigned short b_bcount;  /* number of blocks to transfer */
   void b_devforw;           /* system use only */
   void b_devback;           /* system use only */
   void b_avlflow;           /* system use only */
   void b_avlback;           /* system use only */
   void *b_bufptr;           /* address of data buffer */
   void b_bufwaitf;          /* system use only */
   void b_bufwaitb;          /* system use only */
   void *b_volptr;           /* system use only */
   unsigned short b_blksize; /* size of blocks in base 2 */
   unsigned short b_dsktype; /* type of disk */
} BUFFER_HEADER;
```

A driver uses only six of the parameters in the buffer header. They are the following:

`b_blockno`	Specifies the starting block number to read or write.
`b_bcount`	Specifies the number of consecutive blocks to read or write.
`b_bufptr`	Supplies the address of a data area; it is either the address of a pHILE+ cache buffer or a user data area. During a read operation, data is transferred from the device to this data area. Data flows in the opposite direction during a write operation.
`b_flags`	Contains a number of flags, most of which are for system use only. However, the low order two bits of this field indicate the block type, as follows:

Bit 1	**Bit 0**	**Explanation**
0	0	Unknown block type
0	1	Data block
1	0	Control block

b_flags is used by more sophisticated drivers that take special action when control blocks are read or written. Most drivers will ignore b_flags.

b_flags low bits = 00 (unknown type) can occur only when read_vol() or write_vol() is issued on a volume that was initialized with intermixed control and data blocks. In this case, the pHILE+ file system manager will be unable to determine the block type. If read_vol() or write_vol() is used to transfer a group of blocks that cross a control block/data block boundary, these bits will indicate the type of the first block.

b_blksize	Specifies the size (in base 2) of blocks to read or write.
b_dsktype	Specifies the type of MS-DOS disk involved. It is set by the dktype parameter of pcinit_vol() and is only valid when pHILE+ calls the driver as a result of a call to pcinit_vol(). During all other system calls, this value is undefined. pcinit_vol() is described in the *pSOSystem System Concepts* and the *pSOSystem System Calls* manuals.

The remaining fields are for system use only.

The contents of the buffer header should not be modified by a driver. It is strictly a read-only data structure.

Driver Initialization Entry

This section discusses initialization requirements unique to disk drivers. This includes media change, disk partitions, logical disk geometry, and write-protect.

Media Change

The only media change initialization required is to zero the stored address of the pHILE+ device removed callback routine. This allows the driver to work with older pHILE+ versions that do not support media change.

Disk Partitions and Logical Disk Geometry

Initialization of disk partitions and logical disk geometry are interrelated. Disk partition initialization reads the on-disk partition tables to initialize the driver's in memory partition table. This table contains for each partition the sys_ind field of the partition table, the start as a disk logical block address, and the size. Both the start and the size are in units of 512-byte blocks.

Logical disk geometry initialization calculates both sectors per track and number of heads and stores them for use by the `de_cntrl()` function `DISK_GET_VOLGEOM: Get volume geometry`. These two initializations are done differently by each type of disk driver. The calculations are briefly explained in the subsections that follow for non-removable media. The explanation for removable media is in *DISK_GET_VOLGEOM: Get Volume Geometry* on page 3-16 since it is not done at driver initialization time. For full details see the pSOSystem disk device drivers.

Your driver should recognize partition 0 as a partition spanning the entire disk; that is, your driver should not perform partition table translation on accesses in partition 0.

Assuming your driver follows these guidelines, prepare and make use of DOS hard drives in the pHILE+ environment as described in the *pSOSystem System Concepts* manual.

Partitioned SCSI Disk

Disk partitions (Done first)—Both the start and the size are computed from the `start_rsect` and `nsects` partition table fields. The `CHS` fields cannot be used since the logical disk geometry is not known. The `start_rsect` field of a primary partition and of the first extended partition is absolute. The `start_rsect` field of all other extended partitions is relative to the first extended partition. The `start_rsect` field of a logical partition is relative to the containing extended partition. See the pSOSystem SCSI driver for more specifics.

The disk drivers supplied with pSOSystem support the following partitioning scheme. The driver reads logical sector 0 (512 bytes) of the disk and checks for a master boot record signature in bytes 510 and 511. The signature expected is 0x55 in byte 510 and 0xAA in byte 511. If the signature is correct, the driver assumes the record is a master boot record and stores the partition information contained in the record in a static table. This table is called the driver's *partition table*.

The driver's partition table contains entries for each partition found on the disk drive. Each entry contains the beginning logical block address of the partition, the size of the partition, and a write-protect flag byte. The driver uses the beginning block address to offset all reads and writes to the partition. It uses the size of the partition to ensure the block to be read or written is in the range of the partition.

If the driver finds a master boot record, it expects the disk's partition table to start at byte 446. The driver expects the disk's partition table to have four entries, each with the following structure:

```
struct ide_part {
  unsigned char boot_ind;     /* Boot indication, 80h=active */
  unsigned char start_head;   /* Starting head number */
  unsigned char start_sect;   /* Starting sector and cyl (hi)*/
  unsigned char start_cyl;    /* Starting cylinder (low) */
  unsigned char sys_ind;      /* System Indicator */
  unsigned char end_head;     /* Ending head */
  unsigned char end_sect;     /* Ending sector and cyl (high) */
  unsigned char end_cyl;      /* Ending cylinder (low) */
  unsigned long start_rsect;  /* Starting relative sector */
  unsigned long nsects;       /* # of sectors in partition */
   };
```

The driver computes the starting relative sector and size of each partition table entry. If the driver is an IDE driver, it computes these values from the cylinder, head, and sector fields (`start_head` through `end_cyl`). If the driver is a SCSI driver, it computes these values from the starting relative sector (`start_rsect`) and number of sector (`nsects`) fields.

The driver checks the system indicator (`sys_ind`) element of the first entry. If the system indicator is 0, the driver considers the entry to be empty and goes on to the next entry. If the system indicator is 0x05, the driver considers the entry to be an extended partition entry that contains information on an extended partition table. If the system indicator is any other value, the driver considers the entry to be a valid entry that contains information on a partition on the disk. The driver then stores the computed starting relative sector and the computed size of the partition in the driver's partition table. (The driver never uses cylinder/head/sector information.)

If an extended partition entry is found, the starting relative sector (`start_rsect`) is read as an extended boot record and checked the same way the master boot record is checked. Each extended boot record can have an extended partition entry. Thus, the driver may contain a chain of boot records. While there is no limit to the number of partitions this chain of boot records can contain, there is a limit to the number of partitions the driver will store for its use in its partition table. This limit is set to a default value of eight. This value may be changed by editing the `SCSI_MAX_PART` define statement found in the `include/drv_intf.h` file in pSOSystem, and compiling the board support package you are using for your application. `SCSI_MAX_PART` can be any integer between 1 and 256, inclusive.

NOTE: Once an extended partition entry is found, no other entries in the current Boot Record are used. In other words, an extended partition entry marks the end of the current disk partition table.

Refer to, *SCSI (Small Computer System Interface) Driver* on page 3-23 for more information on the SCSI driver interface.

Logical disk geometry (Done second)—Computed by solving the equations from equating the logical block address corresponding to the partition table CHS and relative sector fields of the start and end of the first partition. See the pSOSystem SCSI driver for more information.

Unpartitioned SCSI Disk

Disk partitions—Not applicable. Mark the disk as unpartitioned.

Logical disk geometry—If you need to interchange this disk with another computer, you should pick values using the same algorithm as the SCSI disk partition software of that other computer. That allows a pSOSystem partition application to call this `de_cntrl()` function and partition your SCSI disk such that it is interchangeable with that other computer. The Integrated Systems' SCSI disk driver uses an algorithm compatible with Adaptec SCSI adapters. They are the SCSI adapters supported by pSOSystem/x86. If you don't need interchangeability with another computer or never partition a SCSI disk with pSOSystem, pick any legal values as the RAM disk does.

Partitioned IDE Disk

There are three methods. The first two are preferred. The pSOSystem IDE disk driver supports both of the first two.

Logical disk geometry

- (Done first) Query the IDE drive for the physical geometry. Translate that to logical geometry. See the pSOSystem IDE driver.
- (Done first) For x86, obtain the logical geometry from CMOS. See the pSOSystem IDE disk driver.
- (Done second) Compute it the same as the SCSI driver does partitioned disks.

Disk partitions

- (Done second) This is used with the first two logical disk geometry methods. Compute the starting and ending blocks from the CHS partition table fields using the logical disk geometry.
- (Done first) This is used with the third logical disk geometry method. Compute it the same as the SCSI driver does partitioned disks.

Unpartitioned IDE Disk

Disk partitions—Not applicable. Mark the disk as unpartitioned.

Logical disk geometry— There are three methods. The first two are preferred. The pSOSystem IDE disk driver supports both of the first two.

- Query the IDE drive for the physical geometry. Translate that to logical geometry. See Integrated Systems' IDE driver.
- For x86, obtain the logical geometry from CMOS. See Integrated Systems' disk driver.
- Compute it the same as the SCSI driver does unpartitioned disks.

de_read() and de_write() Entries

The `de_read()` and `de_write()` driver entries are required in every pHILE+ device driver. They are used for I/O operations.

In the `de_read()` and `de_write()` entries of a pHILE+ device driver, the `IOPB` parameter block pointed to by `ioparms.in_iopb` is a buffer header. (See *The Buffer Header* on page 3-6).

If you want interchangeability of MS-DOS FAT format file systems with an MS-DOS or Windows computer use only devices with a 512-byte sector size. Although the pHILE+ file system manager allows you to initialize an MS-DOS partition file system on devices with other sector sizes, if you connect such devices to an MS-DOS or Windows system, it will not be able to read them.

You can set the write-protect byte through an I/O control call to the driver. The driver checks this byte whenever a write is attempted on the partition. If the write-protect byte is set, it does not perform the write and returns an error to indicate the partition is write-protected.

I/O Transaction Sequencing

pHILE+ drivers must execute transaction (i.e. read and write) requests that refer to common physical blocks in the order in which they are received. For example, if a request to write blocks 3-7 comes before a request to read blocks 7-10, then, because both requests involve block 7, the first request must be executed first.

If a pSOS+ semaphore is used to control access to a driver, then that semaphore must be created with FIFO queuing of tasks. Otherwise, requests posted to the driver might not be processed in the order in which they arrive.

Logical-to-Physical Block Translation

The b_blockno, b_count, and b_blksize parameters together specify a sequence of logical blocks of a volume that must be read or written by the driver. Up to four translations are required to convert this to the physical block addresses on the device. These translations are listed below in the order that they are applied.

1. Block size scaling

   This translation converts from a sequence of arbitrary-sized logical blocks of a volume to a sequence of logical blocks of a volume that are sized to match the physical block size of the volume.

   This translation is nearly never needed with MS-DOS format since most disks today have a 512 byte physical block size, which is the same as the logical block size of MS-DOS format. However, pHILE+ format can have logical block sizes of any power of 2 from 2^8 = 256 bytes to 2^{15} = 32K bytes. Therefore, this translation is required for pHILE+ format.

   The physical block size of the disk drive can be the same or smaller than the logical block size of the read or write request. Therefore, b_blockno and b_count must be scaled.

   This translation is implemented as follows. If the physical block size is greater than the logical block size, the driver returns an error without any disk access, e.g. SCSI returns ESODDBLOCK in pSOSystem include/drv_intf.h. Otherwise, the driver multiplies both b_blockno and b_count by the quotient of logical block size, i.e. 1 << b_blksize, divided by the physical block size.

2. Partition translation

   This translation converts from a sequence of standard-sized logical blocks of a volume to a sequence of standard-sized logical blocks of a disk.

   If the partition number is 0, the access is to an unpartitioned disk. No translation is made. b_blockno and b_count are compared to the total disk capacity to detect accessing beyond the end of the disk.

   If the partition number is nonzero, the access is to a partition. The first partition is one. The driver looks up in a table the partition's starting block number and number of logical blocks. b_blockno and b_count are compared against the partition size to detect accessing beyond the end of the partition. Then, the partition's starting logical block number is added to b_blockno.

3. Logical block number to Logical CHS (Only IDE disks)

   This translation converts from a sequence of logical block numbers on a disk to a sequence of logical cylinder/head/sector parameters on a disk. The logical CHS parameters match the geometry used in the CHS partition table entries.

   This translation uses the following equations:

   Cylinder = block/(sectors-per-track * heads)
   Head = (block/sectors-per-track) MOD heads
   Sector = (block - ((block/sectors-per-track)*sectors-per-track)) + 1

   Block starts at 0. Cylinder is 0 to 1023. Head is 0 to 254 or 255. Sector is 1 to 63. This gives a maximum partitioned IDE disk capacity of 1024 cylinders * 255 heads * 63 sectors-per track * 512 bytes per sector which is between 7.8 and 7.9 gigabytes.

4. Logical CHS to Physical CHS (Only large IDE disks)

   This translation converts the cylinder/head/sector parameters from the logical geometry used by the partition table entries to the physical geometry used at the IDE hardware interface to the disk. These are not the same for IDE disks with over 1,024 cylinders.

   This translation is a result of differing field widths for cylinder/head/sector information in the partition table entries and the IDE hardware interface. See Table 3-2 on page 3-19 for specifics. If field sizes are limited to the smaller of the two, no translation from partition table geometry to IDE geometry is needed. Above this geometry, i.e. IDE disks with over 1,024 cylinders or over 63 sectors-per-track, translation is required.

   This translation creates another problem. The translation was never standardized and can vary from one BIOS to the next. Thus, an IDE disk that requires translation might not be accessible by a computer other than the one that partitioned it, or even by the same computer if the computer's motherboard is replaced or the BIOS is upgraded. This is a problem for interchanging a disk between two MS-DOS computers, and also between an MS-DOS computer and a pSOSystem computer. Fortunately, one of the translation methods seems to be much more common than the others; that is, the one implemented in the pSOSystem IDE drivers. Therefore, most of the time IDE disks can be interchanged between pSOSystem and MS-DOS.

	Cylinder (bits)	**Head (bits)**	**Sector (bits)**
Partition table	10	8	6
IDE interface	16	4	8
No translation	10	4	6

The implementation of this translation is not explained here. Refer to the pSOSystem IDE device driver.

Media Change

If a pHILE+ device driver supports removable media it can notify pHILE+ whenever media is removed by calling the disk removed callback routine. The address of this callback routine is provided to the `de_cntrl()` device driver entry cmd `DISK_SET_REMOVED_CALLBACK`. It stores the disk removed callback routine's address in a local variable where it is accessible to the `de_read()` and `de_write()` device driver entries. Only one scalar variable is needed since the same callback routine is called for every drive and partition. The driver should not call the callback routine if the stored address is zero. This allows using the driver with older pHILE+ versions that do not support media change.

The device removed callback routine has the following C interface type:

```
unsigned long (*)(unsigned long dn, unsigned long bitmask);
```

The `bitmask` parameter is used to mark removed multiple volumes in one call; that is, all partitions on one disk, or all disks on one SCSI adapter. The zero bits are ignored when checking whether to mark a mounted volume. Therefore, the extreme values of 0 and ~0UL (zero unsigned long, or -1) would mark all mounted volumes, or only one volume, respectively. The SCSI driver supplied by Integrated Systems, Inc. divides the 16-bit minor device number as follows:

3 bits	logical unit number
5 bits	partition number
3 bits	adapter number
5 bits	target ID

Therefore, the bitmask value used should be 0xffffe0ff to ignore only the partition number. The same bitmask value can be used for other Integrated Systems, Inc. disk drivers.

If a custom disk driver implements a different mapping for the minor device number, it would use a different bitmask value that corresponds to its minor device number mapping. It can call the callback multiple times if no bitmask value marks all of the required volumes in a single call.

Two types of devices are supported:

- Type 1—Devices that generate an interrupt when media is removed or the door securing media is opened, as in the case of a floppy disk drive.
- Type 2—Devices that do not report media has been removed until the device is accessed after the removal, for example: a SCSI driver.

The `de_read()` and `de_write()` entries of a type 2 device must be programmed as follows. First, they determine that the media was removed without successfully performing the read or write. This could happen two ways. Either they poll the device before I/O to determine whether the media was changed, or after a media change the device returns an error code instead of performing a requested read or write. Second, they call the device removed callback routine to report the media removal to pHILE+. Third, they return an error code to pHILE+. The value doesn't matter. The error code will be ignored since the media removal was already reported.

The `de_read()` and `de_write()` entries of a type 1 device can be programmed the same as a type 2 device. In this case, either the device's media-removed interrupt is disabled, or the media removed interrupt merely sets a flag in the disk driver which is checked during the poll for media removal above.

Alternately, the `de_read()` and `de_write()` entries of a type 2 device can be programmed to report media removals to pHILE+ within the media-removed interrupt handler. The media-removed interrupt handler calls the pHILE+ disk removed callback. The `de_read()` and `dw_write()` entries do not call the disk removed callback.

de_cntrl() Entry

The `de_cntrl()` driver entry of a pHILE+ device driver is optional. It is used by first time initialization of disk partitions and magneto-optical disks, which is done by `pcinit_vol()` with `dktype DK_DRIVER`. It is also used to set the disk removed callback, which the disk driver can use to notify pHILE+ that a disk has been removed. If not supported, these features are not available.

In the de_cntrl() entry of a pHILE+ device driver, the IOPB parameter block pointed to by ioparms.in_iopb is a structure whose first field is an unsigned long that contains a function code. The remaining fields in the structure, if any, depend on the value of the function code. pHILE+ reserves de_cntrl() function codes 100 through 199. Other function code values can be used to provide additional de_cntrl() features not used by pHILE+ itself. struct disk_ctl_iopb is modified to provide a definition of the IOPB parameter blocks used by all de_cntrl() function codes common to more than one disk driver by adding additional fields to the enclosed union u. Symbols for the pHILE+ reserved function code values and struct disk_ctl_iopb are defined in pSOSystem include/phile.h.

DISK_GET_VOLGEOM: Get Volume Geometry

The de_cntrl() function code DISK_GET_VOLGEOM is used to obtain the volume geometry. pHILE+ pcinit_vol() with dktype DK_DRIVER calls this to obtain the volume geometry and file system parameters needed to compute the MS-DOS boot record of either an unpartitioned disk, such as partition number 0, or a single-disk partition. The IOPB parameter block is struct disk_ctl_iopb with union field vol_geom. The structure definition is listed below. This function returns results in struct vol_geom.

Optional fields are either calculated or defaulted if they are zero. Zero will always work. The only time a nonzero value would be used is if it is necessary to exactly match a standard floppy or magneto-optical disk format and the calculated or default value is different.

```
typedef struct
{
unsigned long begin;     /* Partition: Beginning logical block
                          * address or Unpartitioned: 0 */
unsigned long nsects;    /* Number of sectors in partition/disk */
unsigned long secpfat;   /* (Optional) Number of sectors per FAT */
unsigned short nrsec;    /* (Optional) Number of reserved sectors
                          * Default 1 */
unsigned short nrdent;   /* (Optional) Number of root directory
                          *  entries
                          * Partition: Usually 512
                          * Unpartitioned: Varies
                          * Default 512
                          */
unsigned short secptrk;  /* Number of sectors per track (Maximum
                          * 64) */
unsigned short nheads;   /* Number of heads (Maximum 255 or 256) */
unsigned char sys_ind;   /* Partition: System indicator
                          * Unpartitioned: 0
                          */
unsigned char media;     /* Boot record's media descriptor */
unsigned char secpcls;   /* (Optional) Number of sectors per cluster
                          * This must be a power of 2.
                          * pHILE+ supports 1 to 64.
                          */
unsigned char nfats;     /* (Optional) Number of FATs
                          * pHILE+ supports 1 or 2.
                          * Default 2
                          */
unsigned char pad0[16];  /* Reserved. Should be 0. */
} DISK_VOLUME_GEOMETRY;

/* de_cntrl() iopb */
struct disk_ctl_iopb
{
unsigned long function;  /* Function code - values defined below */
union
 {
 DISK_VOLUME_GEOMETRY vol_geom; /* For DISK_GET_VOLGEOM */
 void * removed_call_back;  /* DISK_SET_REMOVED_CALLBACK */
 } u;
};
```

The fields are grouped below. All fields are in terms of a sector size of 512 bytes.

Size

This describes either the whole unpartitioned disk, if partition number 0, or else a single disk partition.

`nsects`	Number of sectors on the disk or in the partition.

Logical Geometry of the Disk

These fields are the logical geometry used by CHS partition table fields and recorded in the boot record:

`secptrk`	Sectors-per-track
`heads`	Number of heads, i.e. number of surfaces per cylinder.

Partition parameters

`begin`	Partition beginning logical block address, or 0 if unpartitioned.
`table_size`	Size of partition table immediately preceding this partition. If unpartitioned, this is zero. If the first primary partition, this is the same as begin. If the first, and usually only, logical partition within an extended partition, this is the beginning of the immediately enclosing extended partition. Otherwise, this is zero.
`sys_ind`	System indicator field of the partition table entry, or 0 if unpartitioned. (See pSOSystem `include/diskpart.h` for the definition of a partition table entry.)

File System Parameters:

`media`	Boot record's media descriptor.
`nrdent`	(Optional) Number of root directory entries.
`nrsec`	(Optional) Number of reserved sectors.
`nfats`	(Optional) Number of FATs.
`secpfat`	(Optional) Sectors per FAT.
`secpcls`	(Optional) Number of sectors per cluster.

Standard floppy formats can be obtained by specifying all the optional values. The geometry and file system parameters of several standard floppy disk formats are given in Table 3-1 on page 3-5. If this format is built-in `pcinit_vol()` the corresponding `dktype` is given.

TABLE 3-2 Standard Floppy Disk Formats

Format	dktype	size	secptrk	heads	media	nrdent	nrsec	nfats	secpfat	secpcls
DD360 5.25" DD 360K	DK_360	720	9	2	0xfd	112	1	2	2	2
DH120 5.25" DH 1.2M	DK_12	2400	15	2	0xf9	224	1	2	7	1
DD720 3.5" DD 720K	DK_720	1440	9	2	0xf9	112	1	2	3	2
DH144 3.5" DH 1.44M	DK_144	2880	18	2	0xf0	224	1	2	9	1
DQ288 3.5" DQ 2.88M	DK_288	5760	36	2	0xf0	240	1	2	9	2
NEC120 5.25" NEC 1.2M	DK_NEC	2400	15	2	0x98	240	1	2	9	2
M2511A Fuji M2511A 124M Optical	DK_OPT	244824	25	1	0xf8	512	1	2	31	32
SS160 5.25" DD Single sided 160K	N/A	320	8	1	0xfe	64	1	2	1	1
SS180 5.25" DD Single sided 180K	N/A	360	9	1	0xfc	64	1	2	2	1
DD320 5.25" DD 320K	N/A	640	8	2	0xff	112	1	2	1	2
SH320 5.25" DD Single sided 320K	N/A	640	8	1	0xfa	112	1	2	1	2
DH360 5.25" DH Single sided 360K	N/A	720	9	1	0xfc	112	1	2	2	2
DH640 3.5" DH 640K	N/A	1280	8	2	0xfb	112	1	2	2	2
DH720 3.5" DH 720K	N/A	1440	9	2	0xf9	112	1	2	3	2

Different types of disk drivers calculate these fields differently. The calculations for each type of disk driver are briefly explained below. The calculation of `secptrk` and `heads` for non-removable media is explained in section *Driver Initialization Entry* on page 3-7 since it is done by the driver initialization entry and stored for use by this `de_cntrl()` function. The calculation of `secptrk` and `heads` are given here only for removable media.

TABLE 3-3 RAM Disk

`start_rsect`, `sys_ind`	Zero since unpartitioned.
`media`	Always use `0xF8`.
`secptrk`, `heads`	Pick any legal values; for example: `secptrk` 1 to 63 and heads 1 to 16. It is optimal to pick values whose product divides evenly into `nsect`, but this is not necessary for a RAM disk. For a simple way, see the pSOSystem RAM disk driver.
Optional fields	All optional fields are zero. You could supply values to match the standard floppy formats listed in Table 3-2 on page 3-19 when the size matches one of the sizes in the table. Normally, this is not needed. However, it does allow you to test an application that uses a floppy disk with a RAM disk of exactly the same format.

TABLE 3-4 Floppy Disk

`start_rsect`, `sys_ind`	Zero since unpartitioned.
`media`, `secptrk`, `heads`, optional fields	Sense the density of the floppy disk in the drive. If this is not possible, either hard code one size if only one size floppy disk is used, or add a device specific `de_cntrl()` function code to set the size of floppy disk in the drive. Supply the corresponding values from the table of standard floppy formats to reproduce the proper size standard floppy disk format.

TABLE 3-5 SCSI Controlled Floppy Disk

`start_rsect`, `sys_ind`	Zero since unpartitioned.
`media`, `secptrk`, `heads`, optional fields	Use the SCSI READ_CAPACITY command to determine the size of the floppy disk in the drive. Supply the corresponding values from the table of standard floppy formats to reproduce the proper size standard floppy disk format.

TABLE 3-6 Partitioned SCSI Disk

`start_rsect`, `sys_ind`	From the partition table computed by the driver initialization entry.
`media`	Always use `0xF8`.
`secptrk`, `heads`	Use the logical geometry values computed by the driver initialization entry.
Optional fields	All optional fields are zero.

TABLE 3-7 Unpartitioned SCSI Disk

`start_rsect`, `sys_ind`	Zero since unpartitioned.
`media`	Always use `0xF8`.
`secptrk`, `heads`	Use the logical geometry values computed by the driver initialization entry.
Optional fields	All optional fields are zero.

TABLE 3-8 Partitioned IDE Disk

`start_rsect`, `sys_ind`	From the partition table computed by the driver initialization entry.
`media`	Always use `0xF8`.
`secptrk`, `heads`	Use the logical geometry values computed by the driver initialization entry.
Optional fields	All optional fields are zero.

TABLE 3-9 Unpartitioned IDE Disk

`start_rsect`, `sys_ind`	Zero since unpartitioned.
`media`	Always use `0xF8`.

TABLE 3-9 Unpartitioned IDE Disk (Continued)

`secptrk`, `heads`	Use the logical geometry values computed by the driver initialization entry.
Optional fields	All optional fields are zero.

DISK_REINITIALIZE

Repeat the initialization done at the driver's initialization entry or when a disk is changed. Primarily, this consists of re-reading the on-disk partition tables and re-initializing the partition table variables in the disk driver. This `de_cntrl()` function would be called by a pSOSystem disk partition application after it changes disk partition tables so that future disk accesses would be according to the new partition tables.

Media Change

The `IOPB` parameter block is `struct disk_ctl_iopb` with the union field `removed_call_back`. That contains the address of a disk removed call back within pHILE+. The disk driver needs to remember this value in a local variable so that it can be called when media is removed. The callback routine has the following C definition:

```
unsigned long (*)(unsigned long dn, unsigned long bitmask);
```

SCSI (Small Computer System Interface) Driver

Description

The SCSI driver is divided into two components which are referred to as the upper and lower level drivers. The source file named, `scsi.c`, which resides in the `drivers` directory comprises the upper level driver and the source file named, `scsichip.c`, which resides in the `src` directory makes up the lower level driver. The location of the `src` directory depends on the CPU board as the following path illustrates:

bsps/*board*/src

where *board* can be, for example, m162, m167, and so on.

The `scsi.c` source 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` file contains the driver code that responds to a SCSI call by formatting a SCSI command and passing it the lower level driver. The code within the source file, `scsichip.c`, contains instructions that execute the SCSI operation in a method determined by the hardware on the individual board.

The `scsichip.c` lower level driver is the software interface to the SCSI device interface (often one or more SCSI chips). The code within, `scsichip.c`, contains instructions that take the SCSI operation passed from the upper level 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 level driver.

User Interface

The application interacts with the SCSI driver through system calls to the pSOS+ kernel. These system calls are described in this section and are:

```
de_init        de_read        de_write       de_cntrl
```

NOTE: The `de_read` and `de_write` 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_init

The `de_init` function initializes the SCSI driver. It must be the first call to the SCSI driver. The syntax for this function is:

```
de_init(long DEV, long *IOPB, long *return, long *data_area)
```

where:

`DEV`	The major device number of the SCSI driver.
`IOPB`	Not used.
`return`	Not used.
`data_area`	Not used.

NOTE: `IOPB`, `return` and `data_area` are pointers that the SCSI driver does not use and must be set to NULL.

de_read

The `de_read` function is used to read from a SCSI device. The syntax for this function is:

```
de_read(long Dev, long *IOPB, long *return)
```

where:

`DEV`	`Dev` is the result of a logical 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`	`IOPB` is a pointer to the SCSI-specific I/O parameter block structure. (See section, *SCSI Specific I/O Parameter Block* on page 3-26.)
`return`	The `return` parameter is a pointer to the return value.

de_write

The de_write function is used to write to a SCSI device. The syntax for this function is:

```
de_write(long Dev, long *IOPB, long *return)
```

where:

DEV	Dev is the result of a logical 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	IOPB is a pointer to the SCSI-specific I/O parameter block structure. (See section, *SCSI Specific I/O Parameter Block*.)
return	The return parameter is a pointer to the return value.

de_cntrl

The de_cntrl call performs special functions on the SCSI device. Although de_cntrl can be optional for some devices, the SCSI interface requires it. The SCSI commands are described in section, *SCSI Control Functions* on page 3-27. The syntax for this function is:

```
de_cntrl(long Dev, long *IOPB, long *return)
```

where:

DEV	Dev is the major/partition/minor device number. Bits 0 through 7 are the minor number of the SCSI device. This is the SCSI target ID of the device. Bits 8 through 15 are the partition number on the hard disk drive. **NOTE:** Partition is only used for hard disk drives. Bits 16 through 31 are the major number for the SCSI driver. For example, device 0x50203, is the SCSI driver in the device switch table index 5. The partition number of the hard disk drive is 2 and the SCSI target ID of the device is 3.
IOPB	IOPB is a pointer to the SCSI-specific I/O parameter block structure. (See section, *SCSI Specific I/O Parameter Block*.)
return	The return parameter is a pointer to the return value.

SCSI Specific I/O Parameter Block

The specification for the SCSI specific I/O parameter block (IOPB) that the de_read and de_write functions pass, is contained in the phile.h include file. The phile.h file is located in the include directory. The IOPB 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;
```

Table 3-10 lists the only elements from the buffer_header structure that a SCSI driver uses:

TABLE 3-10 buffer_header elements Used by the SCSI Driver.

Element	Description
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 (in base 2) to be read or written.

SCSI Control Functions

The #define statements for the SCSI control commands reside in the drv_intf.h include file. This file resides in the include directory. The SCSI-specific IOPB that the de_cntrl function call passes appears in the drv_intf.h file as follows:

```
struct scsi_ctl_iopb
   {
   long function
   union
      {
      void * arg;
      struct scsi_info info;
      struct scsi_cmd cmd;
      } u;
   }
```

Table 3-11 lists the SCSI control functions supported:

TABLE 3-11 SCSI Control Commands

Command	Description
SCSI_CTL_FORMAT	Formats the disk drive referenced by the minor number specified in Dev.

SCSI_CTL_INFO	Fills in the info element of the scsi_ctl_iopb structure for use by the application. The scsi_info structure appears in the drv_intf.h file as follows:
	```
	struct scsi_info
	 {
	  UCHAR devtype; /* Type of device - values
	                    defined below */
	  UCHAR scsi_id; /* Device address on the
	                    SCSI bus */
	  UCHAR char lun; /* Device LUN */
	  UCHAR removable;/* Removable media */
	  char vendor[SCSI_VENDOR_SIZE];
	                 /* Device's manufacturer */
	  char product[SCSI_PRODUCT_SIZE];
	                 /* Model name */
	  long blocks;   /* Capacity in blocks */
	  long blocksize;/* Size of each block in
	                    bytes */
	   };
	```

TABLE 3-11 SCSI Control Commands (Continued)

Command	Description	

`SCSI_CTL_CMD`	Passes a SCSI command through the SCSI driver. The application must fill in the command structure. The `scsi_cmd` structure appears in the `drv_intf.h` file as follows:
	`struct scsi_cmd`
	`   {`
	`   UINT target_id;`
	`   UCHAR *data_ptr;`
	`   ULONG data_in_len;`
	`   ULONG data_out_len;`
	`   ULONG command_len;`
	`   UCHAR *cdb;`
	`   };`
	where:	

	`target_id`	Is the SCSI device ID.
	`data_ptr`	Points to the data buffer.
	`data_in_len`	The byte length of the buffer that stores input data. If the data is output data, this must be 0.
	`data_out_len`	The byte length of the buffer that stores output data. If the data is input data, this must be 0.
	`command_len`	Is the byte length of the SCSI Command Descriptor Block (CDB)
	`cdb`	Points to the CDB. The CDB describes the actual command that goes to the SCSI device. The CDB and SCSI command must be supported by the SCSI device being addressed.

TABLE 3-11 SCSI Control Commands (Continued)

Command	Description	
SCSI_CTL_TEST_UNIT_READY		
	Issues a test unit ready command to the device. If the device is ready, the `de_cntrl` function returns 0. If the device is not ready, the `de_cntrl` function returns a nonzero value and `retval` contains the SCSI driver error code.	

SCSI_CTL_PARTITION	Partitions a disk drive into a group of primary partitions that can then be made into separate file systems. This command takes as an argument a pointer to an array of four of the following structures:
	
	`typedef struct`
	`{`
	`  int begin;  /* beginning physical block */`
	`  int size;   /* size in physical (512 byte)`
	`                 blocks */`
	`} PARTITION_ENTRY;`
	
	where:	

	`begin`	The `begin` element is the beginning physical block number of the partition.
	`size`	The `size` element is the number of physical (SCSI 512 byte) blocks in the partition. It is not the number of logical (pHILE+) blocks.

TABLE 3-11 SCSI Control Commands (Continued)

Command	Description
	This typedef is located in `include/drv_intf.h` header file. An example of code to partition a disk drive is as follows:

```
PARTITION_ENTRY part_info[4];

part_info[0].begin = 0;
part_info[0].size = 20000;

part_info[1].begin = 20000;
part_info[1].size = 20000;

part_info[2].begin = 40000;
part_info[2].size = 20000;

part_info[3].begin = 60000;
part_info[3].size = 20000;

iopb.function = SCSI_CTL_PARTITION;
iopb.u.arg = (void *)part_info;

if (error= de_cntrl(dev_harddisk, (void
*)&iopb, &retval))
{
      printf("ioctl de_cntrl
SCSI_CTL_PARTITION error %x\n", error);
      k_fatal(retval, 0);
}
else
      printf("ioctl de_cntrl
SCSI_CTL_PARTITION done\n");
```

Command	Description
	This partitioning is not a MS DOS primary partition and cannot be used by MS DOS.
`SCSI_CTL_READ_ONLY_P`	Marks the given partition referenced by `Dev` as read-only and the SCSI driver does not allow writes to this partition.
`SCSI_CTL_READ_WRITE_P`	Marks the given partition referenced by `Dev` as read/write and the SCSI driver allows reads or writes to the partition (default state).
`SCSI_CTL_START_DEVICE`	Issues a start/stop unit command to the device to start the device. This tells a disk drive to spin up and become ready.

TABLE 3-11 SCSI Control Commands (Continued)

Command	Description
`SCSI_CTL_STOP_DEVICE`	Issues a start/stop unit command to the device to stop the device. This tells the device to spin down and stop.
`SCSI_CTL_STOP_COMMANDS`	
	Stops any queued SCSI commands from being sent to the device. This does not abort any command that is in progress.
`SCSI_CTL_START_COMMANDS`	
	Allows commands to be sent to the device. This is the default state.
`SCSI_CTL_SKIP`	Skips to the next file mark on a tape device. The `arg` element of the `scsi_ctl_iopb` must be set to the number of file marks to be skipped.
`SCSI_CTL_REWIND`	Issues a rewind command to a tape device.
`SCSI_CTL_UNLOAD`	Issues an unload command to a tape device.
`SCSI_CTL_ERASE`	Issue an erase tape command to a tape device.
`SCSI_CTL_WRITE_FILE_MARK`	
	Causes a file mark to be written to a tape device.
`SCSI_CTL_SET_BLOCK_MODE`	
	Sets a tape device to block mode and sets the block size. The `arg` element of the `scsi_ctl_iopb` must be set to the block size.

SCSI Tape Drive

The application interface to the SCSI tape driver consists of open, close, read, write, and I/O control calls. These are all done through the pSOS+ `de_` interface function.

Tape Open

The open tape command has the following purposes.

- It allows the user to have exclusive control of the tape device. Only the task that has opened the tape device may use the tape device.
- It tells the driver to rewind the tape device when the device is closed.

The syntax of the tape open command is:

```
de_open(unsigned long dev_tape, void *iopb, void *retval);
```

where:

`dev_tape`	The input of the MAJOR/MINOR number of the tape device.
`iopb`	An input pointer to a `scsi_open_iopb` structure.
`retval`	An output that points to a driver specific return value.

Tape open uses the following `iopb` structure:

```
struct scsi_open_iopb
{
unsigned char exclusive;
unsigned char rewind;
};
```

where:

`exclusive`	The element `exclusive` is set to one by the caller if the tape device is to be used only by the task that is opening the tape device. If `exclusive` is zero then the SCSI tape driver allows other tasks the use of the tape device at the same time.
`rewind`	The element `rewind` is set to one by the caller if the tape is to be rewound on close. If `rewind` is set to zero the tape in the tape device remains at the next block to be accessed on close.

Tape Close

Closing a tape device releases the tape drive from exclusive control of a task, if that is the way it was opened, or decrements the count of tasks that have the tape driver open, if it was not opened with the exclusive control flag set.

The syntax of the tape close command is:

```
de_close(unsigned long dev_tape, void *iopb, void *retval);
```

where:

dev_tape	The input of the MAJOR/MINOR number of the tape device.
iopb	Not used.
retval	An output that points to a driver specific return value.

Tape READ/WRITE

The tape read and write commands are used to read and write data to a tape drive. Both of these commands have the same syntax as follows:

```
de_read(unsigned long dev_tape, void *iopb, void *retval);
de_write(unsigned long dev_tape, void *iopb, void *retval);
```

where:

dev_tape	The input of the MAJOR/MINOR number of the tape device.
iopb	The input pointer to a TAPE_BUFFER_HEADER structure. This structure is found in drv_intf.h and is defined below.
retval	An output that points to a driver specific return value.

The TAPE_BUFFER_HEADER structure is defined as follows:

```
typedef struct scsi_rw_iopb
   {
   unsigned long b_device;      /* device number*/
   unsigned long b_bcount;      /* data length*/
   void *b_bufptr;              /* data area*/
   } TAPE_BUFFER_HEADER;
```

where:

b_device	The MAJOR/MINOR number of the tape device.

`b_bcount`	The data length for this transfer. The data length may be the number of characters to be transferred, if the tape is being used in a variable block size mode, or the number of blocks to transfer, if the tape is in a fixed block mode. The mode is controlled by a `SCSI_CTL_SET_BLOCK_MODE` I/O control call.
`b_bufptr`	Pointer to the data area the transfer takes place from, if this is a read call, or to if this is a write call.

pSOS-to-Driver Interface

Table 3-12 and Table 3-13 list the I/O switch table entries needed to use the SCSI drivers.

To use the SCSI disk driver:

TABLE 3-12 Function Entry Point for SCSI Disk Driver

Function	Entry Point
Initialization (`Init`)	`SdrvInit`
Open	`N/A`
Close	`N/A`
Read	`SdskRead`
Write	`SdskWrite`
I/O Control	`SdrvCntrl`

To use the SCSI tape driver:

TABLE 3-13 Function Entry Point for SCSI Tape Driver

Function	Entry Point
Initialization (`Init`)	`SdrvInit`
Open	`StapeOpen`
Close	`StapeClose`
Read	`StapeRead`

TABLE 3-13 Function Entry Point for SCSI Tape Driver (Continued)

Function	Entry Point
Write	`StapeWrite`
I/O Control	`SdrvCntrl`

The pSOS-to-driver interface consists of data structures and subroutine calls (shown below) 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 upper driver subroutines listed above create the proper SCSI CDB and call lower driver routines to communicate with the SCSI device interface.

Upper-to-Lower Driver Interface

The interface between the upper level and lower level SCSI drivers has the following features:

- Provides the support for the lower level drivers which service more than one SCSI chip.
- Allows the integration of multiple lower level SCSI drivers into the system.

The upper level driver accesses the lower level driver through the SCSI multiplexor which maintains the SCSI driver table. Before a lower level driver can be used, it must be installed by calling `InstallSCSIDriver()`. If the SCSI lower level driver services multiple SCSI chips, the `InstallSCSIDriver()` function must be called for each chip. This function stores the lower level driver and the SCSI chip related information in the SCSI driver table.

To identify a SCSI chip serviced by a lower level driver, the logical adapter number and the physical adapter number are introduced. Each SCSI chip in the system has a unique logical adapter number. This number is actually the index into the SCSI driver table and it is unique because each SCSI chip has a unique entry in the SCSI driver table. The physical adapter number is used to identify a SCSI chip within the given SCSI lower level driver. The physical adapter number is only used by the interface between the SCSI multiplexor and the lower level SCSI drivers. This number is zero based, meaning the first SCSI chip serviced by a lower level driver has the value 0.

The lower driver, which is contained in the `scsichip.c` source file, must include the subroutines listed below. 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 initializes the lower SCSI driver and returns status. The upper driver subroutine `SdrvInit` calls `chipinit`.
`long dma_init()`	This function initializes any DMA device that SCSI operation requires and then returns 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. The `chipexec` function is the entry point to the lower driver from the upper driver.

The status that the `chipexec` function returns must be one of the following:

TABLE 3-14 `chipexec` Return Status

Status	Value	Description
`STAT_OK`	0	Operation was successful.
`STAT_CHECKCOND`	1	Contingent allegiance condition (issue a Request Sense to get cause of failure).
`STAT_ERR`	2	A gross error has occurred. A command may be formatted incorrectly. A retry may correct the problem.
`STAT_TIMEOUT`	3	A selection timeout occurred. More than likely no device exists at this SCSI ID.
`STAT_BUSY`	4	Device is busy. It cannot accept another command.
`STAT_SEMFAIL`	5	An `sm_p` operation has failed, and the driver cannot continue with the current command.

The statuses that can be returned by the `chipexec` function are specified using `#define` statements. These declarations are contained in the `scsi.h` include file, which is located 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 trans_blk transaction structure to the lower driver by executing the chipexec subroutine. The structure contains all the information needed to perform the requested SCSI operation.

Function Calls of the Low Level SCSI Driver

The lower level SCSI driver must provide the following six functions to the upper level driver ($PSS_ROOT/drivers/scsi.c) to execute the corresponding requests:

TABLE 3-15 Low Level SCSI Driver Function Calls

Function	Request Description
Init	Initialize the interface.
DmaInit	Initialize the DMA channel used by the SCSI chip.
Exec	Execute the SCSI commands.
Shutdown	Execute the shutdown operation.
StopCmd	Stopping the SCSI chip.
StartCmd	Starting the SCSI chip.

Init

Before the SCSI chip can be used, the lower level driver and the SCSI chip must be initialized. The syntax of the initialization function is:

```
long  (* Init) (int adaptorNum,
                SCSI_DRV_CFG *drvCfg,
                SCSI_DRV_INFO *drvInfo);
```

where:

`adaptorNum`	The physical adapter number that identifies which SCSI chip is being addressed, within the lower level SCSI driver.
`drvCfg`	Points to the data structure containing the hardware parameters needed by the lower level SCSI driver for its SCSI chip and data structure initialization. (See the description of the function `InstallSCSIDriver()` in *Installing a Lower-Level SCSI Driver* on page 3-42.)

When the lower level driver successfully completes the initialization, `drvInfo`, points to the data structure which is filled by the lower level driver with the following information:

`adptNum`	The physical adapter number of this SCSI chip.
`hostID`	SCSI ID of this SCSI chip.
`maxTargets`	The maximum number of target devices. This number may be either 8 or 16, depending on whether or not the SCSI chip supports wide SCSI.
`queSemFlag`	Indicates that the lower level driver needs a semaphore for interrupt driven I/O processing. The semaphore is created by the upper level driver for each TRANS_BLK.
`dmaSemFlag`	Indicates that the lower level driver needs a semaphore created for each target device for DMA processing.
`chip_tb_size`	The size of the lower level driver specific control block for each SCSI command execution. This block is allocated by the upper level driver and passed to the lower level driver when the upper level driver sends the SCSI command execution request to the lower level driver.

Both data structures of SCSI_DRV_CFG and SCSI_DRV_INFO are defined in the header file, $PSS_ROOT/include/scsi.h.

Upon successful completion of the initialization function, STAT_OK is returned. If initialization fails, an error code is returned.

DmaInit

Some lower level drivers use DMA channels which must be initialized before the driver can be used. The syntax of the initialization function for the DMA channel is:

```
long   (*DmaInit) (int adaptorNum);
```

where:

adaptorNum	The physical adapter number.

When the DMA channel initialization is complete, STAT_OK is returned. If initialization fails, an error code is returned. When no DMA channel is being initialized, the lower level driver must provide an empty function stub which returns, STAT_OK.

Exec

To execute a SCSI command, the upper level SCSI driver calls the SCSI command execution function provided by the lower level driver. The syntax of this function is:

```
long     (* Exec)(TRANS_BLK *pTransBlk);
```

where:

pTransBlk	Points to the TRANS_BLK data structure specifying the SCSI command being executed and the input and output data areas. The structure is defined in the header file, $PSS_ROOT/include/scsi.h.

When the SCSI command execution is complete, STAT_OK is returned. If the SCSI command execution fails, an error code is returned.

Shutdown

To execute a shutdown operation, the upper level SCSI driver calls the shutdown function provided by the lower level driver. The syntax of this function is:

```
long     (* Shutdown)(int adaptorNum);
```

where:

`adaptorNum`	The physical adapter number.

Upon receiving the shutdown request, the lower level driver must release all resources it has been allocated for the specified SCSI chip. The lower level driver must also disable the SCSI chip. After shutdown, the lower level driver does not allow other requests for the specified SCSI chip until the SCSI chip initialization is called again.

When the shutdown operation is complete, STAT_OK is returned. If the shutdown operation fails, an error code is returned.

StopCmd/StartCmd

Sometimes, the application may want to stop the SCSI chip activities and resume them later. To stop and start the SCSI chip activities, the upper level SCSI driver calls the functions provided by the lower level driver. The syntaxes of these functions are:

```
void     (* StopCmd)(int adaptorNum);
void     (* StartCmd)(int adaptorNum);
```

`adaptorNum`	The physical adapter number.

Installing a Lower-Level SCSI Driver

Before a SCSI chip can be accessed from the upper level SCSI driver, the lower level SCSI driver for this SSCI chip must be installed. A lower level SCSI driver is installed by calling the function `InstallSCSIDriver()`.

The syntax of this function is:

```
long InstallSCSIDriver(
                       int logAdaptorNum,
                       int phyAdaptorNum,
                       SCSI_DRV_FUNCS *sfuncs,
                       SCSI_DRV_CFG   *scfg,
                         unsigned long *pMinor);
```

where:

`logAdaptorNum`	Specifies the logical adaptor number for this SCSI chip if it is not `SCSI_ANY_LOG_ADPT`. Otherwise, a free logical adaptor number is allocated.
`phyAdaptorNum`	The physical adaptor number identifying the SCSI chip serviced by the lower level driver.
`sfuncs`	Points the data structure containing the function pointers of the lower level driver. This data structure is not copied, only its address is remembered in the SCSI driver table. Therefore, this data structure should not reside, for example, in the stack area of the caller.
`scfg`	Points to the data structure containing the hardware parameters which are passed to the lower level driver when the initialization function is called. These parameters are copied into the SCSI driver table. The meaning of these parameters are lower level driver specific. They are not used or interpreted by the upper SCSI driver and the SCSI multiplexer. For example, the caller can specify the I/O address of the SCSI chip, the interrupt level and the SCSI ID of the chip in this data structure.
`pMinor`	Points to the location where the minor number for the SCSI chip is returned.

If the lower level SCSI driver services more than one SCSI chip, for each SCSI chip the `InstallSCSIDriver()` function must be called.

The following values are returned by this function:

STAT_OK	The request is successfully executed.
SCSI_ADAPTOR_NUM_BAD	The given adaptor number is invalid, for example, if it exceeds the maximum adapter number specified by BSP_MAX_SCSI_ADAPTORS.
SCSI_MUXFULL	The SCSI driver table is full.
SCSI_DRVINUSE	A SCSI lower level driver is already installed for the logical adaptor specified by the input parameter logAdaptorNum or the lower level SCSI driver is already installed for the SCSI chip specified by phyAdaptorNum.

Uninstalling a Lower level SCSI Driver

To uninstall a SCSI lower level driver, the following function call is used:

```
long UninstallSCSIDriver(unsigned long minor, SCSI_DRV_FUNCS
*sfuncs);
```

minor	The minor number returned through the pMinor parameter when the driver is installed.
sfuncs	Points to the data structure which must contain the same content as the one when the driver has been installed, see the description of InstallSCSIDriver(). This parameter is used to verify if the request is valid or not.

The following values may be returned by this function:

STAT_OK	The request is successfully executed.
SCSI_ADAPTOR_NUM_BAD	The given adaptor number is invalid, for example, if it exceeds the maximum adapter number specified by BSP_MAX_SCSI_ADAPTORS.
SCSI_DRVBADTYPE	No SCSI lower level driver is installed for this SCSI chip or the content of the data structure pointed by sfuncs does not match the one given by the function InstallSCSIDriver().

If the lower level SCSI driver services more than one SCSI chip and you want to remove the lower level driver completely from the SCSI multiplexer, you have to call the `InstallSCSIDriver()` function for each chip serviced by this lower level SCSI driver.

SCSI Multiplexor Initialization

The SCSI Multiplexor is initialized by calling the function, `InitSCSIMux()`. The syntax of this functions is:

```
void InitSCSIMux(void);
```

When the upper level SCSI driver is called, it indirectly calls `InitSCSIMux` if the SCSI Multiplexer is not initialized. The BSP or application can also explicitly call this function before any parts of the upper level driver code is executed.

The `InitSCSIMux()` function calls `brdInstallSCSIDriver()`, the board specific function. The syntax of this function is:

```
void brdInstallSCSIDriver(void);
```

This function must be provided by the BSP which installs its lower level SCSI drivers during the execution of this function.

Driver Error Codes

The error codes returned by the lower level SCSI driver are divided into two groups. One group is common for all lower level drivers and the other group is lower level driver specific. The error codes in the lower level driver specific group have the values between 0 to 0xFF. The error codes in the common group are defined in the header file, `$PSS_ROOT/include/scsi.h` and have the values ranging from 0x0100 to 0xFF00. Below are listed some of the common error codes and their values:

TABLE 3-16 Common SCSI Driver Error Codes

Error Code	Value
`STAT_CHECKCOND`	0x100
`STAT_ERR`	0x200
`STAT_TIMEOUT`	0x300
`STAT_BUSY`	0x400
`STAT_SEMFAIL`	0x500

SCSI Device Minor Number

The 16-bit minor device number passed to the upper level SCSI driver is divided into the following parts:

```
15   13 12          8  7          5  4         0       Bit
Unit    |    Partition   |   Adapter   |    SCSI ID
```

where:

Unit	The logical unit number within a SCSI target.
Partition	The partition number. This is only valid for a partitioned SCSI block device.
Adapter	The logical adaptor number identifying the lower level driver and the SCSI chip serviced by this driver.
SCSI ID	The SCSI ID of the device connected to the SCSI bus serviced by the given SCSI chip.

Board Specific Functions and Macros

The following macros should be contained in the `bsp.h` header file:

`BSP_SCSI_DRV_LEVEL`	Defines whether the lower level SCSI drivers provide the pSOSystem 2.5 interface. If you are using the SCSI driver written for pSOSystem 2.1.x and 2.2.x, you set this macro to 220. If this macro is not defined, the upper level driver assumes the lower level SCSI drivers comply with the pSOSystem 2.5 interface.
`BSP_MAX_SCSI_ADAPTORS`	Specifies the maximum number of SCSI chips supported by this BSP. If this macro is not defined, it assumes the maximum number is 1.

The following function has to be provided by the BSP if the lower level SCSI drivers comply with the pSOSystem 2.5 interface:

```
void brdInstallSCSIDriver(void);
```

This function is called when the SCSI multiplexer is initialized. Normally, it installs the lower level SCSI drivers supported by the BSP.

4 Standard pSOSystem Character I/O Interface

Overview

The standard pSOSystem character I/O Interface, defines the interface through which application programs interact with the character oriented I/O devices. It also serves as a reference to the developers working on writing drivers for character oriented devices to work under pSOSystem environment. The character driver interface definition is an extension to the standard pSOS+ device driver interface, which is documented in the *pSOSystem System Concepts* manual; it merely defines the structure of the I/O Parameter Block (IOPB) passed to the various driver entry points. This chapter assumes that you are already familiar with the standard pSOS+ device driver interface.

The chapter contains a description of the character I/O interface as well as the details for individual character oriented I/O device drivers provided by the pSOSystem.

The individual I/O device drivers described in this chapter are:

- HTTP (See page 4-12)
- pSEUDO (See page 4-40)
- MEMLOG (See page 4-29)
- NULL (See page 4-32)
- PIPE (See page 4-34)
- RDIO (See page 4-40)
- TFTP (See page 4-44)
- DITI (See page 4-49)

Who must understand and follow this specification?

As an application developer, you need to understand this specification if your application needs to directly interact with the device through the de_* services provided by the pSOS+ kernel. If you are using the standard pREPC+ ANSI library functions or the C++ I/O Streams package to perform I/O to and from the device, it is not necessary to have a detailed understanding of the character I/O interface. However, you may still need to understand a few details, like how to address the device (such as the device naming convention) and how to pass optional parameters that control the device.

All the standard pSOSystem drivers that perform character oriented I/O follow this specification, and many custom drivers for character oriented I/O device are expected to follow this specification.

As a driver developer, pSOSystem does not require you to follow any specific I/O driver interface, however, if you want your driver to work with other pSOSystem software (like pREPC+ ANSI C library and C++ I/O Streams) that perform character oriented I/O, you must write the driver to follow this specification.

What is a character oriented I/O device?

A character oriented I/O device does not impose any boundaries like records or blocks around the data (as opposed to block oriented devices). An I/O request to character device can request any amount of data to be read from or written to the device. The character oriented devices, typically, do not have random access capability, and the data originating from (or destined to) the device can be thought of as a single continuous byte stream.

When performing I/O to a character oriented device, once some data is read from the device, it is removed from the data stream; there is usually no way to obtain the same data item again. Similarly, once a data is sent to a character oriented device, it is committed and cannot usually be overwritten or discarded. This behavior is typical of the most common character oriented devices like asynchronous and synchronous serial communication devices.

It is possible to write a driver for a device that has random access capability as a character oriented device driver (and some drivers do support both block and character interfaces). However, there is no way to utilize the random access capability of such a device, through the character oriented device interface.

The Interface

Device Initialization Entry

The device initialization entry point of a driver is invoked as a result of a `de_init()` call made by the application. There is no I/O Parameter Block (IOPB) defined for this entry. If the device is configured to allow auto-initialization by pSOS+, the drivers must expect a NULL pointer to be passed as the IOPB. No other pSOSystem components explicitly invoke the initialization entry of a driver.

Under pSOSystem environment, device initialization is typically the responsibility of the ROOT task. If there are circumstances whereby a custom driver needs to obtain some configuration parameters from the application at the initialization time, it is allowed to have the driver define an IOPB for device initialization.

Device Open Entry

The open entry point of a driver is invoked as a result of `de_open()` call made by the application. A pointer to the following `PssCharOpenIOPB` structure (which is defined in include file, `drv_intf.h`) is passed by the application as the second argument of `de_open()` function:

```
typedef struct {
  char           *params;
  unsigned long   flags;
  unsigned long   cloneDev;
} PssCharOpenIOPB;
```

The `params` field is a null-terminated string that contains optional parameters that are interpreted by the driver being opened. The `params` string consists of one or more comma separated name value pairs of the form:

`name1=value1,name2=value2`

The `flags` field is currently reserved to pass file open mode information to the driver. No values are currently defined for `flags`, and the driver writers should neither define nor depend on any value passed by way of the `flags` parameter.

The `cloneDev` field exists to support cloneable devices. Cloneable devices like PSEUDO, TFTP, PIPE, HTTP, for example, provide multiple I/O channels, each of which offers the same capabilities. Since an application has usually no interest in opening a specific channel, such drivers accept a parameter (typically of the form `channel=-1`) to let an application request the driver to assign an available channel. The `cloneDev` field is used by the driver to return the device number corresponding to the channel opened by it. Note that the drivers that do not support cloning may not necessarily modify this field to indicate the channel opened. However, the drivers that support cloning, always modify this field to indicate the channel opened, even if the request is made to open a specific channel.

An instructive way to understand how these parameters are used is to describe them as defined by the TFTP driver. When opening a channel on a TFTP driver for the purpose of transferring a file from a remote TFTP server, the driver open routine needs to know the IP address of the remote host, as well as the pathname of the file on the remote server to be transferred. For example, in order to request a TFTP driver to open a connection with remote TFTP server at IP address 192.103.54.36 to transfer the file, `/programs/pdemo.hex`, the TFTP driver expects a parameter string of the form:

```
ipaddr=192.103.54.36,pathname=/programs/pdemo.hex
```

Device Close Entry

The close entry point of a driver is invoked as a result of a `de_close()` call made by the application. There is no I/O Parameter Block (IOPB) defined for this entry.

Device Read Entry

The read entry point of a driver is invoked as a result of `de_read()` call made by the application. A pointer to the `PssCharRdwrIOPB` structure (which is defined in include file `drv_intf.h`) is passed by application as the second argument of the `de_read()` function:

```
typedef struct {
  unsigned long   count;
  char           *address;
} PssCharRdwrIOPB;
```

The `count` field contains the maximum number of bytes that the driver should transfer from the device to the buffer pointed to by the `address` field. The actual number of bytes transferred by the driver is returned back to the caller by way of the third argument (`retval`) of the `de_read()` call.

A driver may block until the requested number of bytes are available, or return after transferring whatever amount of data can be transferred without blocking, depending on whether the driver is operating in blocking or non-blocking mode. A non-blocking read request usually returns an error if no data can be transferred and an end of file condition is usually signified by returning 0 as the number of bytes read.

These rules are followed by most of the drivers that follow character I/O driver interface, however, you must refer to individual driver manual page to verify if the driver makes an exception to these rules, and whether or not the blocking and non-blocking I/O modes are supported.

Device Write Entry

The write entry point of a driver is invoked as a result of `de_write()` call made by the application. The IOPB passed by application as the second argument of `de_write()` function is the same as the one passed to the device read entry:

```
typedef struct {
  unsigned long   count;
  char           *address;
} PssCharRdwrIOPB;
```

The `count` field contains the number of bytes that the driver should transfer from the buffer pointed to by the `address` field to the device. The actual number of bytes transferred by the driver is returned back to the caller in the third argument, (`retval`), of the `de_write()` call.

A driver operating in blocking mode may block until the data is committed to the device. A non-blocking write request attempts to write as much data as possible without blocking, and returns the number of bytes written. If it is not possible to write even a single byte of data an error is returned. If an end of file condition is defined, it is also usually signalled by returning an error.

These rules are followed by most of the drivers that follow character I/O driver interface, however, you must refer to the individual driver manual page to verify if the driver makes an exception to these rules, and whether or not the blocking and non-blocking I/O modes are supported.

Device I/O Control Entry

The I/O control entry point of a driver is invoked as a result of a `de_cntrl()` call made by the application. A pointer to the IOPB passed by the application as the second argument to the de_cntrl() function follows the following structure template (which is defined in include file `drv_intf.h`):

```
typedef struct {
  unsigned long   fcode;
  void           *arg[];
} PssCharCtrlIOPB;
```

The `fcode` field contains the function code of the I/O control operation to be performed, and any other parameters required by that I/O control operation are passed in one or more fields following `fcode`.

The I/O control operations are device specific, and you must refer to the documentation of the individual driver for further details.

Character I/O Drivers

Introduction

The remaining portion of this chapter describes the various standard device drivers provided with pSOSystem. The documentation for each driver follows a standard organization convention and the contents of this subsection:

- Explains how the information is organized and presented for the individual drivers
- Describes any special concepts and terminology used in the documentation, as well as, where to find additional information
- Lists the device drivers documented in this chapter, with a brief description of each

Audience

The driver documentation has been written from the perspective of application programmers developing code that performs I/O to or from devices in their applications. In addition to the information contained in the character I/O driver descriptions, developers writing device drivers for pSOSystem can find further information within the *pSOSystem System Concepts* manual, and within the *pSOSystem Advanced Topics* manual.

Organization

Each driver documented contains the following four sections:

- Overview
- Operation
- Reference Information
- Application Examples

These sections are described below:

Overview This section contains information about the purpose of the driver, the special features offered by it, and a high level overview of its operation.

Operation

This section describes the specifics of each I/O operation supported by the driver. Included is a description of the six services offered by each of the device entry points: initialization, open, close, read, write and control. This section contains information for those who wish to understand the device operation in detail.

Reference Information

This section contains a table of concise information related to driver configuration and usage. If you are already familiar with the operation of a driver, this section provides useful reference information. The table entries contained in the Reference Information section are described below:

Supported interfaces	This entry lists the standard I/O interfaces supported by the device driver. There are three standard I/O driver interfaces defined by pSOSystem, which are: character, block, and streams. It is recommended that you familiarize yourself with the standard interfaces supported before attempting to utilize a specific device driver. Character (for example, DITI and DISI) and block (for example, SCSI) device driver descriptions are covered in this chapter. Stream I/O device driver information can be found in the *OpEN User's Guide*.
Name registered in pSOS+ DNT	DNT (Device Name Table) is a feature offered by pSOS+ kernel that allows one to register symbolic names for a device number. This table entry lists the default device names that are registered for a device at pSOSystem system startup. More information about DNT can be found in the *pSOS+ Real-Time Kernel* chapter of the *pSOSystem System Concepts* manual.

pRNC name	pRNC (pSOSystem Resource Naming Convention) names are used when opening devices through standard ANSI functions provided by pREPC+. This table entry identifies the pRNC name of the device. Further information regarding pREPC+ can be found in the *pREPC+ ANSI C Library* chapter of the *pSOSystem System Concepts* manual.
Minor number encoding	The pSOSystem device is identified by a 32 bit device number. The lower 16 bits specify a *minor device* number. The individual device drivers define how the minor device numbers are encoded. This table entry explains how the driver interprets the information encoded in the minor number. For example, a driver serving multiple physical or logical devices, would assign a unique minor number to identify each device it serves. The upper 16 bits of the device number specify the *major device number*, and identifies the location of the device in the pSOS+ I/O Jump Table. The minor device number is not used by the pSOS+ device manager.
Driver specific parameters accepted by device open entry	The I/O parameter block, passed to a standard character driver contains a field called `params` which contains a string encoded *name value* pairs. These name value pairs are used for passing parameters to the device open function, and may contain essential information that a driver may need at the time of opening an I/O channel. This reference information lists the parameters that are accepted by the driver open entry, and their significance.

Associated `sys_conf.h` parameters	This entry contains information about the various configurable parameters, defined in the include file, `sys_conf.h`, that control device operation.
pSOSystem components required	This table entry lists the pSOSystem components whose services are accessed by the driver. The pSOSystem components currently defined are: pSOS+, pROBE+, pHILE+, pREPC+, pLM+, pNA+, pSE+, and pMONT+.
Resources used by the driver	This reference information describes various system resources (mutex, semaphores, queues, memory, sockets, for example) that are used by the driver. This information can assist you in configuring the system, ensuring that adequate resources are available during run-time for proper driver operation.

Application Examples

This section contains code fragments demonstrating how the main features of the driver could be accessed by application programs. In some cases, this may be the first place to look for developing key insight into the driver operation.

Supported Drivers

Table 4-1 lists the standard pSOSystem device drivers that are documented in this chapter.

TABLE 4-1 pSOSystem Device Drivers Documented in this Chapter

Driver	Description	Page
HTTP	A driver for transferring files from a remote server using the Hyper Text Transport Protocol (HTTP).	4-12
pSEUDO	A driver for dynamically mapping the standard I/O channels to other pSOSystem drivers that follow the Character I/O Interface.	4-17
MEMLOG	A driver for logging system wide error and diagnostic messages.	4-29
NULL	A driver that acts as a null data-source and infinite data sink.	4-32

TABLE 4-1 pSOSystem Device Drivers Documented in this Chapter (Continued)

Driver	Description	Page
PIPE	A driver that provides similar named POSIX like inter-task data communication facility.	4-34
RDIO	A driver for performing I/O to or from the terminal window of a remote debugger communicating with the target ROM monitor, pROBE+, over a serial channel or network connection	4-40
TFTP	A driver for transferring files from a remote server using the Trivial File Transfer Protocol (TFTP)	4-44
DITI	A driver for performing Device Independent Terminal I/O over synchronous and asynchronous serial communication channels.	4-49

HTTP

Driver for transferring files from an HTTP server

Overview

The HTTP driver provides a standard character device driver interface for obtaining files from an HTTP server. HTTP is a widely adopted protocol used on the Internet. The HTTP driver provides a standard interface so that any data stream obtained from an HTTP server can be manipulated using standard ANSI-C stdio facilities.

The driver supports multiple channels that can operate simultaneously. The number of channels can be configured by the `SC_MAX_HTTP_CHAN` parameter located in the `sys_conf.h` header file. The number of channels that can be simultaneously active may, however, be limited by the availability of other resources mentioned in the Reference Information section, later.

Operation

Device Initialization

The device initialization of the HTTP driver involves allocating memory for bookkeeping, and initializing the mutexes used for mutual exclusion. It is not necessary to explicitly initialize the driver because it is automatically initialized at the time of processing the first request to open the device. The initialization routine does not expect an IOPB, and ignores any IOPB passed to it.

Device Open

The HTTP driver open function establishes connection with the HTTP server, specified by the drivers specific parameters passed via the standard character driver open IOPB, as defined in section titled *Character Device Interface: Device Open* of this manual. The `params` field of the IOPB points to a string of comma separated name value pairs that is parsed by the driver open routine to obtain the IP address and port number of the HTTP server, the path name that identifies the file (or other data) to be obtained, and optionally the channel number of the HTTP driver to engage for the purpose of data transfer. The `flags` field of the IOPB is ignored.

Table 4-2 describes the format of various parameters as expected by the HTTP driver open routine.

TABLE 4-2 HTTP Parameters

Parameter	Syntax and Description
`channel`	A numeric value that specifies the channel number to be opened. This is an optional parameter. If the `channel` parameter is omitted then the minor number of the device passed to the `de_open` call specifies the channel to be opened, otherwise the device minor number to which de_open call is directed is ignored and the value of channel parameter determines the channel to be opened as follows: ■ If a positive or zero value is specified, it is taken as the channel to be opened. ■ If a negative value is specified, the driver allocates an available channel and passes the device major minor number corresponding to the allocated channel back to the caller by way of the `cloneDev` field of the IOPB.
`ipaddr`	This is a required parameter that specifies the IP address of the HTTP server. It can be a decimal, octal, or hexadecimal number. It can also be specified in the standard dot notation used to specify IP addresses.
`port`	This is an optional parameter that specifies the port number of the HTTP server. If it is omitted, the port number defaults to 80.
`pathname`	This is a required parameter and it specifies the pathname component of the URL (Uniform Resource Locator) that specifies the remote resource to be accessed. The remote resource can either be a file served by the server, or the output generated by a CGI (Common Gateway Interface) program.

Device Close

The HTTP driver close function terminates the connection with the HTTP server.

Device Read

The read operation follows the same interface and semantics as defined for the read operation for a standard pSOSystem character device driver interface. Refer to section, *Device Read Entry* on page 4-4, for details.

Device Write

The write operation is not defined for HTTP driver.

Device Control

There are no device control functions defined for HTTP driver.

Reference Information

Supported Interfaces		Standard character I/O interface
Name registered in pSOS+ DNT		`http`
pRNC Name		`///dev/http?<params>`
Minor number encoding		0 through 0xFFFF - channel number
Driver specific parameters accepted by device open entry	`channel`	The channel number to open
	`ipaddr`	The IP address of the HTTP server
	`port`	The port number on the HTTP server which to connect
	`pathname`	The path name component of the URL specifying the file to transfer from the HTTP server

Associated `sys_conf.h` parameters	`SC_DEV_HTTP`	The major number of the device driver, or 0 if not to be configured
	`DEV_HTTP`	The major number of device driver shifted left by 16 bits; can be used to create a valid pSOS+ device number
	`SC_MAX_HTTP_CHAN`	Maximum number of channels that can be simultaneously activated
pSOSystem Components required		pSOS+, pREPC+, pNA+
Resource requirement	Mutexes	One system wide
	Memory	12 bytes per channel
	Socket	One per active channel

Application Examples

The following example code fragment demonstrates how the de_open call can be used to open a connection with the HTTP server at IP address 192.103.54.36 and (default) port 80 to transfer a file named app.hex from the directory /download relative to the ROOT of document tree on HTTP server. The channel parameter is passed as -1 to request that the driver allocate any available channel.

```
PssCharOpenIOPB   open_iopb;
PssCharRdwrIOPB   rdwr_iopb;
Unsigned long     rc, retval;
char              buf[0x200];

open_iopb.params =
 "channel=-1,ipaddr=192.103.54.36,pathname=/download/app.hex";

rc = de_open(DEV_HTTP, &open_iopb, &retval);
if (rc != 0)
    k_fatal(rc, 0);

rdwr_iopb.count = 0x200;
rdwr_iopb.address = buf;

rc = de_read(open_iopb.cloneDev, &rdwr_iopb, &retval);
```

The following example code fragment demonstrates how HTTP driver can be used with the standard ANSI stdio functions provided by pREPC+.

```
FILE            *fp;
char              buf[0x200];

/*
 * Open the channel number two of the HTTP driver
 */
fp = fopen(
    "///dev/http?channel=2,ipaddr=204.71.177.159,pathname=/",
     "r" );

if (fp == NULL) {
    perror("Open");
    exit(1);
}

while(!eof(fp)) {
     count = fread(buf, 0x200, 1, fp);
     fwrite(buf, count, 1, stdout);
}
```

pSEUDO Driver

General purpose I/O redirection driver.

Overview

This section describes the implementation of the pREPC+ pSEUDO driver in the pSOSystem environment. In earlier versions of pSOSystem, the serial console is the default `stdin`, `stdout` and `stderr` device. By providing a pSEUDO driver, the user can redirect input and output to and from a specific device, other than the default serial device. With this framework in place, the redirection of input and output can be carried out on low-level drivers like Memory Log Driver (`memlog`), RBUG driver (`rdio`), Network driver, DITI driver or NULL (`null`) device driver.

This section also provides the programming interface to redirect the input and output to the desired device. Also provided is sample application code to redirect input and output located in the section name *Sample Application Source Listing* on page 4-27.

The pSEUDO driver provides a standard character driver interface for redirecting I/O to:

- another device driver that follows the standard character driver interface (example of character device drivers that can be used with pSEUDO driver are: DITI, HTTP, MEMLOG, NULL, PIPE, and TFTP), or
- an open file that resides on a volume managed by pHILE+ file system component, or
- an open connection oriented socket managed by the pNA+ networking component, or
- an application defined module that can perform I/O operations as per the pSEUDO driver specification

The pSEUDO driver provides up to 32K channels per task, the I/O to which can be re-directed to a different file, device or socket, individually for each task in the system. These are also referenced to as task private channels. It also provides up to 32K system wide channels to which the I/O can be redirected to a different file, device, or socket. The system wide channels are shared by all the tasks in the system and hence they are known as task shared channels. One of the task shared channels is used as a redirectable system console (`///dev/pscnsl`) by pSOSystem. Also, three task private channels are used as redirectable standard input (`///dev/stdin`), output (`///dev/stdout`), and error-output (`///dev/stderr`) channels by pSOSystem. Other channels, if configured, are available for application specific use.

The redirection is controlled through device control operations provided by the pSEUDO driver, and can be performed any time. At first, the redirection facility may seem similar to what can be achieved by the `freopen()` pREPC+ library service call. However, there are distinct differences between the two facilities, and in some situations the functionality provided by the pSEUDO driver cannot be achieved by using `freopen()`. The major differences are listed below:

- With pSEUDO driver the redirection can be done either on a per-task-basis, or for all the tasks in the system. The `freopen()` based redirection works only on a per task basis.
- With `freopen()`, the redirection has to be done programmatically by the task for which the redirection has to be performed. It is not possible to perform I/O redirection using `freopen()` for any arbitrary task without its cooperation. With the pSEUDO driver, redirection for a task can be performed, even for task private channels of a different task, by another task in the system, and at any time during the execution of a task.
- The `freopen()` call does not work with sockets or open files, but pSEUDO driver does.
- The pSEUDO driver supports inheritance of task private redirected channels of a task by the children tasks that it creates. Since inheritance is not defined for open files and sockets by standard pSOSystem components, it is hard to achieve the POSIX-like semantics of inheriting the descriptors of the parent processes by a child process, otherwise.

When redirecting I/O to another device driver, the pSEUDO driver acts as a pass-through driver; all device I/O requests are passed through to the target driver or low-level driver with minimal intrusion. The target driver performs the actual I/O operation.

When redirecting I/O to a file managed by pHILE+, the pSEUDO driver performs the I/O using the `read_f` and `write_f` services of pHILE+. For task private channels the I/O is performed directly in the context of the task that initiated the I/O operation.

When redirecting I/O to a socket managed by pNA+, the pSEUDO driver performs the I/O using the `recv` and `send` services of pNA+. For task private channels the I/O is performed directly in the context of the task that initiated the I/O operation.

When redirecting I/O to an application defined module, the pSEUDO driver passes all I/O requests to the custom I/O functions for that module, which services the actual requested I/O operations.

Figure 4-1 illustrates the implementation of the pSEUDO driver and other modules in a pSOSystem environment.

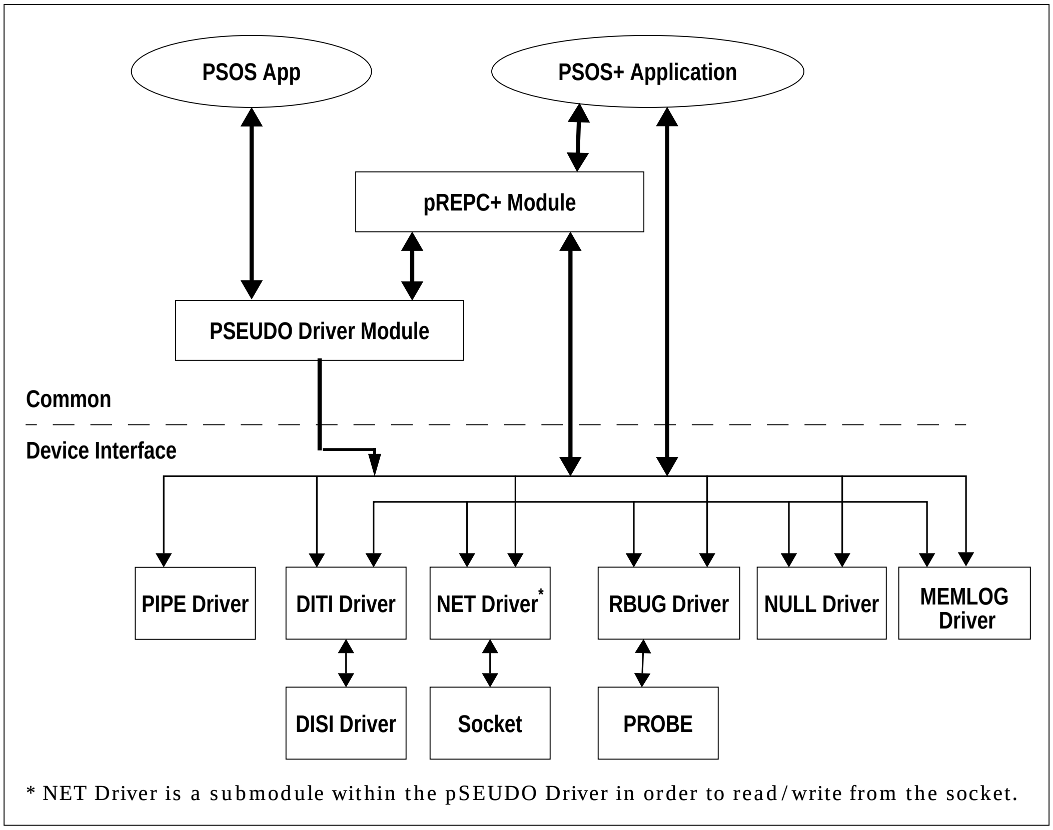


FIGURE 4-1 pSEUDO Driver Implementation in pSOSystem

Operation

Device Initialization

The application-callable initialization routine is simply a NULL function. The device initialization of pSEUDO driver occurs automatically, early during system initialization process, even before the ROOT task has been created. This is done using the driver system startup callout family.

Device Open

This is a null function and always returns zero.

Device Close

This is a null function and always returns zero.

Device Read

The read operation follows the same interface as defined for the read operation for a standard pSOSystem character device driver interface. Refer to section *Device Read Entry* on page 4-4 for details. The semantics of the read operation is the same as the semantics of a read operation from the underlying driver, file, socket connection, or the application defined module from which the I/O is directed.

Device Write

The write operation follows the same interface as defined for the read operation for a standard pSOSystem character device driver interface. Refer to section *Device Write Entry* on page 4-5 for details. The semantics of the write operation is the same as the semantics of a write operation to the underlying driver, file, socket connection, or the application defined module to which the I/O is directed.

Device Control

The pSEUDO driver provides device control functions for performing I/O redirection, and to obtain the current redirection status of a channel. If the I/O control request passed to the pSEUDO driver is not targeted to it, and I/O on that channel has been redirected to another device driver or custom application defined module, the request is passed through to the underlying device driver or module for processing.

As explained in section *Device I/O Control Entry* on page 4-5, the device control function is passed an IOPB structure, whose first field (`fcode`) identifies the I/O control operation to be performed, and the remaining fields (`args[]`), identify the other parameters specific to the I/O control operation being performed. In each case the device minor number specifies the channel number on which to operate. Table 4-3 on page 4-21 lists the various device control operations supported by the pSEUDO driver.

TABLE 4-3 Device Control Operations Supported by the pSEUDO Driver

Function Code	Description
MAPIO_SET_REDIRECTION	Set the I/O redirection for a channel.
MAPIO_GET_REDIRECTION	Get the I/O redirection status of a channel.

MAPIO_SET_REDIRECTION: Setting the I/O Redirection for a Channel

The function code for redirecting the I/O is MAPIO_SET_REDIRECTION. It takes as the first parameter (arg[0]), an unsigned long value that specifies whether the redirection is to be performed to a device, a file, a socket, or an application defined module. The various values for arg[0] and the other required parameters that go with them are as follows:

MAPIO_DEVICE	Specifies that the I/O on this channel be redirected to the device whose device number is specified by arg[1].
MAPIO_FILE	Specifies that the I/O on this channel be redirected to an open file whose file handle is specified by arg[1].
MAPIO_SOCKET	Specifies that I/O on this channel be redirected to an open connection oriented socket whose socket descriptor is specified by arg[1].
MAPIO_CUSTOM	Specifies that I/O on this channel be redirected to a custom application defined module whose module ID is specified by arg[1].

MAPIO_GET_REDIRECTION: Obtaining the Redirection Status of a Channel

The function code for registering an application defined custom module is MAPIO_GET_REDIRECTION, and it takes as the first parameter (arg[0]) a pointer to a PssMapioRedirectStatus structure, defined in the include file drv_intf.h and reproduced below, through which the current redirection status of a channel is returned.

```
typedef struct {
  unsigned long      type;
  unsigned long      target;
  unsigned long      useCount;
} PssMapioRedirectStatus;
```

The `useCount` field specifies the current value of in use counter associated with the channel. The `type` field can have a value of:

- `MAPIO_DEVICE`
- `MAPIO_FILE`
- `MAPIO_SOCKET`
- `MAPIO_CUSTOM`

The `type` field value depends on whether the I/O is currently redirected to a device, file, socket, or application defined module, respectively. Accordingly, the `target` field is set to the device number, file handle, socket descriptor, or the module ID of an application- specific module to which the I/O on this channel is currently redirected.

If the I/O is not currently redirected, the type field is set to `MAPIO_NONE`, and the values in other fields are undefined.

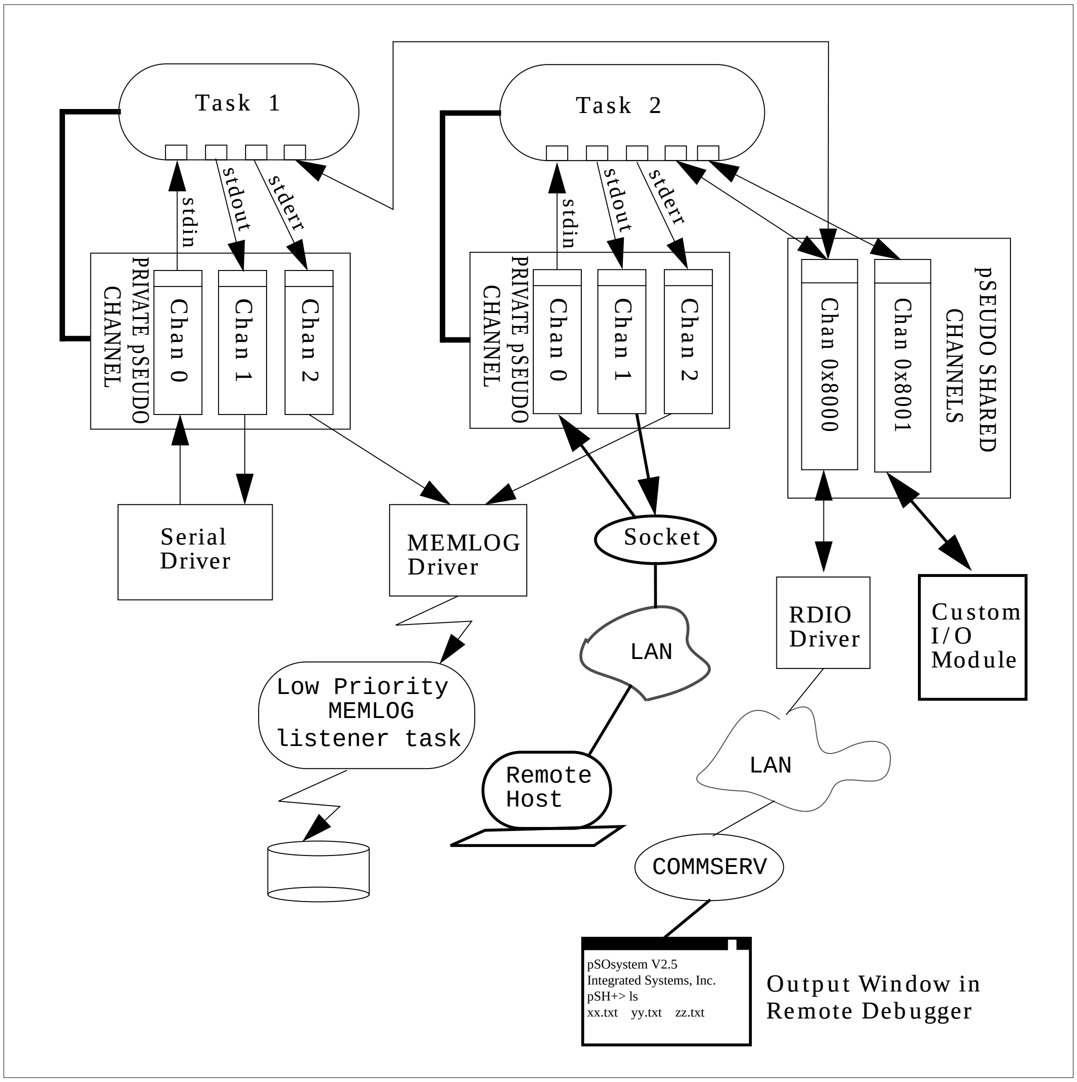
Reference Information

Supported Interfaces		Standard character I/O interface
Names registered in pSOS+ DNT		`stdin`, `stdout`, `stderr`, `psconsole`
pRNC Names		`///dev/stdin`, `///dev/stdout`, `///dev/stderr`, `///dev/psconsole`,
Minor number encoding	0	Default task private `stdin` channel
	1	Default task private `stdout` channel
	2	Default task private `stderr` channel
	3 through 0x7FFF	Other task private channels
	0x8000	Default system console shared by all the tasks in the system
	0x8001 through 0xFFFF	Other task shared channels
Driver specific parameters accepted by device open entry	None	
Associated `sys_conf.h` parameters	`SC_DEV_PSCONSOLE`	The major number of the device driver, or 0 if not to be configured
	`DEV_PSCONSOLE`	The major number of device driver shifted left by 16 bits; can be used to create a valid pSOS+ device number
	`DEV_STDIN`	`stdin` device major minimum number encoding
	`DEV_STDOUT`	`stdout` device major minimum number encoding

Associated sys_conf.h parameters (continued)	DEV_STDERR	stderr device major minimum number encoding
	SC_PSCNSL_SHARED_CHAN	Maximum number of task-shared multiplexing channels that can be active simultaneously
	SC_PSCNSL_PRIVATE_CHAN	Maximum number of task-private multiplexing channels that can be active simultaneously
	SC_PSCNSL_DEFAULT_DEV	The default device driver to which all the channels are re-directed at the time of system startup (DEV_SERIAL default)
	SC_PSCNSL_MAX_CUSTOM	The maximum number of custom I/O modules that can be registered with the MAPIO driver
pSOSystem Components required		pSOS+, pREPC+ and optionally: pNA+ and pHILE+
Resource requirement	Memory	((40 * SC_PSCNSL_PRIVATE_C_CHAN * KC_NTASK) + (40 * SC_PSCNSL_SHARED_CHAN)) bytes
	TSD Control Blocks	One system-wide
	Callout Control Blocks	One system-wide

Application Examples

Figure 4-2 on page 4-25 demonstrates a scenario where the pSEUDO driver is used for various types of I/O redirection supported by the pSEUDO driver. It is followed by the example code fragments required to set up the redirections demonstrated by the figure. For reasons of clarity, some of the obvious variable declarations and error checking have been omitted.



```
pSOsystem V2.5
Integrated Systems, Inc.
pSH+> ls
xx.txt    yy.txt    zz.txt
```

FIGURE 4-2 pSEUDO Redirection Example

In this example, there are two tasks both of which are performing all the device I/O through the pSEUDO driver. The first task has the `stdin` and `stdout` channels redirected to the serial driver, that happens to be the default redirection device for the pSEUDO driver (set using the `SC_PSCNSL_DEFAULT_DEV` parameter which is located in the `sys_conf.h` header file). The second task has redirected its `stdin` and stdout channels to a socket that is potentially communicating to a remote client simulating a simple remote terminal capability. Both the devices have redirected their stderr channels to the MEMLOG driver for the purpose of error logging. The messages logged to the MEMLOG are stored in a circular buffer for later examination (for details on MEMLOG driver, refer to section *MEMLOG* on page 4-29).

The I/O to the system console (pSEUDO channel), in this example, has been redirected by the RDIO driver to the output window in the source level debugger, running on a remote host (for details on how to do this, refer to section *RDIO* on page 4-40). Finally, the pSEUDO driver has been configured to have an additional task shared channel (0x8001), the I/O to which is being redirected to an application defined custom I/O module.

The various relevant sys_conf.h entries that correspond to this example are:

```
#define LC_STDIN     "///dev/stdin"
#define LC_STDOUT    "///dev/stdout"
#define LC_STDERR    "///dev/stderr"

#define SC_PSCNSL_SHARED_CHAN        2
#define SC_PSCNSL_PRIVATE_CHAN       3

#define SC_PSCNSL_DEFAULT_DEV        DEV_SERIAL
#define SC_PSCNSL_MAX_CUSTOM         1
```

The sample application program below demonstrates the use of the pSEUDO driver, to redirect I/O onto the memory log driver. Basically, the application opens the pSEUDO driver, and writes a string to the default console. Next, the application issues an IOCTL command (MAPIO_GET_REDIRECTION) to get the current mapping, and prints the major device number that is currently mapped. Finally, the application issues the MAPIO_SET_REDIRECTION to remap the device to a new device with major number 5 (say for example, The Memory Log driver is assigned a Major number 5), and writes a different string to the new device.

Sample Application Source Listing

```
/*********************************************************************/
/* root: Task to demonstrate the pSEUDO driver capabilities using  */
/*       the PSEUDO Driver Interface routines.                     */
/*                                                                  */
/*      INPUTS: None                                                */
/*                                                                  */
/*     RETURNS:                                                     */
/*     OUTPUTS:                                                     */
/*     NOTE(S):                                                     */
/*                                                                  */
/*********************************************************************/
void
root(void)
{
    ULONG rc, ioretval, tid;
    PssCharCtrlIOPB     ioctl_iopb;

                                  :
                                  :
                                  :
                                  :
                                  :

/*-------------------------------------------------------------*/
/* Set the STDIN to DEV_MEMLOG (Memory Log Driver)             */
/*-------------------------------------------------------------*/
iopb.fcode   = MAPIO_SET_REDIRECTION;
iopb.args[0] = MAPIO_DEVICE;
iopb.args[1] = DEV_MEMLOG;
iopb.args[2] = tid;

if ((rc = de_cntrl(DEV_STDIN, &ioctl_iopb, &dummy)) != 0) {
     PrintErrMessage(__FILE__, __LINE__, rc);
}

/*-------------------------------------------------------------*/
/* Set the STDOUT to DEV_MEMLOG (Memory Log Driver)            */
/*-------------------------------------------------------------*/
iopb.fcode   = MAPIO_SET_REDIRECTION;
iopb.args[0] = MAPIO_DEVICE;
iopb.args[1] = DEV_MEMLOG;
iopb.args[2] = tid;

if ((rc = de_cntrl(DEV_STDOUT, &ioctl_iopb, &dummy)) != 0) {
     PrintErrMessage(__FILE__, __LINE__, rc);
}

/*-------------------------------------------------------------*/
/* Set the STDERR to DEV_RDIO (Remote Debugger).               */
/*-------------------------------------------------------------*/
iopb.fcode   = MAPIO_SET_REDIRECTION;
iopb.args[0] = MAPIO_DEVICE;
iopb.args[1] = DEV_RDIO;
iopb.args[2] = tid;

if ((rc = de_cntrl(DEV_STDERR, &ioctl_iopb, &dummy)) != 0) {
     PrintErrMessage(__FILE__, __LINE__, rc);
}

/*-------------------------------------------------------------*/
/* Loop to write and read from Memlog driver and later output  */
/* to stderr                                                   */
/*-------------------------------------------------------------*/
while (1) {

    fputs(WriteBuf, stdout);
    fgets(ReadBuf, sizeof(ReadBuf), stdin);
    fputs(ReadBuf, stderr);
    tm_wkafter(360);
   }
}
```

MEMLOG Driver for logging error and diagnostic messages

Overview

The MEMLOG driver provides a standard character device driver interface for logging system wide error and diagnostic messages. The logged messages are maintained in a circular buffer, the size of which is a configurable parameter. Message boundaries are maintained by delimiting each message by a NULL character. Any data written by a single call to the write routine of the MEMLOG driver is treated as a message, and the driver ensures that every message is written in an atomic manner such that two concurrent write requests do not result in garbled data. The driver allows messages to contain NULL characters, though NULL characters embedded in the message would cause new message boundaries to be defined.

The read from MEMLOG driver always returns a NULL terminated message if sufficient amount of buffer space is provided, else a partial message is returned. The reads from MEMLOG drivers do not block while waiting for the availability of data. If there are no messages available, a read call returns immediately, suggesting 0 bytes read.

Operation

Device Initialization

The device initialization of MEMLOG driver involves initializing the buffer parameter, and creating the mutex used for mutual exclusion. The initialization routine does not expect an IOPB, and ignores any IOPB passed to it.

Device Open

The device open function of MEMLOG driver simply returns a successful status.

Device Close

The device close function of MEMLOG driver simply returns a successful status.

Device Read

A read operation from the MEMLOG driver returns a NULL terminated message, if the requested byte count is greater than the size of the next available message in the buffer. Otherwise it returns a non-NULL terminated partial message of the size requested. The maximum number of bytes that can be read from the MEMLOG driver equals `SC_LOG_BUFSIZE`. The read operation is performed after obtaining a mutex lock such that two concurrent read operations, or two concurrent read and write operations do not interfere with each other.

Other than the behavior mentioned in the previous paragraph, the read operation follows the same interface and semantics as defined for the read operation for a standard pSOSystem character device driver. Refer to *Device Read Entry* on page 4-4 for details.

Device Write

A write operation to the MEMLOG driver results in writing the requested number of bytes up to `SC_LOG_BUFSIZE`-1, into the circular buffer maintained by the driver. Any unread messages are overwritten. For messages that are bigger than `SC_LOG_BUFSIZE`-1, the initial portion of the message that does not fit in the buffer is discarded. The write operation is done by obtaining a mutex lock on the circular buffer so that two concurrent write operations to the driver do not result in garbled data. A NULL character is placed in the circular buffer after writing the data, and if any task has been registered to be notified of the availability of a message, the registered set of events are sent to that task.

Other than the behavior mentioned in the previous paragraph, the write operation follows the same interface and semantics as defined for the write operation for a standard pSOSystem character device driver. Refer to section *Device Write Entry* on page 4-5 for details.

Device Control

The only device control function defined for the MEMLOG driver supports flushing the log buffer and attaching an external buffer for logging the data.

Reference Information

Supported Interfaces		Standard character I/O interface
Name registered in pSOS+ DNT		memlog
pRNC Name		///dev/memlog
Minor number encoding		None defined
Driver specific parameters accepted by device open entry		None
Associated sys_conf.h parameters	SC_DEV_MEMLOG	The major number of the device driver, or 0 if not to be configured
	DEV_MEMLOG	The major number of device driver shifted left by 16 bit; can be used to create a valid pSOS+ device number
	SC_LOG_BUFSIZE	Size of the circular buffer in number of bytes
pSOSystem Components required		pSOS+
Resource requirement	Mutexes	One system wide

Application Examples

The following example code fragment demonstrates how the stderr stream of a task can be redirected to the MEMLOG driver.

```
FILE    *fp;

fprintf(stderr, "This error message gets printed.\n");

fp = freopen("///dev/memlog", "w", stderr);

if (fp == NULL) {
    perror("freopen");
    exit(1);
}

fprintf(stderr, "This error message is logged!\n");
```

NULL

Driver that functions like a NULL data source and an infinite data sink

Overview

The NULL driver provides a standard character device driver interface that acts as a sink of infinite depth for write operations, and as a data source with no data to be read.

Operation

Device Initialization

The device initialization of NULL driver is not required. It simply returns a successful status.

Device Open

The device open function of NULL driver simply returns a successful status.

Device Close

The device close function of NULL driver simply returns a successful status.

Device Read

A read operation from the NULL driver is always successful and returns 0 as the number of bytes read. In other words, the device always acts as if the end of file condition has been encountered. The read operation follows the same interface as defined for the read operation for a standard pSOSystem character device driver. Refer to section *Device Read Entry* on page 4-4 for details.

Device Write

A write operation to the NULL driver is always successful. All the data passed to the driver is simply discarded and the number of bytes passed to the driver is returned back as the number of bytes written. The write operation follows the same interface as defined for the write operation for a standard pSOSystem character device driver. Refer to section *Device Write Entry* on page 4-5 for details.

Device Control

There are no device control functions defined for NULL driver. The device control command and any data passed to the driver is ignored and a successful status is returned.

Reference Information

Supported Interfaces		Standard character I/O interface
Name registered in pSOS+ DNT		`null`
pRNC Name		`///dev/null`
Minor number encoding		None defined
Driver specific parameters accepted by device open entry		None
Associated `sys_conf.h` parameters	`SC_DEV_NULL`	The major number of the device driver, or 0 if not to be configured
	`DEV_NULL`	The major number of device driver shifted left by 16 bit; can be used to create a valid pSOS+ device number
pSOSystem Components required		None
Resource requirement		None

Application Examples

The following example code fragment demonstrates how the `stderr` stream of a task can be redirected to the NULL driver such that any error messages produced by the program are discarded.

```
FILE    *fp;

fprintf(stderr, "This error message gets printed!\n");

if ((fp = freopen("///dev/null", "w", stderr)) == NULL) {
    perror("freopen");
    exit(1);
}

fprintf(stderr, "This error message goes to the kitchen
sink!\n");
```

PIPE

Driver for inter-task data communication

Overview

The PIPE driver provides a standard character device driver interface for creating a simplex channel for exchanging data between tasks. Functionally, it is similar to the pipe facility provided by POSIX. The data stream obtained from a PIPE can be manipulated using standard ANSI-C stdio facilities.

The driver supports multiple channels that can operate simultaneously. The number of channels can be configured using the SC_MAX_PIPE_CHAN parameter located in the sys_conf.h header file. The number of channels that can be simultaneously active may, however, be limited by the availability of other resources mentioned in the Reference Information section located on page 4-38.

There could be multiple readers or writers reading or writing to a single pipe. The driver guarantees that for a given channel:

- Readers and writers get access based on their priority
- The read operation is performed atomically, if there is enough data in the PIPE buffer such that the operation can be completed without blocking to wait for a writer to write data to the PIPE
- The write operation is performed atomically, if there is enough space in the PIPE buffer such that the operation can be completed without blocking to wait for a reader to empty the PIPE
- In situations where all the requests can be completed without waiting for readers to flush the PIPE or writers to fill the PIPE, no tasks are blocked for an unbounded time.

The driver supports both blocking and non-blocking I/O operations for both reading from the PIPE and writing to the PIPE.

Operation

Device Initialization

The device initialization of PIPE driver involves allocating memory for bookkeeping information, and initializing the mutex used for mutual exclusion. It is not necessary to explicitly initialize the driver. The driver is automatically initialized at the time of processing the first request to open the device. The initialization routine does not expect an IOPB, and ignores any IOPB passed to it.

Device Open

The PIPE driver opens the read or write end of one of the channels, specified by either the device minor number passed by the caller or, optionally, the driver specific parameters passed via the standard character driver open IOPB, as defined in section, *Device Open Entry* on page 4-3.

- The flags field of the IOPB is 0 if the channel is being opened for blocking I/O, or it has a value of `O_NONBLOCK` (defined in the header file, `fcntl.h`) if the channel is to be opened for non-blocking I/O. Note that non-blocking status for a channel is maintained separately for the read and write ends of the pipe, and the end of the pipe being opened determines the end for which the blocking or non-blocking mode is set. Also, note that if multiple readers (or writers) open a channel, then even if a single reader (or writer) changes the mode for the read (or write) end of the pipe to non-blocking, that mode becomes effective for the rest of the readers (or writers) as well.
- The `params` field of the IOPB points to a string of comma separated name value pairs that is parsed by the driver open routine to obtain the size of buffer to be associated with the channel being opened, the channel number of the PIPE driver to engage for the purpose of data transfer, and whether the read or write end of the driver is being opened. The `flags` field of the IOPB is ignored. Table 4-4 describes the format of various parameters as expected by the PIPE driver open routine.

TABLE 4-4 PIPE Driver Parameters

Parameter	Syntax and Description
`channel`	A numeric value that specifies the channel number to be opened. This is an optional parameter. If the `channel` parameter is omitted then the minor number of the device passed to the `de_open` call specifies the channel to be opened, otherwise the device minor number passed to the `de_open` call is ignored. ■ If a positive or zero value is specified, it is taken as the channel to be opened. ■ If a value of -1 is specified, the driver allocates an available channel and passes the device major minor number corresponding to the allocated channel back to the caller in `cloneDev` field of the IOPB.

TABLE 4-4 PIPE Driver Parameters (Continued)

Parameter	Syntax and Description
`mode`	This is an optional parameter that takes one of the two string values, `read` or `write`, that respectively specify whether the read or write end of the PIPE driver is being opened. If this parameter is omitted, then the most significant bit of the minor number determines which end of the PIPE is opened. Read contains the most significant bit 0. Write is set to 1.
`bufsize`	This is an optional parameter and it specifies the size of buffer (in bytes) that shall be allocated and associated with this channel for the purpose of buffering the data. If this parameter is omitted, the buffer size is defaulted to 1024 bytes. Also, this parameter is honored only for the first open request for an inactive channel.

Device Close

The PIPE driver close function closes the channel and end (read/write) of the PIPE as specified by the device minor number passed by the caller. If there are no more tasks that have this channel open, the resources like mutex, condition variables, and memory associated with the channel are freed.

Device Read

The behavior of the PIPE for read operations (which is also consistent with the behavior defined for POSIX pipes) is as follows:

- If an attempt is made to read from an empty pipe, then
  - If no writers have opened the PIPE for writing, a value of zero is returned as the number of bytes read indicating end of file
  - If one or more tasks have opened the pipe for writing and the `O_NONBLOCK` flag for the read end is set, then the read operation returns an error `PIPE_EMPTY`
  - If one or more tasks have opened the pipe for writing and the `O_NONBLOCK` flag for the read end is clear, then the read operation blocks until the request can be satisfied, or until all the writers close their end of the PIPE

The read operation follows the same interface as defined for the read operation for a standard pSOSystem character device driver interface. Refer to section *Device Read Entry* on page 4-4 for details.

Device Write

The behavior of the PIPE for write operations (which is also consistent with the behavior defined for POSIX pipes) is as follows:

- If the O_NONBLOCK flag for the write end of the PIPE is clear (for example. the PIPE is operating in the blocking mode), a write request always writes all the data, and if the amount of data being written is more than what the PIPE can buffer, the writer may block waiting for some reader to flush the data from the buffer.
- If the O_NONBLOCK flag for the write end of the PIPE is set then the behavior depends on whether the amount of data being written is more or less than the size of buffer associated with the PIPE:
  - If the write request is less than or equal to the size of the buffer associated with this PIPE then:
    - i. If there is sufficient space to write all the data to the buffer, it is written
    - ii. Else no data is transferred and error PIPE_FULL is returned to the caller
  - If the write request is more than the size of the buffer associated with this PIPE then:
    - i. If at least one byte of data can be written to the PIPE, it is written
    - ii. Else if no data can be written, error PIPE_FULL is returned to the caller

The write operation follows the same interface as defined for the write operation for a standard pSOSystem character device driver interface. Refer to section *Device Write Entry* on page 4-5 for details

Device Control

There are no device control functions defined for PIPE driver.

Reference Information

Supported Interfaces		Standard character I/O interface
Name registered in pSOS+ DNT		`pipe`
pRNC Name		`///dev/pipe?<params>`
Minor number encoding	0 to 0x7FFF	Channel numbers of the read end of the pipe
	0x8000 to 0xFFFF	Channel numbers of the corresponding write end of the pipe
Driver specific parameters accepted by device open entry	`channel`	The channel number to open
	`bufsize`	The size (in bytes) of buffer to allocate for buffering the data in the channel. (note: default is 1024 bytes)
Associated `sys_conf.h` parameters	`SC_DEV_PIPE`	The major number of the device driver, or 0 if not to be configured
	`DEV_PIPE`	The major number of device driver shifted left by 16 bits; can be used to create a valid pSOS+ device number
	`SC_MAX_PIPE_CHAN`	Maximum number of channels that can be simultaneously activated
pSOSystem Components required		pSOS+, pREPC+
Resource requirement	Mutexes	One system wide and one per active channel
	Condition Variable	One per active channel
	Memory	56 bytes per channel, plus the size of data buffer associated with the channel

Application Examples

The following example code fragment demonstrates how the de_open call can be used to open the write end of the PIPE. The channel parameter is passed as -1 to request that the driver allocate any available channel.

```
PssCharOpenIOPB   open_iopb;
PssCharRdwrIOPB   rdwr_iopb;
Unsigned long     rc, retval;
char              buf[512];

open_iopb.params = "channel=-1,bufsize=512,mode=write";

rc = de_open(DEV_PIPE, &open_iopb, &retval);
if (rc != 0)
    k_fatal(rc, 0);

rdwr_iopb.count = 512;
rdwr_iopb.address = buf;

rc = de_write(open_iopb.cloneDev, &rdwr_iopb, &retval);
```

The following example code fragment demonstrates how PIPE driver can be used with the standard ANSI stdio functions provided by pREPC+. This example demonstrates how the PIPE driver can be requested to open a specific channel (3 in this case).

```
FILE            *fp;
char              buf[512];

fp = fopen("///dev/pipe?channel=3,mode=read", "r" );

if (fp == NULL) {
    perror("Open");
    exit(1);
}

while(!eof(fp)) {
     count = fread(buf, 512, 1, fp);
     fwrite(buf, count, 1, stdout);
}
```

RDIO

Driver that performs I/O to or from a window in the remote debugger

Overview

The RDIO driver provides a standard character device driver interface for interacting with the target system from a window in the remote debugger over the communication link between the pROBE+ debugger running on the target, and the pRISM+ Communication Server running on the remote host system. It is especially useful for redirecting the `stdin`, `stdout` and `stderr` streams of one or more tasks to a window in remote debugger when either:

- The target system does not have an asynchronous serial port to which a terminal could be connected for interaction, or
- There is only one asynchronous serial port available on the target and it is being used by pROBE+ for the purpose of remote debugging, or
- The target is remotely located, and it is not possible to attach a terminal (for the purpose of interacting with the target system) to one of the asynchronous serial ports available on the target due to the distance limitations posed by the asynchronous serial protocol.

Operation

Device Initialization

The device initialization of RDIO driver is not required. It simply returns a successful status.

Device Open

The device open function of RDIO driver simply returns a successful status.

Device Close

The device close function of RDIO driver simply returns a successful status.

Device Read

A read request to RDIO device is converted into a data input request to the remote debugger. The remote debugger, typically, designates a separate window for interacting with the target. Any input typed in this window is what is passed back in response to a data input request from the target. Data obtained from the remote debugger is then returned to the task performing the read operation on RDIO driver; there is no data buffering done by pROBE+ or the RDIO driver. The amount of data returned from a read operation depends largely on how the remote debugger implements the interactive I/O. Most remote debuggers usually implement line buffered I/O. Therefore, the amount of data returned by the remote debugger may not be equal to the amount of data requested, though it may not signify there is no more data to be read.

Other than the behavior mentioned in the previous paragraph, the read operation follows the same interface and semantics as defined for the read operation for a standard pSOSystem character device driver. Refer to section *Device Read* on page 4-14 for details.

Device Write

A write operation to the RDIO driver is converted into a data output request to the remote debugger. The remote debugger, typically, displays the data in a separate window designated for interacting with the target. The write operation follows the same interface and semantics as defined for the write operation for a standard pSOSystem character device driver. Refer to section *Device Write* on page 4-14 for details.

Device Control

There are no device control functions defined for RDIO driver. The device control command and any data passed to the driver is ignored and a successful status is returned.

Reference Information

Supported Interfaces		Standard character I/O interface
Name registered in pSOS+ DNT		`rdio`
pRNC Name		`///dev/rdio`
Minor number encoding		None defined
Driver specific parameters accepted by device open entry		None
Associated `sys_conf.h` parameters	`SC_DEV_RDIO`	The major number of the device driver, or 0 if not to be configured
	`DEV_RDIO`	The major number of device driver shifted left by 16 bits; can be used to create a valid pSOS+ device number
pSOSystem Components required		pSOS+, pROBE+, and optionally pNA+ or pNET+
Resource requirement		None

Application Examples

The following example code fragment demonstrates how the `stdin` and `stdout` streams of a task can be redirected to the RDIO driver.

```
FILE    *fp;
char      buf[80];

printf("Re-directing stdin and stdout to remote debugger!\n");

fp = freopen("///dev/rdio", "r", stdin);

if (fp == NULL) {
    perror("freopen stdin");
    exit(1);
}

fp = freopen("///dev/rdio", "w", stdout);
```

```
if (fp == NULL) {
    perror("freopen stdout");
    exit(1);
}

printf("This message appears in the remote debugger window\n");

/* Read a line from the remote debugger window and echo back */
fgets (buf, 80, stdin);
fputs (buf, stdout);
```

TFTP

Driver for transferring files from a TFTP server

Overview

The TFTP (Trivial File Transfer Protocol) driver provides a standard character device driver interface for obtaining files from a TFTP server. TFTP is a rather simple protocol used for transferring data between networked devices that use a datagram socket for data communication. The TFTP driver provides a standard interface so that any data stream obtained from a TFTP server can be manipulated using standard ANSI-C stdio facilities.

The driver supports multiple channels that can operate simultaneously. The number of channels can be configured using the `SC_MAX_TFTP_CHAN` parameter which is located in `sys_conf.h` header file. The number of channels that can be simultaneously active may, however, be limited by the availability of other resources.

In order to provide maximum concurrency between the activities of data transfer over network and data processing by application, the TFTP driver creates a separate task that performs the job of data transfer over the network using the TFTP protocol and handles the protocol specific details like error detection, reordering datagrams, and retrying to obtain lost packets. This task communicates with the application task by way of a pSOS+ message queue. The driver sets aside 10 buffers of 516 bytes each for data transfer between the server task and the application.

NOTE: Despite the buffering provided by the driver, if a large amount of data is being transferred that the application is unable to consume in a timely manner, the TFTP server may timeout for lack of data request and may drop the connection, resulting in an error returned by the device read operation to the application. However, this is seldom a problem since the server timeout value is of the order of several seconds.

Operation

Device Initialization

The device initialization of the TFTP driver involves allocating memory for bookkeeping information, and initializing the mutexes used for mutual exclusion. It is not necessary to explicitly initialize the driver. The driver is automatically initialized at the time of processing the first request to open the device. The initialization routine does not expect an IOPB, and ignores any IOPB passed to it.

Device Open

The TFTP driver open function establishes connection with the TFTP server, specified by the driver specific parameters passed by way of the standard character driver open IOPB, as defined in section *Device Open* on page 4-20. The `params` field of the IOPB points to a string of comma separated name value pairs that is parsed by the driver open routine to obtain the IP address of the TFTP server, the pathname that identifies the file to be transferred, and optionally the channel number of the TFTP driver to engage for the purpose of data transfer. The `flags` field of the IOPB is ignored. Table 4-5 describes the format of various parameters as expected by the TFTP driver open routine.

TABLE 4-5 TFTP Driver Parameters

Parameter	Syntax and Description
`channel`	A numeric value that specifies the channel number to be opened. This is an optional parameter. If the `channel` parameter is omitted then the minor number of the device passed to the `de_open` call specifies the channel to be opened, otherwise the device minor number passed to the `de_open` call is ignored. ■ If a positive or zero value is specified, it is taken as the channel to be opened. ■ If a negative value is specified, the driver allocates an available channel and passes the device major minor number corresponding to the allocated channel back to the caller in the `cloneDev` field of the IOPB.
`ipaddr`	This is a required parameter that specifies the IP address of the TFTP server. It can be a decimal, octal, or hexadecimal number. It can also be specified in the standard dot notation used to specify IP addresses.
`pathname`	This is a required parameter and it specifies the pathname of the remote file to be accessed.

Device Close

The TFTP driver close function terminates the connection with the TFTP server, and deletes the server task created at the time of device open.

Device Read

The read operation follows the same interface and semantics as defined for the read operation for a standard pSOSystem character device driver interface. Refer to section *Device Read* on page 4-20 for details.

Device Write

The write operation is not defined for TFTP driver.

Device Control

There are no device control functions defined for TFTP driver.

Reference Information

Supported Interfaces		Standard character I/O interface
Name registered in pSOS+ DNT		`tftp`
pRNC Name		`///dev/tftp?<params>`
Minor number encoding		0 through 0xFFFF - channel number
Driver specific parameters accepted by device open entry	channel	The channel number to open
	ipaddr	The IP address of the TFTP server
	pathname	The complete path name of the file to be transferred from the TFTP server
Associated `sys_conf.h` parameters	SC_DEV_TFTP	The major number of the device driver, or 0 if not to be configured
	DEV_TFTP	The major number of device driver shifted left by 16 bits; can be used to create a valid pSOS+ device number
	SC_MAX_TFTP_CHAN	Maximum number of channels that can be simultaneously activated
pSOSystem Components required		pSOS+, pREPC+, pNA+/pNET+

Resource requirement	Mutexes	One system wide and one per active channel
	Task	One per active channel
	Queue	Two per active channel
	Memory	56 bytes per channel, and 5160 bytes per active channel
	Socket	One per active channel

Application Examples

The following example code fragment demonstrates how de_open call can be used to open a connection with the TFTP server at IP address 192.103.54.36 to transfer a file named app.hex from the default directory of the TFTP server. The channel parameter is passed as -1 to request that the driver allocate any available channel.

```
PssCharOpenIOPB   open_iopb;
PssCharRdwrIOPB   rdwr_iopb;
Unsigned long     rc, retval;
char              buf[0x200];

open_iopb.params =
    "channel=-1,ipaddr=192.103.54.36,pathname=app.hex";

rc = de_open(DEV_TFTP, &open_iopb, &retval);
if (rc != 0)
    k_fatal(rc, 0);

rdwr_iopb.count = 0x200;
rdwr_iopb.address = buf;

rc = de_read(open_iopb.cloneDev, &rdwr_iopb, &retval);
```

The following example code fragment demonstrates how TFTP driver can be used with the standard ANSI stdio functions provided by pREPC+. This example demonstrates how the TFTP driver can be requested to open a specific channel 3 in this case).

```
FILE            *fp;
char              buf[0x200];

fp = fopen(
 "///dev/tftp?channel=3,ipaddr=204.71.177.159,pathname=/", "r" );

if (fp == NULL) {
    perror("Open");
    exit(1);
}

while(!eof(fp)) {
     count = fread(buf, 0x200, 1, fp);
     fwrite(buf, count, 1, stdout);
}
```

DITI (Device Independent Terminal Interface)

Overview

The Device Independent Terminal Interface (DITI) is the interface between a task and the terminal driver through the I/O Jump Table. The terminal driver uses the Device Independent Serial Interface (DISI) to complete the device dependent part of the driver.

A complete understanding of this interface is required only if the developer wants to modify the behavior of the terminal driver or obtain the details of its behavior. If the application is going to use the terminal driver only to read or write to the device, then the developer can accomplish this by using pREPC+ calls such as `printf` and `scanf` found in the C library. In this case, the application can use the AUTOINIT feature to make pSOS+ initialize the terminal driver. This allows the application to start using the driver directly for reading and writing.

Operation

The DITI is called by way of the pSOS+ I/O Jump Table. It can be used for serial communications to devices such as terminals, printers, and other computers. The DITI supplies an interface to:

- Setup a serial channel to the requirements of the user. For example, baud rate, time-outs, character size, and flow control,
- Send and receive data,
- Be used by pREPC to transfer data in asynchronous mode.

The DITI separates the task from the terminal driver and is the standard interface to the terminal driver. Figure 4-3 on page 4-51 shows the DITI interface and how it fits into the pSOSystem.

The DITI defines six functions, shown in Table 4-6 on page 4-50, that correspond to the pSOS+ I/O driver table functions.

TABLE 4-6 DITI Functions

Entry for DITI	DITI Call	Description
`de_init`	`TermInit`	Initialize the console driver.
`de_open`	`TermOpen`	Open a channel.
`de_read`	`TermRead`	Read from a channel.
`de_write`	`TermWrite`	Write to a channel.
`de_cntrl`	`TermIoctl`	Perform a control operation on a channel.
`de_close`	`TermClose`	Close a channel.

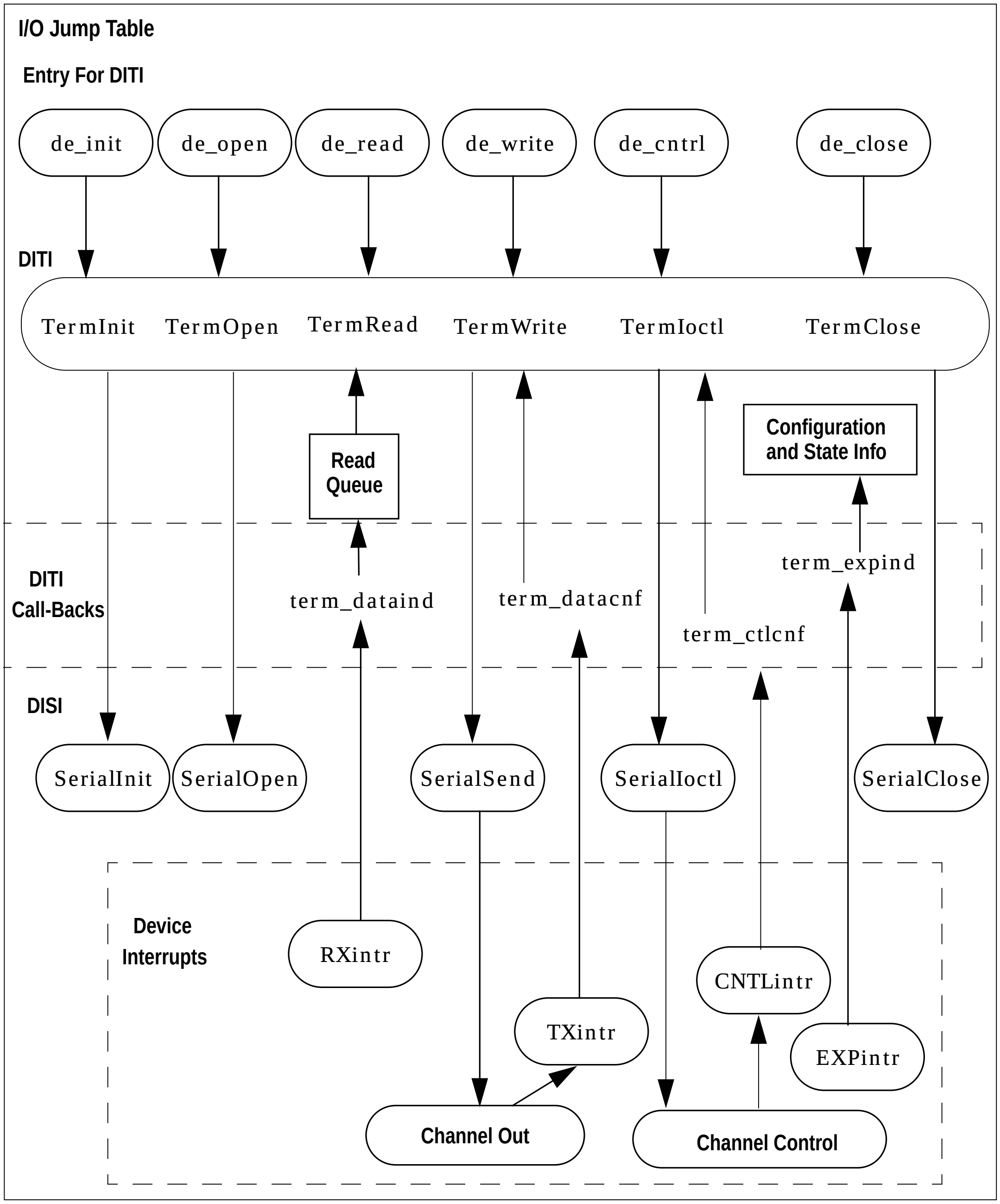


FIGURE 4-3 DITI Interface

DITI Features

When a channel is opened it causes the task to wait until a connection is established. A terminal associated with the opened channel ordinarily operates in full-duplex mode (input and output can occur simultaneously). Input is stored in the input buffers until the input buffers become full. Note that when the input buffer is full, without warning to the application, the additional characters are deleted.

Canonical mode input processing

Terminal input is processed and assembled in units of lines. A line is delimited by a newline (ASCII LF) character, an end of file (ASCII EOT) character, or an end of line (ASCII CR) character. A newline character must be inputted before characters are returned by the read call regardless of the number of characters specified by a requesting program. Therefore, a program attempting a read is suspended until the line delimiter, NL, has been inputted to the read call.

It is not necessary, however, for the program to retrieve a whole line at once from the read call; one or more characters may be requested by the program in each read without loss of information. The read will return from its input buffer the number of characters requested by the program. If the input buffer already contains the number of characters requested by the program, the read returns immediately. If fewer characters are contained in the input buffer than what is requested by the program, the read fetches more characters from the channel.

During input, erase processing is normally done. The erase function (by default, the character DEL) deletes the last character inputted. The erase function does not remove characters beyond the beginning of the line. The ASCII character assigned to the **ERASE** special character (see, *Special Characters* on page 4-55) can be configured.

Non-Canonical mode input processing

In non-canonical mode input processing, input characters are not assembled into lines and erase processing does not occur. Two parameters, MinChar and MaxTime are used in non-canonical mode processing (CANON bit not set in flags) to determine how to process the characters received.

MinChar represents the minimum number of characters that are received to complete the read and return to the calling program. MaxTime is a timer of one-tenth second granularity that is used to time-out burst and short term data transmissions. The four possible values for MinChar and MaxTime and their interactions are described below.

Case A: MinChar > 0 and MaxTime > 0

In this case MaxTime serves as an inter-character timer and is activated after the first character is received and is reset after each subsequent character input. The interaction between MinChar and MaxTime is as follows:

- As soon as one character is received, the inter-character timer is started.
- If MinChar characters are received before the inter-character timer expires, the read is satisfied.
- If the timer expires before MinChar characters are received, the characters received to that point are returned to the user.

NOTE: If MaxTime expires, at least one character is returned because the timer is enabled only if a character was received. In this case where MinChar > 0 and MaxTime > 0 the read blocks until the MinChar and MaxTime mechanisms are activated by the receipt of the first character.

Case B: MinChar > 0 and MaxTime = 0

In this case only MinChar is significant because MaxTime is set to zero. A pending read is blocked until MinChar characters are received.

NOTE: A program that uses this case to read record based terminal I/O may block indefinitely in the read operation.

Case C: MinChar = 0 and MaxTime > 0

In this case MaxTime no longer represents an inter-character timer but serves as a read timer because MinChar is set to zero. It is activated as soon as a read call is made. A read is satisfied on input of a single character or expiration of the read timer. If no character is received within .1 x MaxTime seconds after the read is initiated, the read returns with zero characters.

Case D: MinChar = 0 and MaxTime = 0

In this case, the return is immediate. The minimum of either the number of characters requested or the number of characters currently available is returned without waiting for more character input.

Comparison of the different cases (A, B, C, D) of MinChar, MaxTime interaction

Some points to note about MinChar and MaxTime:

- In the following explanations, note that the interactions of MinChar and MaxTime are not symmetric. For example, when MinChar > 0 and MaxTime = 0, MaxTime has no effect. However, in the opposite case, where MinChar = 0 and MaxTime > 0, both MinChar and MaxTime play a role (MinChar is satisfied with the receipt of a single character).
- Also note that in Case A, MaxTime represents an inter-character timer, whereas in case C, MaxTime represents a read timer.

These two points highlight the dual purpose of the MinChar/MaxTime feature. Cases A and B, where MinChar > 0, exists to handle burst mode activity (for example, file transfer programs), where a program would like to process at least MinChar characters at a time. In case A, the inter-character timer is activated by a user as a safety measure; in case B, the timer is turned off.

Cases C and D exist to handle single character, timed transfers. These cases are readily adaptable to screen based applications that need to know if a character is present in the input queue before refreshing the screen. In case C, the read is timed, whereas in case D, it is not.

Writing Characters

When one or more characters are written, after the processing of previously written characters, they are sent to the terminal. When echoing is enabled, input characters are echoed as they are typed. If a process produces characters faster than they can be typed, causing its output queue to exceed a specified limit, it is blocked. When the queue is drained down to its specified threshold, the program is resumed.

Special Characters

Certain characters cause special functions on input. These functions and their default character values are summarized as follows:

ERASE (DEL)	Special character on input and is recognized if the ICANON flag is set. It erases the previous character in the current line. The **ERASE** character does not erase beyond the start of a line as delimited by a **NL**, **EOF** or **EOL** character. If ICANON is set, the **ERASE** character is discarded when processed.
EOF (CTRL-D)	Special character on input and is recognized if the ICANON flag is set. When received, all the bytes waiting to be read are immediately passed to the process, without waiting for a newline, and the **EOF** is discarded. Thus, if there are no bytes waiting (that is, the **EOF** occurred at the beginning of a line), a byte count of zero is returned from the `read()`, representing an end of file indication. If ICANON is set, the **EOF** character is discarded when processed.
NL (ASCII LF)	Special character on input and is recognized if the ICANON flag is set. This is the normal end of line delimiter. This cannot be changed.
EOL	Special character on input and is recognized if the ICANON flag is set. It is an additional end of line delimiter.
CR (ASCII CR)	Special character on input and is recognized if the ICANON flag is set. When ICANON and IGNCR are set, this character is ignored. When ICANON and ICRNL are set and IGNCR is not set, this character is translated into an **NL** and has the same effect as an **NL** character. This cannot be changed.
STOP (CTRL-S or ASCII DC3)	Special character used to stop output temporarily. It is used with CRT terminals to prevent output from disappearing before it can be read. While output is suspended, **STOP** characters are ignored and not read.
START (CTRL-Q or ASCII DC1)	Special character used to restart output that has been stopped by a **STOP** function. If output is not suspended, **START** is ignored.

NOTE: **ERASE**, **EOL**, **STOP**, and **START** values can be changed.

Terminal Parameters

The parameters that control the behavior of devices and modules providing the termios interface are specified (and shown below) by the `termios` structure defined in `termios.h`:

```
c_iflag;            /* input modes */
c_oflag;            /* output modes */
c_cflag;            /* control modes */
c_lflag;            /* local modes */
c_cc[NCCS];         /* control chars */
```

The special control characters are defined by the array `c_cc`. The size of the control character array is specified by NCCS which is also defined in `termios.h`.

Input Modes

Table 4-7 describes the basic terminal input control modes. These modes are controlled by the field settings of the `c_iflag` flag which are defined in `termios.h`.

TABLE 4-7 Input Mode Fields

Field	Description
BRKINT	Interrupt on break. If BRKINT is set, the break condition discards characters in the input and output queues. The next read of the channel returns with 0 characters read and the error code of TERM_BRKINT. If BRKINT is not set, a break condition is read as a single ASCII NULL character (\0).
INLCR	Map **NL** to **CR** on input. If INLCR is set, a received **NL** is translated into a **CR**.
IGNCR	Ignore **CR**. If IGNCR is set, a received **CR** is not read.
ICRNL	Map **CR** to **NL** on input. If ICRNL is set, a received **CR** is translated into a **NL**.

TABLE 4-7 Input Mode Fields (Continued)

Field	Description
IUCLC	Map upper case to lower case on input. If IUCLC is set, a received upper case character is translated into the corresponding lower case character.
IXON	Enable start/stop output and/or input control. If IXON is set, output and/or input control is enabled. A received **STOP** suspends output and/or input and a received **START** restarts output and/or input. **STOP** and **START** perform flow control functions. **NOTE:** The usage of IXON differs from the POSIX standard. As per POSIX, two flags IXON and IXOFF are used to control software flow control. If IXON is set, start/stop output control is enabled. If IXOFF is set, start/stop input control is enabled. In case of DITI, just one flag IXON is used to control both input and output flow control.

The initial input control value sets the following flags:

- BRKINT
- ICRNL
- IXON
- IMAXBEL.

Output Modes

Table 4-8 describes system treatment of output. Output treatment is controlled by the fields of the c_oflag which are defined in the file, termios.h.

TABLE 4-8 Output Mode Fields

Field	Description
OPOST	Post-process output. If OPOST is set, the output characters are post-processed as indicated by the remaining flags; otherwise, characters are sent without change.
OLCUC	Map lower case to upper case on output. If OLCUC is set, a lower case character is sent as the corresponding upper case character. This function is often used with IUCLC.

TABLE 4-8 Output Mode Fields (Continued)

Field	Description
ONLCR	Map **NL** to **CR-NL** on output. If ONLCR is set, the **NL** character is transmitted as the **CR-NL** character pair.
OCRNL	Map **CR** to **NL** on output. If OCRNL is set, the **CR** character is transmitted as the **NL** character.
ONOCR	No **CR** output at column 0. If ONOCR is set, no **CR** character is transmitted when at column 0 (first position).
ONLRET	**NL** performs **CR** function. If ONLRET is set, the **NL** character is assumed to do the **CR** function and the column pointer is set to 0. Otherwise, the **NL** character is assumed to do just the line feed function; the column pointer remains unchanged. The column pointer is also set to 0 if the **CR** character is actually transmitted.

The initial output control value sets the following flags:

- `OPOST`
- `ONLCR`

Control Modes

Table 4-9 describes the fields used to specify the hardware flow control, (BAUD rate, character size, and stop bits, for example) of the terminal. Hardware control is determined by the settings of the `c_cflag` fields which are defined in the file, `termios.h`.

TABLE 4-9 Hardware Control Mode Fields

`c_cflag` Field	Description	
CBAUD	The CBAUD field controls the BAUD rate. The CBAUD bits specify the baud rate for both the input and output baud rates. For any particular hardware, impossible speed changes are ignored and the previous BAUD setting is used. Below are listed the possible BAUD rate settings.	
	Setting	BAUD Rate
	B50	50
	B75	75

TABLE 4-9 Hardware Control Mode Fields (Continued)

`c_cflag` Field	Description	
	B110	110
	B134	134
	B150	150
	B200	200
	B300	300
	B600	600
	B1200	1200
	B1800	1800
	B2400	2400
	B4800	4800
	B9600	9600
	B19200	19200
	B38400	38400
CSIZE	The CSIZE field controls the character size. The CSIZE bits specify the character size in bits for both transmission and reception. This size does not include the parity bit. The possible settings are listed below.	
	Setting	Character Size in Bits
	CS5	5
	CS6	6
	CS7	7
	CS8	8
CSTOPB	Send two stop bits, else one. If CSTOPB is set, two stop bits are used; otherwise, one stop bit is used. For example, at 110 baud, two stops bits are required.	

TABLE 4-9 Hardware Control Mode Fields (Continued)

`c_cflag` Field	Description
CREAD	Enable receiver. If CREAD is set, the receiver is enabled. Otherwise, no characters are received.
PARENB	Parity enable. If PARENB is set, parity generation and detection is enabled, and a parity bit is added to each character. See also, PARODD.
PARODD	Odd parity, else even. If parity is enabled, the PARODD flag specifies odd parity if set; otherwise, even parity is used. See also, PARENB.
CRTSCTS	Enable output and/or input hardware flow control. If CRTSCTS is set, output and/or input hardware flow control using the RTS and CTS signals are enabled.

The initial hardware control value is B9600, CS8 and CREAD.

Local Modes

The `c_lflag` field of the argument structure is used by the line discipline to control terminal functions. The basic line discipline controls are described in Table 4-10.

TABLE 4-10 Local Mode Fields

Field	Description
ICANON	Canonical input (erase processing). If ICANON is set, canonical processing is enabled. This enables the erase function, and the assembly of input characters into lines delimited by **NL**. If ICANON is not set, read requests are satisfied directly from the input queue. A read is not satisfied until at least MinChar characters have been received or the time-out value MaxTime has expired between characters. This allows fast bursts of input to be read while still allowing single character input. The time value represents tenths of seconds.
ECHO	Enable echo. If ECHO is set, input characters are echoed as received. If ECHO is not set, input characters are not echoed.
ECHONL	Echo **NL**. If ECHONL and ICANON are set, the **NL** character is echoed even if ECHO is not set.

The initial line-discipline control value is ICANON and ECHO.

Minimum and Timeout

The MinChar and MaxTime values are described in the section, *Canonical mode input processing* on page 4-52. The initial value of MinChar is 1, and the initial value of MaxTime is 0.

Modem Lines

The modem control lines supported by the hardware can be read, and the modem status lines supported by the hardware can be changed. Table 4-11 describes the modem control and status lines.

TABLE 4-11 Modem Control Lines

Control Line	Description
`TIOCM_DTR`	Data terminal ready
`TIOCM_RTS`	Request to send
`TIOCM_CTS`	Clear to send
`TIOCM_CD/TIOCM_CAR`	Carrier detect
`TIOCM_RI/TIOCM_RNG`	Ring
`TIOCM_DSR`	Data set ready

NOTE: Not all of these modem control lines are necessarily supported by any particular device.

DITI Functions

The following subsections describe the DITI functions that are entered into the I/O switch table during the pSOSystem initialization. These DITI functions should not be called directly by the application.

To access these DITI functions the application must use the following pSOS+ I/O driver table functions:

- `de_init()`
- `de_open()`
- `de_read()`

- `de_write()`
- `de_cntrl()`
- `de_close()`

TermInit

```
void TermInit (struct ioparms *parms);
```

This function initializes the terminal driver and is accessed by the application through the pSOS+ call, `de_init()`:

```
unsigned long de_init(unsigned long dev, void *iopb, void
*retval, void **data_area);
```

This function sets the default values for all channels. All channels are closed except for the pROBE+ host and console channels. This function should be called only once in the system before using the terminal driver.

The parameters of `de_init()` are mapped to the fields in `TermInit` input parameter `parms` by pSOS+. The `de_init()` parameters are:

`dev`	The parameter, `dev`, is mapped to `parms->in_dev` and specifies the major and minor device numbers, which are stored in the upper and lower 16 bits, respectively.
`iopb`	This parameter is not used by `TermInit`. This can be any uninitialized unsigned long pointer.
`retval`	The parameter, `retval`, is mapped to `parms->out_retval` and contains any additional information that the `TermInit` function needs to return to the application, else it is set to `0`.
`data_area`	This parameter is not used by the `TermInit` function. This can be any uninitialized void pointer.

Returns

The value returned by the `TermInit` function is obtained from `parms->err`. This routine returns `0` on success or one of the error codes (see, Table 4-13 on page 4-80) on failure.

Notes

1. If SC_AUTOINIT is enabled, pSOS+ calls TermInit() directly. It initializes the terminal driver. If SC_APP_CONSOLE is not 0 then the application channel is opened. In this case, the application need not call de_init() again.
2. If SC_AUTOINIT is not enabled, the application needs to call de_init() to initialize the terminal driver. If the minor device number passed to de_init() is zero, de_init() returns after initializing the terminal driver. If it is nonzero, that channel is opened after the initialization.
3. If the terminal driver is already initialized, de_init() just returns.

Examples

Refer to the examples in section, *TermOpen*, which follows.

TermOpen

```
void TermOpen (struct ioparms *parms);
```

This DITI function opens a specific serial channel for use. It is accessed by the application through pSOS+ call de_open():

```
unsigned long de_open(unsigned long dev, void *iopb, void
*retval);
```

The parameters of de_open() are mapped to the fields in TermOpen input parameter parms by pSOS+. The parameters are:

dev	This parameter is mapped to parms->in_dev and specifies the major and minor device numbers, which are stored in the upper and lower 16 bits, respectively.
iopb	Not used by the TermOpen function. This can be any uninitialized unsigned long pointer.
retval	The retval parameter is mapped to parms->out_retval and contains any additional information that the TermOpen function needs to return to the application, else it is set to 0.

Returns

The value returned by the TermOpen function is obtained from parms->err. This routine returns 0 on success, or upon failure, one of the error codes listed in Table 4-13 on page 4-80.

Examples

Example 1: If the application only uses a serial channel for reading and writing:

1. Set SC_AUTOINIT in sys_conf.h to IO_AUTOINIT.
2. Set SC_APP_CONSOLE in sys_conf.h to the minor device number of the channel to be used for reading and writing.

   No de_init() or de_open() calls are needed in the application as pSOS+ initializes the terminal driver and opens the channel specified by SC_APP_CONSOLE.

Example 2: If the application does not want pSOS+ to perform terminal driver initialization:

1. Set SC_AUTOINIT in sys_conf.h to IO_NOAUTOINIT.
2. Set SC_APP_CONSOLE to the minor device number of the channel to be used by the application.
3. In the application, add the code:

```
{
unsigned long retval, ioretval;
unsigned long iopb[4];
void *data;

if (retval = de_init(DEV_SERIAL|SC_APP_CONSOLE, iopb,
                     &ioretval, &data))
   printf("Error in de_init : %x\n", retval);
}
```

   In this case, the de_init() call not only initializes the terminal driver but also opens the channel specified by the minor device number, SC_APP_CONSOLE. Hence a separate de_open() call is not needed.

Example 3: If the application is going to use more than one channel:

1. Set SC_AUTOINIT in sys_conf.h to IO_AUTOINIT to initialize the terminal driver.
2. Set SC_APP_CONSOLE in sys_conf.h to the minor device number of the default channel to be opened for reading and writing.
3. For opening all other channels, add the following code into the application:

```
#define APP_CONSOLE1 X1
#define APP_CONSOLE2 X2
:
{
  unsigned long retval, ioretval;
  unsigned long iopb[4];
  void *data;

  if (retval = de_open(DEV_SERIAL|APP_CONSOLE1, iopb,
      &ioretval))
    printf("Error in de_open for %d channel : %x\n",
            APP_CONSOLE1, retval);
  if (retval = de_open(DEV_SERIAL|APP_CONSOLE2, iopb,
      &ioretval))
    printf("Error in de_open for %d channel : %x\n",
            APP_CONSOLE2, retval);
:
}
```

In this example, pSOS+ initializes the terminal driver and opens the default channel specified by SC_APP_CONSOLE. The rest of the channels, APP_CONSOLE1, APP_CONSOLE2, and so on, are opened by the application.

TermRead

```
void TermRead (struct ioparms *parms);
```

This DITI function reads from a channel and is accessed by the application through the pSOS+ call `de_read()`,

```
unsigned long de_read(unsigned long dev, void *iopb, void
*retval);
```

The parameters of the `de_read()` function are mapped to the fields in `TermRead` input parameter `parms` by pSOS+. The parameters are:

`dev` This parameter is mapped to `parms->in_dev` and specifies the major and minor device numbers, which are stored in the upper and lower 16 bits, respectively.

`iopb` This parameter is mapped to `parms->in_iopb`. This is a pointer of type `TermIO` structure:

```
typedef struct {
     unsigned long length; /* length of read/write */
     unsigned char *buffp; /* pointer to data buffer */
}TermIO;
```

The `TermRead` function fills the buffer pointed to by `buffp` according to the terminal parameters. The parameter, `length`, specifies the number of characters to be read.

`retval` This parameter is mapped to `parms->out_retval` and contains the number of characters read.

Returns

The value returned by the `TermRead` function is obtained from `parms->err`. This routine returns 0 on success, or upon failure, one of the error codes listed in Table 4-13 on page 4-80.

Notes

`SC_APP_CONSOLE` is the default serial channel used for reading when no minor device number is specified in a `de_read()`call.

Examples

Example 1: To read 10 characters into an array `str[]` from `SC_APP_CONSOLE`:

```
{
 unsigned long ioretval, retval;
 struct TermIO tst_io;
 unsigned char str[150];

 tst_io.length = 10;
 tst_io.buffp = str;
 if (retval = de_read(DEV_SERIAL, (void *)&tst_io, &ioretval))
     printf("Error in de_read : %x\n", retval);
}
```

Example 2: To read 10 characters into an array, `str[]`, from `SC_APP_CONSOLE` without using `struct TermIO`:

```
{
 unsigned long ioretval, retval;
 unsigned long iopb[4];
 unsigned char str[150];

 iopb[0] = 10;
 iopb[1] = (unsigned long)str;
 if (retval = de_read(DEV_SERIAL, (void *)&iopb, &ioretval))
  printf("Error in de_read : %x\n", retval);
}
```

NOTE: Instead of using `struct TermIO`, an array of type unsigned long with first parameter set to length and second to the string can also be used in a `de_read()`call.

Example 3: To read 10 characters from a channel other than `SC_APP_CONSOLE`:

```
{
  unsigned long ioretval, retval;
  unsigned long iopb[4];
  unsigned char str[150];
  iopb[0] = 10;
  iopb[1] = (unsigned long) str;
  if (retval = de_read(DEV_SERIAL|<minor_dev_no>, (void*)&iopb,
                       &ioretval))
     printf("Error in de-read: %x\n", retval;
}
```

NOTE: *<minor_dev_no>* is the channel number to read.

TermWrite

```
void TermWrite (struct ioparms *parms);
```

This DITI function writes to a channel. It is accessed by the application through pSOS+ call `de_write()`,

```
unsigned long de_write(unsigned long dev, void *iopb, void
*retval);
```

The parameters of the `de_write()` function are mapped to the fields in `TermWrite` input parameter `parms` by pSOS+. The parameters are:

`dev`	This parameter is mapped to `parms->in_dev` and specifies the major and minor device numbers, which are stored in the upper and lower 16 bits, respectively.
`iopb`	This parameter is mapped to `parms->in_iopb`. This is a pointer of type `TermIO` structure:

```
typedef struct {
 unsigned long length; /* length of read/write */
 unsigned char *buffp; /* pointer to data buffer */
}TermIO;
```

The `TermWrite` function writes the characters pointed to by `buffp` according to the terminal parameters. The parameter, `length`, specifies the number of characters to be written.

`retval`	This parameter is mapped to `parms->out_retval` and contains the number of characters written.

Returns

The value returned by `TermWrite` is obtained from `parms->err`. This routine returns 0 on success, or upon failure one, of the error codes listed in Table 4-13 on page 4-80.

Notes

`SC_APP_CONSOLE` is the default serial channel used for writing when no minor device number is specified in a `de_write()` function call.

Examples

Example 1: To write a string `Hello World` to `SC_APP_CONSOLE`:

```
{
 unsigned long ioretval, retval;
 struct TermIO tst_io;

 tst_io.length = sizeof("Hello World\n");
 tst_io.buffp = "Hello World\n";
 if (retval = de_write(DEV_SERIAL, (void *)&tst_io, &ioretval))
    printf("Error in de_write : %x\n", retval);
}
```

Example 2: To write a string `Hello World` to `SC_APP_CONSOLE` without using `struct TermIO`:

```
{
 unsigned long ioretval, retval;
 unsigned long iopb[4];

 iopb[0] = sizeof("Hello World\n");
 iopb[1] = (unsigned long)Hello World\n";
 if (retval = de_write(DEV_SERIAL, (void *)&iopb, &ioretval))
   printf("Error in de_write: %x\n", retval);
}
```

NOTE: Instead of using `struct TermIO`, an array of type unsigned long with the first parameter set to length and second to the string can also be used in a `de_write()`call.

Example 3: To write to a channel other than `SC_APP_CONSOLE`,

```
{
  unsigned long ioretval, retval;
  struct TermID tst_io;
  tst_io.length = sizeof("Hello World\n");
  tst_io.buffp = "Hello World\n";
  if (retval = de_write(DEV_SERIAL|<minor_dev_number>, (void *)
                        &tst_io, &ioretval))
    printf("Error in dewrite: %x\n", retval);
}
```

NOTE: The channel number to write to is specified by *`<minor_dev_number>`*.

TermIoctl

```
void TermIoctl(struct ioparms *parms);
```

The DITI function, `TermIoctl`, performs control functions on the channel. This function is accessed by the application through pSOS+ call `de_cntrl()`,

```
unsigned long de_cntrl(unsigned long dev, void *iopb, void
*retval);
```

Most of the `TermIoctl` control functions are used to change the configuration of a device or get the current value of the configuration parameters.

The parameters of the `de_cntrl()` function are mapped to the fields in `TermIoctl` input parameter `parms` by pSOS+. The parameters are:

`dev` This parameter is mapped to `parms->in_dev` and specifies the major and minor device numbers, which are stored in the upper and lower 16 bits, respectively.

`iopb` This parameter is mapped to `parms->in_iopb`. This is a pointer of type `TermCtl` structure:

```
typedef struct {
  unsigned long function; /* function code for the I/O
                             control call */
  void *arg;              /* pointer to function dependent
                             information */
}TermCtl;
```

The `function` argument can be one of the functions listed in section, *TermIoctl Functions* on page 4-71, and `arg` is the argument for that function.

`retval` This parameter is mapped to `parms->out_retval` and contains any additional information that `TermIoctl` needs to return to the application. Otherwise it is set to `0`.

Returns

The value returned by `TermIoctl` is obtained from `parms->err`. This routine returns 0 on success, or upon failure, one of the error codes listed in Table 4-13 on page 4-80.

Notes

SC_APP_CONSOLE is the default serial channel used when no minor service number is specified in de_cntrl().

TermIoctl Functions

Table 4-12 describes the TermIoctl functions. Refer to section, *TermIoctl* on page 4-70 for a description of the DITI TermIoctl function call syntax.

TABLE 4-12 TermIoctl Control Functions

Control Function	Description
TCGETS	The parameter, arg, is a pointer to a termios structure. The current terminal parameters are fetched and stored into that structure.
TCSETS	The parameter, arg, is a pointer to a termios structure. The current terminal parameters are set from the values stored in that structure. The change is immediate. This DITI TermIoctl is equivalent to POSIX tcsetattr() function except that the DITI TermIoctl does not interpret the baud rate being equal to 0 condition. As per POSIX, if baud rate is 0, the connection should be terminated. In DITI, no such action is taken.
TCSETSW	The parameter, arg, is a pointer to a termios structure. The current terminal parameters are set from the values stored in that structure. The change occurs after all characters queued for output have been transmitted. This form should be used when changing parameters that affect output.
TCSETSF	The parameter, arg, is a pointer to a termios structure. The current terminal parameters are set from the values stored in that structure. The change occurs after all characters queued for output have been transmitted; all characters queued for input are discarded and then the change occurs.
TCGETA	This TermIoctl function is the same as TCGETS.
TCSETA	This TermIoctl function is the same as TCSETS.

TABLE 4-12 `TermIoctl` Control Functions (Continued)

Control Function	Description
`TCSETAW`	This `TermIoctl` function is the same as `TCSETSW`.
`TCSETAF`	This `TermIoctl` function is the same as `TCSETSF`.
`TCSBRK`	The parameter, `arg`, is not used. A break character is sent out the channel. This DITI `TermIoctl` function is equivalent to POSIX `tcsendbreak()` function except that the DITI `TermIoctl` function just sends a **BREAK** character out the channel and does not use any argument to indicate the duration of the **BREAK** (zero valued bits) sent out the channel.
`TCXONC`	Start/stop control. The `arg` parameter is an int type value. If `arg` is `TCIOFF(2)`, input is suspended. If `arg` is `TCION(3)`, suspended input is restarted. This DITI `TermIoctl` function is equivalent to the POSIX `tcflow()` function except that the DITI `TermIoctl` function does not support two actions, `TCOOFF` (suspend output) and `TCOON` (restart suspended output).
`TCFLSH`	The `arg` parameter is an int type value. If `arg` is: `TCIFLUSH(0)`, flush the input queue `TCOFLUSH(1)`, flush the output queue `TCIOFLUSH(2)`, flush both input and output queues
`TIOCMBIS`	The `arg` parameter is a pointer to an int type whose value is a mask containing modem control lines to be turned on. The control lines whose bits are set in the `arg` are turned on; no other control lines are affected.
`TIOCMBIC`	The `arg` parameter is a pointer to an int type whose value is a mask containing modem control lines to be turned off. The control lines whose bits are set in the `arg` are turned off; no other control lines are affected.
`TIOCMGET`	The parameter, `arg`, is a pointer to an `int` type. The current state of the modem status lines is fetched and stored in the int pointed to by `arg`.

TABLE 4-12 TermIoctl Control Functions (Continued)

Control Function	Description
TIOCMSET	The parameter, arg, is a pointer to an int type containing a new set of modem control lines. The modem control lines are turned on or off, depending on whether the bit for that mode is set or clear.
TERMGETDEFTERM	Copies the current minor number of the system console to the unsigned long pointed to by arg.
TERMPUTDEFTERM	Changes the current value of the system console to the unsigned long pointed to by arg.
TERMHWFC	The channel uses CTS/RTS flow control. This command has no effect if the channel does not support hardware flow control.
TERMNUMOFCHANNELS	The arg is a pointer to a unsigned long that is filled in with the total number of serial channels.
TERMGETASYNCSTAT	The arg parameter is a pointer to a ASYNC_STAT structure which is filled in with the channels asynchronous status. This command is used by the MIB (Management Information Base) agent to get information on the channel. The ASYNC_STAT structure is as follows:

```
typedef struct
   {
   unsigned long ParityErrs;   /* number of parity errors */
   unsigned long FramingErrs;  /* number of framing errors */
   unsigned long OverrunErrs;  /* number of overrun errors */
   unsigned long AutoBaudEnb;  /* is auto baud enabled */
   } ASYNC_STAT
```

TABLE 4-12 `TermIoctl` Control Functions (Continued)

Control Function	Description
TERMGETSYNCSTAT	The `arg` parameter is a pointer to a SYNC_STAT structure which is filled in with the channels synchronous status. This command is used by the MIB agent to get information on the channel. The SYNC_STAT structure is as follows:

```
typedef struct
{
 unsigned long FrameCheckErrs;  /* number of frame check errors */
 unsigned long TransmitUnderrunErrs;  /* number of transmit
                                         underrun errors */
 unsigned long ReceiveOverrunErrs;    /* number of receive
                                         overruns errors */
 unsigned long InterruptedFrames;     /* number of frames that
                                         were stopped */
 unsigned long AbortedFrames; /* number of frames that were aborted */
} SYNC_STAT;
```

Examples

Example 1: To get `termio` structure and extract the following information:

- Baud rate of the channel
- Whether operating in canonical or non-canonical mode
- Values of MinChar and MaxTime
- Output mode information

```
#include <disi.h>
#include <diti.h>
#include <termios.h>
{
  unsigned long retval, ioretval, baudrate;
  TermCtl tst_ctrl;
  struct termio tst_termio;

  /* Get the current configuration */
  tst_ctrl.function = TCGETS;
  tst_ctrl.arg = (void *)&tst_termio;
  if (retval = de_cntrl(DEV_SERIAL, (void *)&tst_ctrl,
                        &ioretval))
    printf("Error in de_cntrl TCGETS : %x\n", retval);
  /* Extract baud rate information */
  baudrate = tst_termio.c_cflag & CBAUD;
  printf("Baud rate of the channel is : %d\n", baudrate);

  /* Extract information of input mode processing */
  if (tst_termio.c_lflag & ICANON) {
    printf("Input mode processing is CANONICAL\n");
    printf("MinChar and MaxTime do not play any role\n");
  }
  else {
    printf("Input mode processing is NON-CANONICAL\n");
    printf("MinChar : %d\n", tst_termio.c_cc[VMIN]);
    printf("MaxTime : %d\n", tst_termio.c_cc[VTIME];
  }

  /* Extract output mode information */
  if (tst_termio.c_oflag & OPOST) {
    if (tst_termio.c_oflag & ONLCR)
      printf("ONLCR set\n");
    if (tst_termio.c_oflag & ONOCR)
      printf("ONOCR set\n");
  }
  else
    printf("No post processing of output\n");
}
```

Example 2: To change the following parameters:

- Baud rate to 19200
- Set to non-canonical mode with MinChar=10 and MaxTime=10 seconds
- Set to no ECHO
- Set CSIZE to 7
- Set only OLCUC in output flags
- Enable IXON in input flags

```
  #include <disi.h>
  #include <diti.h>
  #include <termios.h>
  {
    unsigned long retval, ioretval;
    TermCtl tst_ctrl;
    struct termio tst_termio;

    /* Get the current configuration */
```

```
      tst_ctrl.function = TCGETS;
      tst_ctrl.arg = (void *)&tst_termio;
      if (retval = de_cntrl(DEV_SERIAL, (void *)&tst_ctrl,
                            &ioretval))
        printf("Error in de_cntrl TCGETS : %x\n", retval);

      /* Set baudrate to 19200 */
      tst_termio.c_cflag = tst_termio.c_cflag & ~CBAUD | B19200;

      /* Set to non-canonical and no echo mode */
      tst_termio.c_lflag = tst_termio.c_lflag & ~ICANON & ~ECHO;

      /* Set MinChar=10, MaxTime=10 sec or 100 0.1 secs */
      tst_termio.c_cc[VMIN] = 10;
      tst_termio.c_cc[VTIME] = 100;

      /* Set CSIZE to 7 */
      tst_termio.c_cflag = tst_termio.c_cflag & ~CSIZE | CS7;

      /* Set only OLCUC in output flags */
      tst_termio.c_oflag = OPOST | OLCUC;

      /* Enable IXON in input flags */
      tst_termio.c_iflag |= IXON;

      /* Set the changed configuration after discarding any input */
      /* and completing all output */
      tst_ctrl.function = TCSETSF;
      tst_ctrl.arg = (void *)&tst_termio;
      if (retval = (de_cntrl(DEV_SERIAL, (void *)&tst_ctrl,
                    &ioretval))
        printf("Error in de_cntrl TCSETSF : %x\n", retval);
    }
```

Example 3: To activate hardware flow control:

```
    #include <disi.h>
    #include <diti.h>
    #include <termios.h>
    {
    unsigned long retval, ioretval;
    TermCtl tst_ctrl;

    tst_ctrl.function = TERMHWFC;
    tst_ctrl.arg = 0;  /* not used */
    if (retval = (de_cntrl(DEV_SERIAL, (void *)&tst_ctrl, &ioretval))
      printf("Error in de_cntrl TERMHWFC : %x\n", retval);
    }
```

Example 4: To suspend input:

```
#include <disi.h>
#include <diti.h>
#include <termios.h>
{
  unsigned long retval, ioretval;
  TermCtl tst_ctrl;

  tst_ctrl.function = TCXONC;
  tst_ctrl.arg = TCIOFF;  /* Suspend input */
  if (retval = (de_cntrl(DEV_SERIAL, (void *)&tst_ctrl,
                &ioretval))
    printf("Error in de_cntrl TCXONC: %x\n", retval);
}
```

Example 5: To get status of RTS and CTS modem lines:

```
#include <disi.h>
#include <diti.h>
#include <termios.h>
{
  unsigned long retval, ioretval;
  TermCtl tst_ctrl;

  tst_ctrl.function = TIOCMGET;
  tst_ctrl.arg = 0;  /* Not used as input parameter */
  if (retval = (de_cntrl(DEV_SERIAL,  (void *)&tst_ctrl,
                &ioretval))
    printf("Error in de_cntrl TIOCMGET : %x\n", retval);
  if (*(ULONG *)arg & TIOCM_RTS)
    printf("RTS signal set");
  if (*(ULONG *)arg & TIOCM_CTS)
    printf("CTS signal set");
}
```

Example 6: To determine errors such as framing or parity on an asynchronous channel:

```
#include <disi.h>
#include <diti.h>
#include <termios.h>
{
  unsigned long retval, ioretval;
  TermCtl tst_ctrl;

  tst_ctrl.function = TERMGETASYNCSTAT;
  tst_ctrl.arg = 0;  /* Not used as input parameter */
  if (retval = (de_cntrl(DEV_SERIAL, (void *)&tst_ctrl,
                &ioretval))
    printf("Error in de_cntrl TERMGETASYNCSTAT : %x\n", retval);
  printf("Parity errors  : %d\n",
         (ASYNC_STAT *)arg)->ParityErrs);
  printf("Framing errors : %d\n",
         (ASYNC_STAT *)arg)->FramingErrs);
  printf("Overrun errors : %d\n",
         (ASYNC_STAT *)arg)->OverrunErrs);

}
```

NOTE: In all the above examples, the channel used is SC_APP_CONSOLE. If any other channel needs to be used, logically OR DEV_SERIAL with that minor device number.

TermClose

```
void TermClose (struct ioparms *parms);
```

This DITI function closes the channel and is accessed by the application through pSOS+ call de_close(),

```
unsigned long de_close(unsigned long dev, void *iopb, void
*retval);
```

This function flushes all transmit buffers, discards all pending receive buffers and disables the receiver and transmitter of the channel. All buffers associated with the channel are released (freed) and the device will hang up the line. All further reads, writes or TermIoctl functions to the channel return with the error TERM_NOPEN.

All channels that are opened either through de_init() or de_open() calls need to be closed by using a de_close() function call.

The parameters of the de_close() function are mapped to the fields in TermClose input parameter parms by pSOS+. The parameters are:

dev	This parameter is mapped to parms->in_dev and specifies the major and minor device numbers, which are stored in the upper and lower 16 bits, respectively.
iopb	This parameter is not used by the TermClose function. This can be any uninitialized unsigned long pointer.
retval	This parameter is mapped to parms->out_retval and contains any additional information that the TermIoctl function needs to return to the application. Otherwise, it is set to 0.

Returns

The value returned by TermClose is obtained from parms->err. This routine returns 0 on success, or upon failure, one of the error codes listed in Table 4-13 on page 4-80.

Notes

SC_APP_CONSOLE is the default channel used if no minor device number is specified in de_close().

Examples

Example 1: To close SC_APP_CONSOLE:

```
{
  unsigned long retval, ioretval;
  unsigned long iopb[4];

  if (retval = de_close(DEV_SERIAL, iopb, &ioretval))
    printf("Error in de_close : %x\n", retval);
}
```

Example 2: To close any channel other than SC_APP_CONSOLE:

```
{
  unsigned long retval, ioretval;
  unsigned long iopb[4];
  if (retval = de_close(DEV_SERIAL|<minor_dev_no>, iopb,
                        &ioretval))
    printf("Error in de_close: %n", retval);
}
```

NOTE: *<minor_dev_no>* is the channel number to be closed.

Error Codes

Table 4-13 lists the error codes that are located in the header file, `diti.h`.

TABLE 4-13 DITI Functions Error Codes

```
#define TERM_HDWR       0x10010200   /* Hardware error */

#define TERM_MINOR      0x10010201   /* Invalid minor device */

#define TERM_BAUD       0x10010203   /* Invalid baud rate */

#define TERM_NINIT      0x10010204   /* driver not initialized */

#define TERM_DATA       0x10010205   /* Unable to allocate driver
                                        data area */

#define TERM_SEM        0x10010206   /* Semaphore error */

#define TERM_AINIT      0x10010210   /* Terminal already
                                        initialized */

#define TERM_CHARSIZE   0x10010211   /* bad character size */

#define TERM_BADFLAG    0x10010212   /* flag not defined */

#define TERM_NHWFC      0x10010213   /* Hardware flow control not
                                        supported */

#define TERM_BRKINT     0x10010214   /* Terminated by a break
                                        character */

#define TERM_DCDINT     0x10010215   /* Terminated by loss of
                                        DCD */

#define TERM_NBUFF      0x10010216   /* No buffers to copy
                                        characters */

#define TERM_NOPEN      0x10010217   /* minor device has not been
                                        opened */

#define TERM_AOPEN      0x10010218   /* channel already opened */

#define TERM_ADOPEN     0x10010219   /* channel already opened
                                        by another driver */

#define TERM_CFGHSUP    0x10010220   /* hardware does not
                                        support channel as
                                        configured */
```

TABLE 4-13 DITI Functions Error Codes (Continued)

```
#define TERM_OUTSYNC    0x10010221   /* out of sync with DISI */

#define TERM_BADMIN     0x10010222   /* MinChar > RBuffSize */

#define TERM_LDERR      0x10010223   /* Lower driver error may
                                        be corrupted structure */

#define TERM_QUE        0x10010224   /* que error */

#define TERM_RXERR      0x10010225   /* data receive error */

#define TERM_TIMEOUT    0x10010226   /* Timer expired for read
                                        or write */

#define TERM_ROPER      0x10010228   /* redirect operation
                                        error */

#define TERM_MARK       0x10010229   /* received a SIOCMARK */

#define TERM_FRAMING    0x10010230   /* framing error */

#define TERM_PARITY     0x10010231   /* parity error */

#define TERM_OVERRUN    0x10010232   /* overrun error */

#define TERM_NMBLK      0x10010233   /* no buffer headers
                                        (esballoc failed) */

#define TERM_TXQFULL    0x10010234   /* transmit queue is full */

#define TERM_WNWCONF    0x10010235   /* MaxWTime & WNWAIT
                                        both set*/

#define TERM_BADCONSL   0x10010236   /* Bad default console
                                        number */

#define TERM_WABORT     0x10010237   /* Write was aborted */

#define TERM_NOMEM      0x10010238   /* Not enough memory in
                                        Region 0 */
```

5 pSOSystem Configuration File

The pSOSystem software is a scalable environment and a large part of its functionality is provided in software component building blocks. Note that not all of the software components are built into a pSOSystem configuration, because the components to include depend on the capabilities required by the system being implemented. Each software component, however, has its own configuration and startup requirements. By default, the pSOSystem startup code takes care of these requirements. This chapter describes the configuration and startup elements used for customizing the startup code.

Many system parameters are controlled by `#define` statements located within the `sys_conf.h` header file. These statements range from specifying the device drivers that are built into the system to the maximum number of tasks that can be active concurrently. You can easily change the pSOSystem configuration by editing the `sys_conf.h` header file which resides in the application directory. This chapter documents the `sys_conf.h` system parameters and their significance.

Overview

Software components are the basic building blocks of the pSOSystem environment. Each component has an associated *configuration table*, which the component uses to obtain all of its configurable parameters. Figure 5-1 shows the relationships between the various elements that the components use to find their parameters, data areas, and other configuration information.

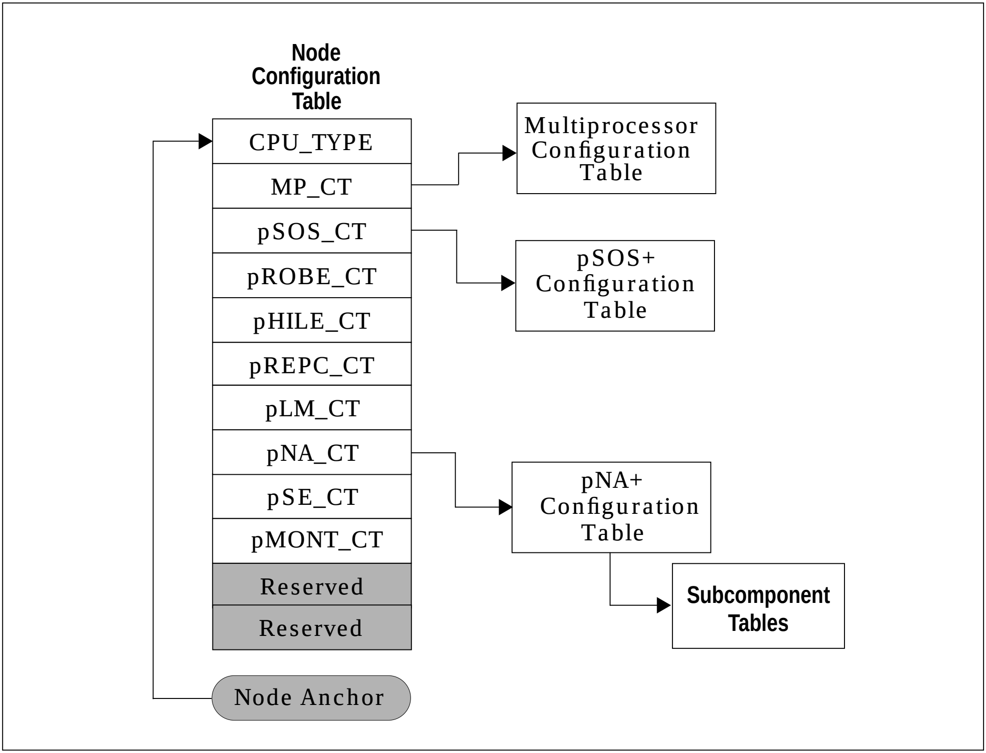


FIGURE 5-1 Node Configuration Table

The *node anchor* is the single, fixed point of reference for all the installed software components in the system. This anchor is a critical link because each component is code and data position independent and thus depends on the anchor to locate its configuration information.

In the pSOSystem environment, the global symbol anchor serves as the node anchor. An anchor is a pointer to the node configuration table. Each configuration table entry is described in, Chapter 6, *Configuration Tables*.

Since the pSOSystem startup code sets up these tables, you do not need to understand all table entries during the initial phases of development. However, as you experiment with changing the pSOSystem configuration, this manual provides the reference information that enables you to configure the pSOSystem to your exact requirements.

System Configuration File

During system startup, the pSOSystem software initializes all the required component configuration tables. The code that initializes the configuration tables resides in the shared directory `PSS_ROOT/configs/std`. The source code in these files contains many conditional compiled C statements where compilation depends on values defined in the `sys_conf.h` header file. This includes the following:

- The core components that are built into the system.
- The serial channel characteristics of the target.
- Whether the system includes a LAN driver and, if so, the IP address of the target system.
- Whether the system includes a shared memory network interface (SMNI) and, if so, IP address of the system.
- The optional device drivers in the system: the SCSI and RAM disk drivers, the TFTP driver, pseudo-driver, and any application specific drivers added to the system.
- The values used for most of the component configuration table entries. For example, definitions in the `sys_conf.h` header file determine the maximum number of concurrently active tasks and message queues of the system.

Each of the sample applications supplied with the pSOSystem software includes a `sys_conf.h` header file that contains configuration values appropriate for that application. See section, *sys_conf.h* on page 5-5, for the description of the values you must define in `sys_conf.h`.

Parameter Storage and the Startup Dialog

Sometimes the only differences between pSOSystem configurations are the values of the parameters defined in `sys_conf.h`. For this reason, the pSOSystem software allows you to place some of the `sys_conf.h` parameter values in a dedicated storage area in the memory of the target system. An optional startup dialog can be built into the operating system that allows review and possible modification of these parameters when the pSOSystem software initializes itself. The pSOSystem Boot ROMs is an example of a pSOSystem application that uses the startup dialog.

sys_conf.h

This section describes the parameters that sys_conf.h must supply. Parameter definitions in sys_conf.h have the form of C macro definitions, as in the following example:

```
#define SC_PROBE     YES  /* pROBE+ processor services) */
#define KC_NTASK     20   /* Max. of 20 active tasks */
```

You may find it helpful to refer to the example sys_conf.h files in the pSOSystem sample applications while reading this section.

To improve the readability of sys_conf.h, macros should be used to define the values of some of the parameters. The code in sysinit.c (and other files that include sys_conf.h) assumes the use of these macros. Therefore, sys_conf.h should include the macro definitions file before any of the parameter values are defined:

```
#include   <sys/types.h>
#include   <sysvars.h>
#include   <psos.h>

#define USE_RARP           NO
```

Storage and Dialog Parameters

Table 5-1 lists the storage and dialog parameters, which alter the way many of the other parameters in sys_conf.h are used.

TABLE 5-1 Storage and Dialog Parameters

Parameter	Possible Values
SC_SD_PARAMETERS	STORAGE (default), SYS_CONF
SC_STARTUP_DIALOG	NO (default), YES
SC_BOOT_ROM	NO (default), YES. NO for RAM application, YES for ROM applications.
SD_STARTUP_DELAY	Any decimal integer (Default is 60 seconds.)
SC_SD_DEBUG_MODE	STORAGE (default), DBG_SA, DBG_XS, DBG_XN, DBG_AP

The values of the parameters in sys_conf.h with names beginning with SD_ can be determined either by the definitions given in the sys_conf.h file or by the data in the parameter storage area of the target. For example, if SC_SD_PARAMETERS is set to SYS_CONF, the values in the sys_conf.h file are always used for the SD_ parameter values. If SC_SD_PARAMETERS is set to STORAGE, then the pSOSystem software attempts to use the values in the parameter storage area of the target for the SD_ variables. The values located in sys_conf.h become default values to use if the parameter storage area has not been initialized or has been corrupted.

If SC_SD_PARAMETERS is defined as STORAGE, you can enable the startup dialog by setting SC_STARTUP_DIALOG to YES. The startup dialog executes on the target system at startup time and allows you to view and optionally change the parameter values in the storage area. If the dialog is enabled, SD_STARTUP_DELAY specifies the number of seconds the dialog waits for input before it boots the system.

The SC_SD_DEBUG_MODE setting determines how the system operates according to the possible values shown in Table 5-2.

TABLE 5-2 Debug Mode Values

Value	Description
DBG_SA	Boot the pROBE+ debugger in standalone mode.
DBG_AP	Boot the pROBE+ debugger in standalone mode, and do a silent startup.
DBG_XS	Boot into the pROBE+ debugger and wait for the host debugger through a serial connection.
DBG_XN	Boot into the pROBE+ debugger and wait for the host debugger through a network connection.
STORAGE	Use the mode found in the parameter storage area (DBG_SA, DBG_XS, DBG_XN, or DBG_AP). If a valid mode is not found, use DBG_SA.

Operating System Components

The following parameters, listed in Table 5-3, indicate which components are in the system. The USEROM option is dependent on the CPU architecture and may not be available for all the components.

TABLE 5-3 Operating System Components

Parameter	Possible Values	Component
SC_PSOS	YES, NO, USEROM	pSOS+ real-time kernel
SC_PSOSM	YES, NO, USEROM	pSOS+ real-time multiprocessing kernel
SC_PSOS_QUERY	YES, NO	pSOS+ kernel query services
SC_PROBE	YES, NO, USEROM	pROBE+ processor services
SC_PROBE_DISASM	YES, NO, USEROM	pROBE+ disassembler
SC_PROBE_CIE	YES, NO, USEROM	pROBE+ console executive
SC_PROBE_QUERY	YES, NO, USEROM	pROBE+ query services
SC_PROBE_DEBUG	YES, NO, USEROM	pROBE+ debug interface executive
SC_PROBE_HELP	YES, NO, USEROM	pROBE+ help command handler
SC_PHILE	YES, NO, USEROM	pHILE+ file system manager
SC_PHILE_PHILE	YES, NO, USEROM	pHILE+ real-time file system
SC_PHILE_MSDOS	YES, NO, USEROM	pHILE+ MS-DOS FAT file system
SC_PHILE_NFS	YES, NO, USEROM	pHILE+ NFS client
SC_PHILE_CDROM	YES, NO, USEROM	pHILE+ ISO 9660 CD-ROM file system
SC_PREPC	YES, NO, USEROM	pREPC+ C runtime library
SC_PNA	YES, NO, USEROM	pNA+ TCP/IP networking manager
SC_PNET	YES, NO, USEROM	pNET library for BOOT ROMs
SC_PRPC	YES, NO, USEROM	pRPC+ Remote Procedure Call (RPC) component
SC_PSE_PRPC	YES, NO, USEROM	pRPC+ component over pSE+

TABLE 5-3 Operating System Components (Continued)

Parameter	Possible Values	Component
SC_PSE	YES, NO,USEROM	pSE+ streams component
SC_PSKT	YES, NO,USEROM	pSKT SKT library component
SC_PTLI	YES, NO,USEROM	pTLI+ TLI library component
SC_PMONT	YES, NO,USEROM	pMONT+ component
SC_PLM	YES, NO,USEROM	pLM+ Shared Library Manager
SC_PROFILER	YES, NO	RTA profiler configuration
SC_RTEC	YES, NO	RTA run-time error checker library
SC_POSIX	YES, NO	POSIX core component
SC_POSIX_MESSAGE_PASSING		
	YES, NO	POSIX message queue services
SC_POSIX_SEMAPHORES		
	YES, NO	POSIX semaphore services
SC_POSIX_THREADS	YES, NO	POSIX pthread services
SC_POSIX_TIMERS	YES, NO	POSIX clock and timer services

A component parameter set to YES causes the component to be built into the system. The USEROM option allows you to use the component from the boot ROM. The USEROM option allows you to save memory in the specified component. To use this option the component should be built into the boot ROM.

NOTE: It is incorrect to set both SC_PSOS, for the pSOS+ kernel, and SC_PSOSM, for the pSOS+m multiprocessing kernel, to YES.

The target resident pROBE+ debugger consists of five submodules, which you may include in the executable image. To include any of these modules, the system must have the processor services module. To include the processor services module, set SC_PROBE to YES. SC_PROBE_DISASM and SC_PROBE_QUERY are used to enable the disassembler and query services, respectively (both are optional). You should set one or both of SC_PROBE_CIE and SC_PROBE_DEBUG to YES. If you plan to use the pROBE+ debugger in console mode, set SC_PROBE_CIE to YES. If you plan to use the pROBE+ debugger as a back end for a source level debugger, set SC_PROBE_DEBUG to YES.

SC_PROBE_HELP can be set to YES to enable the help (he) command. This command displays all the pROBE+ console mode commands.

Bindings for pSOS+ System Calls

Table 5-4 lists the parameter associated with the Quick bindings function.

TABLE 5-4 Quick Bindings Parameter

Parameter	Possible Values
SC_QBIND	YES, NO (default)

There are two paths for entry into the pSOS+ kernel when a pSOS+ system call is invoked from an application. In the normal path (with normal bindings), a trap or system call exception is generated to enter the kernel, and switch the processor supervisory mode. In the quick path (with quick bindings), no trap is raised for a pSOS+ system call. Instead, the entry into the kernel is by way of a simple jump to the service entry point. The application can choose to employ normal bindings by setting SC_QBIND to NO. If SC_QBIND is set to YES, then the quick bindings for the pSOS+ system calls is used. Quick bindings cannot be supported for applications with USER mode tasks. So, if SC_QBIND is set to YES, it should be ensured that the USER mode tasks in the application do not access any pSOS+ system calls.

Auto Initialization Sequence Enabling

Table 5-5 lists the parameter associated with auto-initialization.

TABLE 5-5 Auto Initialization Parameter

Parameter	Possible Values
SC_AUTOINIT	IO_AUTOINIT, IO_NOAUTOINIT

If SC_AUTOINIT is set to IO_AUTOINIT, device drivers that are installed by the InstallDriver function have their auto-initialization field set. This causes pSOS+ to call the initialization function of the driver when pSOS+ starts up. If this is done, the de_init call does not have to be invoked for each driver.

NOTE: Auto-initialization does not work on all drivers. Setting IO_AUTOINIT only effects drivers that can use the auto-initialization feature.

Serial Channel Configuration

The designation for each serial channel on a target takes the form of the channel number and driver number. Channel numbers start at one and driver numbers start at zero. For example, a target with eight serial channels per driver and with three drivers would have channel numbers shown in Table 5-6.

The serial driver number (`SERIAL_DRVRNUM(x)`) macro extracts the driver number based on the maximum number of serial ports per driver (`BSP_MAX_SER_PORTS_PER_DRIVER`) and maximum number of serial drivers (`BSP_MAX_SER_DRIVERS`) defined in `bsp.h`. Table 5-6 lists the channel number calculations based on the default setting, which are:

```
#define BSP_MAX_SER_DRIVERS            8
#define BSP_MAX_SER_PORTS_PER_DRIVER   8
```

TABLE 5-6 Serial Driver Number to Channel Number Mapping Table

Serial Driver Name	Driver Number	CHANNEL Number	Effective Channel
Driver0	0	1+(SERIAL_DRVRNUM(0))	1
Driver0	0	2+(SERIAL_DRVRNUM(0))	2
Driver0	0	3+(SERIAL_DRVRNUM(0))	3
Driver0	0	4+(SERIAL_DRVRNUM(0))	4
Driver0	0	5+(SERIAL_DRVRNUM(0))	5
Driver0	0	6+(SERIAL_DRVRNUM(0))	6
Driver0	0	7+(SERIAL_DRVRNUM(0))	7
Driver0	0	8+(SERIAL_DRVRNUM(0))	8
Driver1	1	1+(SERIAL_DRVRNUM(1))	9
Driver1	1	2+(SERIAL_DRVRNUM(1))	10
Driver1	1	3+(SERIAL_DRVRNUM(1))	11
:	:	:	:
:	:	:	:
Driver1	1	8+(SERIAL_DRVRNUM(1))	16
Driver2	2	1+(SERIAL_DRVRNUM(2))	17
:	:	:	

The parameters shown in Table 5-7 control the serial channels.

TABLE 5-7 Serial Channel Configuration Parameters

Parameter	Possible Values
SD_DEF_BAUD	4800, 9600 (default), 19200, 38600 (valid baud rates)
SC_APP_CONSOLE	Valid serial channel number of the form *Channel* + *DriverNum(n)* as described in Table 5-6.
SC_PROBE_CONSOLE	Valid serial channel number of the form *Channel* + *DriverNum(n)* as described in Table 5-6.
SC_RBUG_PORT	Valid serial channel number of the form *Channel* + *DriverNum(n)* as described in Table 5-6. 0 + (SERIAL_DRVRNUM(*n*)) is used to disable the channel.
SC_NumNon_pSOSChan	1, 2

SC_DEF_BAUD specifies the default baud rate for the serial channels. A de_cntrl() call can be used to change the baud rate dynamically.

SC_APP_CONSOLE specifies the effective serial channel number used for the console channel of the application. The console channel can be changed dynamically by making a de_cntrl() call to the serial driver.

NOTE: You must assign different serial channels to SC_APP_CONSOLE.

SC_PROBE_CONSOLE specifies the effective serial channel number that the pROBE+ debugger should use for its console channel (in standalone mode). The pROBE+ console displays output and receives commands on this channel.

SC_RBUG_PORT specifies the serial channel to use for remote host debugger communication if debugger over serial channel is enabled. The default setting is 0 for disabled.

The number of non-pSOS users of serial channels is indicated using SC_NumNon_pSOSChan. These are users that are initiated before pSOS, such as pROBE+.

NOTE: These channels are not closed on a soft reset of the target board.

LAN Configuration

Table 5-8 lists the parameters that control the configuration of the LAN interface.

TABLE 5-8 LAN Configuration

Parameter	Explanation
SD_LAN1	Setting this to YES enables the LAN interface, and NO disables it
SD_LAN1_IP	IP address to use for LAN interface (SD_LAN1_IP can be set to USE_RARP, in which case the pSOSystem software uses RARP to obtain the IP address)
SD_LAN1_SUBNET_MASK	Subnet mask to use for LAN interface, or 0 for none
SC_LAN1_NMCAST	Maximum number of multicast addresses

If the target board has a LAN interface, you can enable it by setting SD_LAN1 to YES. If SD_LAN1 is NO, the values of the other SD_LAN1_* parameters are ignored or not used.

The maximum number of multicast addresses to be used by the LAN interface is specified by SC_LAN1_NMCAST. The value must not exceed the maximum number of addresses supported by the LAN driver.

VME Bus Configuration

The following parameters control the configuration of the VME interface:

Parameter	Explanation
SD_VME_BASE_ADDR	Specifies the VME bus address of dual ported RAM of a target board. The default is 0x0100'0000.

SD_VME_BASE_ADDR specifies the base address of the dual ported memory on the VME bus of the target board. This parameter is insignificant for non-VME based target boards.

Shared Memory Configuration

Table 5-9 lists the parameters that control the configuration of the shared memory interfaces.

TABLE 5-9 Shared Memory Configuration Parameters

Parameter	Explanation
SD_SM_NODE	Node number for this node.
SD_NISM	YES to enable SMNI, otherwise NO (default).
SD_NISM_IP	IP address of this node.
SD_NISM_DIRADDR	Bus address (global) of SMNI directory structure.
SC_NISM_BUFFS	Number of buffers (default is 30).
SC_NISM_LEVEL	For downloaded system specify 2, for ROM resident use 1.
SD_NISM_SUBNET_MASK	Subnet mask to use for SMNI.
SD_KISM	Number of nodes in the system that use SMKI, 0 for none (default).
SD_KISM_DIRADDR	Bus address (global) of SMKI directory structure.
SC_KISM_BUFFS	Number of buffers for SMKI.

On systems that support shared memory, you can configure a shared memory network interface (SMNI) for use with the pNA+ network manager, a shared memory kernel interface (SMKI) for use with the pSOS+m kernel or both. In either case, you need to assign a node number to each target board in the system. Node numbers are integers and start at 1. The SD_SM_NODE #define setting must be the same as the node number of the board.

SD_NISM must be set to either YES or NO. It depends on whether a shared memory network interface is included. If SD_NISM is YES, then SD_NISM_IP, SD_NISM_SUBNET_MASK, and SC_NISM_BUFFS specify the IP address, subnet mask, and number of buffers of the interface, just as the corresponding parameters do for the LAN interface.

SD_NISM_DIRADDR is the bus address of a system wide directory structure which must be accessible to all nodes in the system.

SC_NISM_LEVEL should be set to 2, with one exception. For the pSOSystem Boot ROMs, it should be 1 to allow a second, downloaded shared memory system to share the same directory structure that the Boot ROMs use.

To configure a shared memory kernel interface (SMKI), set SD_KISM to the number of nodes in the system. Setting this to 0 disables SMKI. SD_KISM_DIRADDR is set to the bus address of the system wide directory structure.

Usually, the directory structures are placed in one of the dual ported RAM areas of the board. Each pSOSystem board support package reserves some space for these structures. Refer to *pSOSystem Advanced Topics* manuals to find the locations for these structures.

Miscellaneous Parameters

This section describes the miscellaneous defines.h file.

TABLE 5-10 Gateway Node Default IP Address

Parameter	Explanation
SD_DEF_GTWY_IP	IP address of default gateway node 0 for none)

SD_DEF_GTWY_IP specifies the default gateway for the pNA+ network manager to use for packet routing. The default gateway is explained in the Boot ROM section of this manual.

NOTE: If SC_PNA is set to NO, SD_DEF_GTWY_IP is meaningless.

TABLE 5-11 Target Memory Amount

Parameter	Explanation
SC_RAM_SIZE	Amount of target memory to use (0 for all)

Normally, the pSOSystem software uses all of the unassigned memory on a board for dynamic allocation (Region 0). You can override this by setting SC_RAM_SIZE to a nonzero value. If you do, pSOSystem does not use any memory beyond SC_RAM_SIZE bytes. This is useful when building a Boot ROM because it allows you to make most of the RAM on the board available for downloading code.

I/O Devices

The parameters shown in Table 5-12 control the configuration of the I/O devices.

TABLE 5-12 I/O Devices

Parameter	Explanation
SC_DEV_SERIAL	Major device number of serial driver (preset to 1; value should not be changed)
SC_DEV_TIMER	Major device number of periodic tick timer (preset to 2; value should not be changed)
SC_DEV_RAMDISK	Major device number of RAM disk (0 for none)
SC_DEV_CONSOLE	PC-Console driver
SC_DEV_SCSI	Major device number of SCSI driver (0 for none)
SC_DEV_SCSI_TAPE	Major device number for SCSI bus, tape device
SC_DEV_IDE	IDE driver
SC_DEV_FLOPPY	Floppy driver
SC_DEV_NTFTP	New TFTP pseudo driver
SC_DEV_TFTP	Old TFTP pseudo-driver 0 for none)
SC_DEV_HTTP	HTTP pseudo driver
SC_DEV_SPI	SPI driver
SC_DEV_DLPI	DLPI pseudo driver
SC_DEV_OTCP	Major device number for TCP/IP for OpEN†
SC_IP	Major device number for IP for OpEN†
SC_ARP	Major device number for ARP for OpEN†
SC_TCP	Major device number for TCP for OpEN†
SC_UDP	Major device number for UDP for OpEN†
SC_RAW	Major device number for RAW for OpEN†
SC_LOOP	Major device number for LOOP for OpEN†

TABLE 5-12 I/O Devices (Continued)

Parameter	Explanation
SC_DEV_SOSI	Major device number for OSI for OpEN†
SC_DEV_PSCONSOLE	Pseudo console driver
SC_DEV_MEMLOG	Memory log driver
SC_DEV_RDIO	pROBE+ RDIO driver
SC_DEV_NULL	Null device driver
SC_DEV_PARALLEL	Parallel port driver
SC_DEV_CMOS	CMOS driver
SC_DEV_WATCHDOG	Watchdog driver
SC_DEV_OLAP	LAP drivers
SC_PHPI	Phpi driver
SC_LAPB	LAPB driver change 0 to 1
SC_DEV_OX25	X25 drivers
SC_X25	X.25 plp driver
SC_SNDCF	sndcf driver
SC_IPCONV	ip convergence driver
SC_DEV_ISDN	ISDN drivers
SC_PH	PH driver
SC_LAPD	LAPD driver change 0 to 1
SC_IPCD	IPCD driver change 0 to 2
SC_DEV_MLPP	MultiLink PPP drivers
SC_FRMUX	FRMUX driver
SC_PPP	PPP driver change 0 to 1
SC_PIM	PIM driver change 0 to 2
SC_DEV_LOG	Streams log driver

TABLE 5-12 I/O Devices (Continued)

Parameter	Explanation
`SC_DEV_PSMUX`	Sample Mux driver
`SC_DEV_PSLWR`	Sample loopback driver
`SC_DEV_SLLWR`	Sample loopback driver
`SC_DEV_PIPE`	Pipe driver
`SC_DEVMAX`	Maximum major device number in system

†. For information about OpEN, see the *OpEN User's Manual.*

To include a device in the system, you must specify a major device number for it. The major device number determines the slot number for the device in the pSOS+ I/O switch table. To leave a device driver out of the system, use 0 or `NO` for the major number.

NOTE: When working with major device numbers, consider the following:

- Major device 0 is reserved and cannot be used to specify a device because it means the device is not built into the system.
- The major device number cannot exceed `SC_DEVMAX`. However, you can raise the value of `SC_DEVMAX`, if necessary.

You should include the following lines in `sys_conf.h` after the `SC_DEV_*` definitions:

```
#define DEV_SERIAL      (SC_DEV_SERIAL     << 16)
#define DEV_PARALLEL    (SC_DEV_PARALLEL   << 16)
#define DEV_TIMER       (SC_DEV_TIMER      << 16)
#define DEV_RAMDISK     (SC_DEV_RAMDISK    << 16)
#define DEV_SCSI        (SC_DEV_SCSI       << 16)
#define DEV_SCSI_TAPE   (SC_DEV_SCSI_TAPE  << 16)
#define DEV_PSCONSOLE   (SC_DEV_PSCONSOLE  << 16)
#define DEV_SYSCONSOLE  ((SC_DEV_PSCONSOLE  << 16) + SYSCONSOLE_DEV)
#define DEV_PSEUDO      ((SC_DEV_PSCONSOLE  << 16) + PSEUDO_DEV)
#define DEV_STDIN       ((SC_DEV_PSCONSOLE  << 16) + STDIN_DEV)
#define DEV_STDOUT      ((SC_DEV_PSCONSOLE  << 16) + STDOUT_DEV)
#define DEV_STDERR      ((SC_DEV_PSCONSOLE  << 16) + STDERR_DEV)
#define DEV_NULL        (SC_DEV_NULL       << 16)
#define DEV_MEMLOG      (SC_DEV_MEMLOG     << 16)
#define DEV_RDIO        (SC_DEV_RDIO       << 16)
#define DEV_DLPI        (SC_DEV_DLPI       << 16)
#define DEV_TFTP        (SC_DEV_TFTP       << 16)
#define DEV_NTFTP       (SC_DEV_NTFTP      << 16)
#define DEV_HTTP        (SC_DEV_HTTP       << 16)
#define DEV_SPI         (SC_DEV_SPI        << 16)
#define DEV_WATCHDOG    (SC_DEV_WATCHDOG   << 16)
#define DEV_FLOPPY      (SC_DEV_FLOPPY     << 16)
#define DEV_IDE         (SC_DEV_IDE        << 16)
#define DEV_CMOS        (SC_DEV_CMOS       << 16)
#define DEV_CONSOLE     (SC_DEV_CONSOLE    << 16)
#define DEV_IP          (SC_IP             << 16)
#define DEV_ARP         (SC_ARP            << 16)
#define DEV_TCP         (SC_TCP            << 16)
#define DEV_UDP         (SC_UDP            << 16)
#define DEV_RAW         (SC_RAW            << 16)
#define DEV_LOOP        (SC_LOOP           << 16)
#define DEV_PHPI        (SC_PHPI           << 16)
#define DEV_LAPB        (SC_LAPB           << 16)
#define DEV_X25         (SC_X25            << 16)
#define DEV_SNDCF       (SC_SNDCF          << 16)
#define DEV_IPCONV      (SC_IPCONV         << 16)
#define DEV_PH          (SC_PH             << 16)
#define DEV_LAPD        (SC_LAPD           << 16)
#define DEV_IPCD        (SC_IPCD           << 16)
#define DEV_FRMUX       (SC_FRMUX          << 16)
#define DEV_PIM         (SC_PIM            << 16)
#define DEV_PPP         (SC_PPP            << 16)
#define DEV_LOG         (SC_DEV_LOG        << 16)
#define DEV_PSMUX       (SC_DEV_PSMUX      << 16)
#define DEV_PSLWR       (SC_DEV_PSLWR      << 16)
#define DEV_SLLWR       (SC_DEV_SLLWR      << 16)
#define DEV_PIPE        (SC_DEV_PIPE       << 16)
```

Pseudo Driver Parameters

Table 5-13 lists the pSEUDO driver configuration parameters.

TABLE 5-13 pSEUDO Driver Configuration Parameters

Parameter	Value	Description
SC_PSCNSL_SHARED_CHAN	2	Number of shared channels
SC_PSCNSL_PRIVATE_CHAN	4	Number of private channels
SC_PSCNSL_MAX_CUSTOM	1	Number of custom device
SC_PSCNSL_DEFAULT_DEV	CONSOLE	Default console device (default device is CONSOLE)

The SC_PSCNSL_SHARED_CHAN parameter controls the number of shared channels that needs to be configured in the system. The shared channels are devices attached to the pseudo driver and are used by more that one task for I/O. The system console is one such channel. This parameter is used to allocate the control block associated with a shared channel.

The SC_PSCNSL_PRIVATE_CHAN parameter controls the number of private channels needed to be configured in the system. Private channels are endpoints used by tasks for exclusive use for I/O on the channel. The stdin, stdout and stderr functions associated with a task are examples of the private channels. This parameter is used to allocate the control block for with private channel.

The SC_PSCNSL_MAX_CUSTOM parameter controls the number of custom modules that can be plugged into the pseudo driver for use.

SC_PSCNSL_DEFAULT_DEV parameter controls the default device to be mapped by the pseudo device as the default console. All reads and writes to the pSEUDO device driver are transferred to the default device. Users can change the default device by issuing I/O controls to the pSEUDO device driver. Refer to Chapter 4, *Standard pSOSystem Character I/O Interface* for more information on the pSEUDO device driver.

Other Parameters

Table 5-14 lists the TFTP, HTTP and PIPE configuration parameters.

TABLE 5-14 Pseudo Console, TFTP, HTTP and PIPE Configuration Parameters

Parameter	Value	Description
SC_MAX_TFTP_CHAN	1	Maximum number of TFTP channels
SC_MAX_HTTP_CHAN	1	Maximum number of HTTP channels
SC_MAX_PIPE_CHAN	1	Maximum number of PIPE channels

Table 5-15 contains the Memory Management Library configuration parameter.

TABLE 5-15 Memory Management Library Configuration Parameters

Parameter	Value
SC_MMULIB	NO

The SC_MMULIB parameter controls the MMU library callouts when pROBE+ is entered. Setting this to YES enables the callouts, while setting this to NO disables the callouts when pROBE+ is entered, and MMU (BSP_MMU in bsp.h) is enabled.

Component Configuration Parameters

The values of many configuration table entries are controlled by #define statements in sys_conf.h. The following subsections list the parameters that #define statements can control. The configuration tables section of this manual describes these parameters. Note that the names of the configuration table entries shown in this section are in lowercase, and the corresponding sys_conf.h parameters are in uppercase. For example, fc_nbuf in the pHILE$^+$ configuration table is controlled by FC_NBUF in sys_conf.h.

Although most of the component configuration table entries are determined by sys_conf.h, others are not because sys_conf.h cannot specify them. For example, kc_code in the pSOS+ configuration table contains the starting address of the pSOS+ kernel, but as this is determined by where the linker places the pSOS+ kernel, sys_conf.h cannot specify the address.

pSOS+ Configuration Table Parameters

Table 5-16 lists the parameters found in `sys_conf.h` that control the values of the corresponding entries in the pSOS+ configuration table.

TABLE 5-16 pSOS+ Configuration Table Parameters

Parameter	Explanation
KC_RN0USIZE	Region 0 unit size (the smallest unit of allocation). (0x100 default)
KC_NTASK	Maximum number of tasks
KC_NQUEUE	Maximum number of message queues
KC_NSEMA4	Maximum number of semaphores
KC_NTIMER	Maximum number of timers
KC_NMUTEX	Maximum number of mutexes
KC_NCVAR	Maximum number of conditional variables
KC_NTVAR	Maximum number of task variables
KC_NCOCB	Maximum number of callouts
KC_NTSD	Maximum number of TSD objects
KC_NLOCOBJ	Maximum number of local objects
KC_NMSGBUF	Maximum number of message buffers
KC_TICKS2SEC	Clock tick interrupt frequency (100 default)
KC_TICKS2SLICE	Time slice quantum, in ticks (10 default)
KC_MAXDNTENT	Maximum number of device names in DNT
KC_DNLEN	Maximum length of a device name in DNT
KC_SYSSTK	pSOS+ system stack size (bytes)
KC_ROOTSSTK	ROOT supervisor stack size
KC_IDLESTK	IDLE stack size
KC_ROOTUSTK	ROOT user stack size

TABLE 5-16 pSOS+ Configuration Table Parameters (Continued)

Parameter	Explanation
KC_ROOTMODE	ROOT initial mode (T_SUPV | T_ISR)
KC_ROOTPRI	ROOT initial priority (230 default)
KC_NIO	Number of devices in initial iojtab (SC_DEVMAX + 1)
KC_MAXIO	Maximum number of devices in the system ((SC_DEVMAX + 3))
KC_FATAL	Fatal error handler address (disabled by default)
KC_STARTCO	Address of user-defined callout at task activation
KC_DELETECO	Address of user-defined callout at task deletion
KC_SWITCHCO	Address of user-defined callout at task switch
KC_IDLECO	IDLE task callout

pSOS+m Configuration Table Parameters

Table 5-17 lists the parameters found in sys_conf.h that control the values of the corresponding entries in the multiprocessor (pSOS+m) configuration table:

TABLE 5-17 pSOS+ Configuration Table Parameters

Parameter	Explanation
MC_NGLBOBJ	Size of global object table on each node
MC_NAGENT	Number of RPC agents in this node
MC_FLAGS	Operating mode flags (SEQWRAP_ON)
MC_ROSTER	Callout address for user roster change (0)
MC_KIMAXBUF	Maximum length of KI packet buffer (100)
MC_ASYNCERR	Error callout address for asynchronous calls (0)

pROBE+ Configuration Table Parameters

Table 5-18 lists the parameters in sys_conf.h that control the values of the corresponding entries in the pROBE+ configuration table.

TABLE 5-18 pROBE+ Configuration Table Parameters

Parameter	Explanation
TD_BRKOPC	Instruction break trap (0)
TD_DBGPRI	Priority of debugger system tasks (244)
TD_ILEV	pROBE+ interrupt mask (MAXILEV << 12)
TD_FLAGS	Initial pROBE+ flag settings (NODOTS_MASK | TD_ILEV | NOUPD_MASK)
TD_DATASTART	pROBE+ data area starting address

For additional information on pROBE+ parameters, see the *pROBE+ User's Guide*.

pHILE+ Configuration Table Parameters

Table 5-19 lists the parameters located in sys_conf.h that control the values of the corresponding entries in the pHILE+ configuration table. The default settings are in parentheses.

TABLE 5-19 pHILE+ Configuration Table Parameters

Parameter	Explanation
FC_LOGBSIZE	Block size with base-2 exponent (9 default)
FC_NBUF	Number of cache buffers (6 default)
FC_NMOUNT	Maximum number of mounted volumes (3 default)
FC_NFCB	Maximum number of opened files per system (10 default)
FC_NCFILE	Maximum number of opened files per task (2 default)
FC_NDNLC	Maximum number of cached directory entries for CD-ROM (0 default)
FC_ERRCO	I/O error call out (no default)

TABLE 5-19 pHILE+ Configuration Table Parameters (Continued)

Parameter	Explanation
FC_DATA	Address of pHILE+ data area or 0 to allocate from region 0
FC_DATASIZE	Size of pHILE+ data area or 0 to allocate from region 0

pLM+ Configuration Parameters

You must include the plm.h header file in your application source code to use pLM+ definitions in your application.

Table 5-20 list the pLM+ configuration parameters.

TABLE 5-20 pLM+ Configuration Parameters

Parameter	Explanation
LM_MAXREG	Maximum number of registered libraries
LM_DATA	pLM+ data area or 0 to allocate from region 0
LM_DATASIZE	pLM+ data area size or 0 to allocate from region 0
LM_DEFAULT_COUTS	Controls pSOSystem default pLM callout
LM_LOADCO	Load callout
LM_UNLOADCO	Unload callout

Set LM_DEFAULT_COUTS to YES if you need to use the pSOSystem default pLM call-outs. Set it to NO if you are providing the callouts. Setting LM_DEFAULT_COUTS to NO needs the LM_LOADCO and LM_UNLOADCO definitions to be valid function addresses.

If the application is going to use the pSOSystem default call outs, set LM_LOADCO and LM_UNLOADCO to 0.

User defined call outs must be added to the pSOSystem callouts table using PssRegister_pLM_couts and removed with PssDeregister_pLM_couts.

To have user callouts follow the steps below:

1. In the user application source file add the following lines (these definitions are required by `plmcfg.c`):

```
#include <plm.h>
extern ULONG user_load_co(const char *libname, ULONG scope,
                          ULONG version, const void *libinfo,
                          sl_attrib *attr);
extern ULONG user_unload_co(const sl_attrib *attr);
```

2. In the `sys_conf.h` file set the `LM_LOADCO` and `LM_UNLOADCO` define statements as follows:

```
#define LM_LOADCO      user_load_co
#define LM_UNLOADCO    user_unload_co
```

pREPC+ Configuration Table Parameters

The following parameters located in the `sys_conf.h` header file, and listed in Table 5-21, control the values of the corresponding entries in the pREPC+ configuration table. The default settings are in parentheses.

TABLE 5-21 pREPC+ Configuration Table Parameters

Parameter	Explanation
LC_BUFSIZ	I/O buffer size (1 << FC_LOGBSIZE)
LC_NUMFILES	Maximum number of open files per task (5)
LC_WAITOPT	Wait option for memory allocation (0)
LC_TIMEOPT	Timeout option for memory allocation (0)
LC_STDIN	Default standard in device
LC_STDOUT	Default standard out device
LC_STDERR	Default standard error device
LC_TEMPDIR	Default `TEMPDIR` device

pNA+ Configuration Table Parameters

The following parameters located in the sys_conf.h header file, and listed in Table 5-22, control the values of the corresponding entries in the pNA+ configuration table. The default settings are in parentheses.

TABLE 5-22 pNA+ Configuration Table Parameters

Parameter	Explanation
NC_NNI	Size of pNA+ Network Interface (NI) table (5)
NC_NROUTE	Size of pNA+ routing table (10)
NC_NARP	Size of pNA+ ARP table (20)
NC_DEFUID	Default User ID of a task, used for NFS (0)
NC_DEFGID	Default Group ID of a task, used for NFS (0)
NC_HOSTNAME	Host name of the node ("scg")
NC_NHENTRY	Number of host table entries (8)
NC_NMCSOCS	Number of IP multicast sockets (0)
NC_NMCMEMB	Number of distinct IP multicast group memberships per interface (0)
NC_NNODEID	Network NODE ID for unnumbered link (0)
NC_NSOCKETS	Sockets in the system(4)
NC_NDESCS	Socket descriptors or tasks (4)
NC_MBLKS	Message blocks in the system (300)
NC_BUFS_0	Number of 0-length buffers (64)
NC_BUFS_32	Number of 32-byte buffers (0)
NC_BUFS_64	Number of 64-byte buffers (0)
NC_BUFS_128	Number of 128-byte buffers (256)
NC_BUFS_256	Number of 256-byte buffers (0)
NC_BUFS_512	Number of 512-byte buffers (0)
NC_BUFS_1024	Number of 1-Kbyte buffers (16)

TABLE 5-22 pNA+ Configuration Table Parameters (Continued)

Parameter	Explanation
NC_BUFS_2048	Number of 2-Kbyte buffers (48)
NC_BUFS_4096	Number of 4-Kbyte buffers (0)
NC_MAX_BUFS	Maximum number of NC_BUFS types (9)
NC_PNAMEM_NEWSCHEME	Apply new pNA+ memory management scheme (YES)
NC_BUFS_XX_INTERNAL	Number of buffers for internal pNA+ usage (20 of 128-byte buffers)
NC_MBLKS_INT_PERCENT	Percentage of mblks reserved for internal pNA+ usage (15)
NC_MBLKS_TX_PERCENT	Percentage of mblks reserved for transmission memory pool (40)
NC_BUFS_0_TX_PERCENT	Percentage of 0 byte buffers used for transmission (50)
NC_BUFS_32_TX_PERCENT	Percentage of 32 byte buffers used for transmission (50)
NC_BUFS_64_TX_PERCENT	Percentage of 64 byte buffers used for transmission (50)
NC_BUFS_128_TX_PERCENT	Percentage of 128-byte buffers used for transmission (50)
NC_BUFS_256_TX_PERCENT	Percentage of 256-byte buffers used for transmission (50)
NC_BUFS_512_TX_PERCENT	Percentage of 512-byte buffers used for transmission (50)
NC_BUFS_1024_TX_PERCENT	Percentage of 1-Kbyte buffers used for transmission (50)
NC_BUFS_2048_TX_PERCENT	Percentage of 2-Kbyte buffers used for transmission (50)
NC_BUFS_4096_TX_PERCENT	Percentage of 4-Kbyte buffers used for transmission (50)
NC_DATA	pNA+ data area start address (0)

TABLE 5-22 pNA+ Configuration Table Parameters (Continued)

Parameter	Explanation
NC_DATASIZE	pNA+ data area size
NC_DTASK_SSTKSZ	pNAD daemon task supervisor stack size (0x800)
NC_DTASK_USTKSZ	pNAD daemon task user stack size (0x400)
NC_DTASK_PRIO	pNAD daemon task priority (255)
NC_NEW_MULTITASK_SYNC	Deploy pNA+ sync scheme using locks (YES)
NC_USE_MUTEX	Use pSOS mutex primitive (YES)
NC_SIGNAL	pNA+ signal handler (0)

pRPC+ Configuration Parameters

The following parameters located in the `sys_conf.h` header file, and listed in Table 5-23, control the values of the corresponding entries in the pRPC+ configuration table:

TABLE 5-23 pRPC+ Configuration Parameters

Parameter	Explanation
NR_PMAP_PRIO	Task priority (254)
NR_PMAP_SSTACK	Supervisor stack size (0x2000)
NR_PMAP_USTACK	User stack size (0x100)
NR_PMAP_FLAGS	`t_create` flags (T_LOCAL default)
NR_PMAP_MODE	`t_start` mode (T_SUPV default)
NR_DEBUG_FLAG	Turn on debug messages from PMAP task (NO)
NR_GETHOSTNAME	Function to get local host name (`nr_gethostname`)
NR_GET_HENTBYNAME	Function to map host name to its IP address (`nr_get_hentbyname`)

TABLE 5-23 pRPC+ Configuration Parameters (Continued)

Parameter	Explanation
NR_DATA	pRPC+ data area start address (0)
NR_DATASIZE	pRPC+ data area size (0)

pSE+ Configuration Parameters

The parameters in sys_conf.h control the values of the corresponding entries in the configuration table for the pSE+ streams component. See *OpEN User's Guide* for additional information.

Loader Configuration Parameters

The following parameters located in the sys_conf.h header file, and listed in Table 5-24, control the configuration for Loader. See the System Services section of this manual for more information on Loader.

TABLE 5-24 Loader Configuration Parameters

Parameter	Explanation
LD_MAX_LOAD	Maximum number of active loads (8 default)
LD_ELF_MODULE	Load ELF object-load-module
LD_SREC_MODULE	Load SREC object-load-module
LD_COFF_MODULE	Load ELF object-load-module
LD_IEEE_MODULE	Load IEEE object-load-module
LD_IHEX_MODULE	Load IHEX object-load-module

pMONT+ Configuration Parameters

Table 5-25 lists the parameters located in the `sys_conf.h` header file that control pMONT+ configuration.

TABLE 5-25 pMONT+ Configuration Parameters

Parameter	Explanation
`PM_CMODE`	pMONT+ communication mode, 1 = networking, 2 = serial
`PM_DEV`	Major or minor device number for serial channel if used
`PM_BAUD`	Baud rate for serial channel
`PM_TRACE_BUFF`	If 0, the address of trace buffer is allocated by pSOSystem
`PM_TRACE_SIZE`	Size of trace buffer
`PM_TIMER`	Second timer for finer timing within data collection

General Serial Block Configuration Parameters

The parameters listed in Table 5-26 are located in the `sys_conf.h` header file, which control the allocation of buffers for the *General Serial Block* buffer manager. These buffers are used by various drivers that use the Device Independent Serial Interface (DISI). See Chapter 2, *Interfaces*, for information on DISI.

TABLE 5-26 General Serial Block Configuration Parameters

Parameter	Explanation
`GS_BUFS_0`	Number of 0-length buffers
`GS_BUFS_32`	Number of 32-byte buffers
`GS_BUFS_64`	Number of 64-byte buffers
`GS_BUFS_128`	Number of 128-byte buffers
`GS_BUFS_256`	Number of 256-byte buffers
`GS_BUFS_512`	Number of 512-byte buffers
`GS_BUFS_1024`	Number of 1K-byte buffers
`GS_BUFS_2048`	Number of 2K-byte buffers

TABLE 5-26 General Serial Block Configuration Parameters (Continued)

Parameter	Explanation
`GS_BUFS_4096`	Number of 4K-byte buffers
`GS_MBLKS`	Number of message blocks
`SE_MAX_GS_BUFS`	Maximum number of stream buffer types

The `GSblkSetup` routine sets up the Global Serial message and data blocks, using the values set by `#define` statements in the `sys_conf.h` file.

Each application may have a different need for a particular size buffer, which depends on the size of the message the application sends.

The parameter `GS_MBLKS` controls the number of message block headers allocated by the `GSblkSetup` routine.

Set `GS_MBLKS` to the total number of data blocks represented by the total number of buffers allocated in the preceding list. If the driver attaches user supplied data buffers to a message block header, add an additional amount for these as necessary. For example, the pSOSystem terminal driver needs additional message block headers for user supplied buffers.

NOTE: Although the code can use a message block that does not have a data buffer attached to it, the serial drivers currently have no use for a header only message block.

TCP/IP for OpEN Configuration Parameters

The parameters located in the `sys_conf.h` header file control the configuration values for TCP/IP OpEN. See *TCP/IP for OpEN User’s Guide* for additional information.

Adding Drivers to the System

To add drivers to the pSOS+ I/O jump table, edit `drv_conf.c` in the application directory. This file contains a function called `SetUpDrivers()`, which calls `InstallDriver()` to install the driver into the I/O jump table.

If you want to add a driver, you should add an `InstallDriver()` call into `SetUpDrivers()` function. The parameters passed to `InstallDriver()` are the major device number, the pointers to the `init`, `open`, `close`, `read`, `write`, and `ioctl` functions of the driver, and the two reserved entries in the pSOS+ I/O switch table. For example, to add a driver with major device 6 (which processes only the `init` and `read` calls), you add the following to `SetUpDrivers()` in `sysinit.c`:

```
InstallDriver(6, DriverInit, NULLF, NULLF, DriverRead, NULLF,
                   NULLF, 0, 0);
```

where `NULLF` is a macro in `sysinit.c` defined as a NULL function pointer.

Network interfaces are added in a manner similar to that of pSOS+ drivers. The source file, `drv_conf.c`, also contains a routine called `SetUpNi()`, which calls `InstallNI()` to install each network interface in the pNA+ initial interface table.

To add a Network driver to pSOSystem, call `InstallNi()` function with the arguments described below. This adds the Network Interface to the pNA+ Network Interface table. The NI entry added using InstallNI() into the system is static, and must be done only during system startup. The `InstallNi()` function takes the following arguments:

`int (*entry)();`	address of NI entry point
`int ipadd;`	IP address
`int mtu;`	maximum transmission length
`int hwalen;`	length of hardware address
`int flags;`	intErface flags
`int subnetaddr;`	subnet mask
`int dstipaddr;`	destination ip address

Refer to Chapter 6, *Configuration Tables*, for more information about the Network Interface table.

pMONT+ Driver Usage

When pMONT+ is configured into the system, the drivers required by it are initialized during system startup (auto-initialized). Note that if you enable auto-initialization for a particular device, the pSOS+ kernel calls the `de_init()` function of the driver with a minor device number of 0 first and does so before any task starts running.

To use auto-initialization, the `SC_AUTOINIT` `#define` statement must be passed as the last argument in the install driver. The following example shows the pSOSystem convention for installing a driver in `drv_conf.c` (file residing in each pSOSystem application directory):

```
InstallDriver(SC_DEV_SERIAL, CnslInit, NULL, NULL, CnslRead,\
CnslWrite, CnslCntrl, 0, SC_AUTOINIT;
```

For pMONT+ to run successfully, you must use auto-initialization to initialize the timer device. For serial communication, you must also initialize the serial driver, which then operates with the following characteristics:

- Blocking I/O
- ASCII mode
- Echoing off
- Carriage return to signal the end of a record
- No conversions for a new-line character

pMONT+ uses the serial driver through the pSOS calls `de_open()`, `de_read()`, and `de_write()`. It makes the `de_open()` call before proceeding to use the driver to make read and write calls. The `de_open()` call should thus set the driver for pMONT+ usage if auto-initialization has not been done. We recommend using auto-initialization to initialize the driver and the `de_open()` call to change settings (if needed). In cases where the installed driver has specific functionality for each of the I/O calls, a dummy driver could be installed for pMONT+ in which the `de_open()` calls the actual serial driver to perform required initialization not done by the auto-initialization function.

Customizing the System Startup Sequence

The startup code supplied with the pSOSystem software, performs the required initialization and startup. The topics in this section are important only if you want to customize the pSOSystem startup code or write your own startup code. This section documents the initialization and startup requirements of the individual software components.

The following checklist shows the required items for a typical pSOSystem based system:

- An optional user supplied boot module to perform power on or reset initialization and self test
- The pSOS+ kernel and other appropriate components (for example, the pROBE+ debugger and the pHILE+ file manager)
- The Node Anchor and node configuration table
- Configuration tables for the pSOS+ kernel and other installed modules
- Task, ISR, and device driver code and initialized data, if any for the application
- Exception vectors with the correct settings

In a ROM based system, most of these items reside in ROM. For a RAM based system or a system that is under test or integration, some of the items may be loaded into RAM either by the user supplied boot module or the pROBE+ System Debug/Analyzer. Furthermore, in a memory mapped system, it is possible to load the application tasks or drivers dynamically at run time.

Figure 5-2 on page 5-35 shows the possible startup sequences.

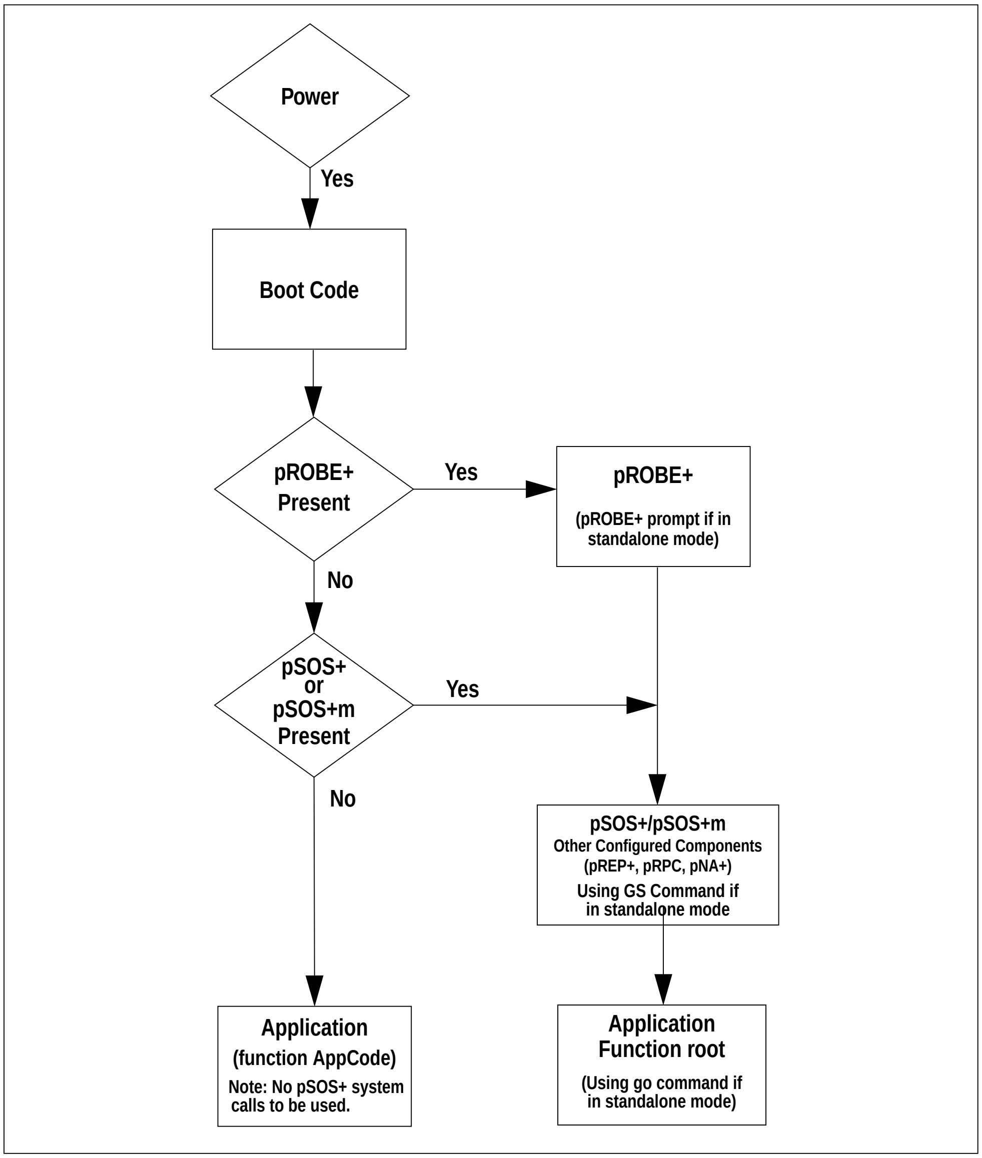

Power
Yes
Boot Code
pROBE+ Present
Yes
pROBE+
(pROBE+ prompt if in standalone mode)
No
pSOS+ or pSOS+m Present
Yes
No
pSOS+/pSOS+m
Other Configured Components
(pREP+, pRPC, pNA+)
Using GS Command if in standalone mode
Application
(function AppCode)
Note: No pSOS+ system calls to be used.
Application Function root
(Using go command if in standalone mode)

FIGURE 5-2 System Startup Sequences

Typically, applying power to the board or resetting the board passes control to the boot code in ROM, which performs any necessary initialization and self test of the board. It should also set up the required configuration environment for the software components in the system. The configuration environment consists of:

- The Node Anchor,
- The Node Configuration Table and,
- Other component configuration tables.

You can set up this environment as a single process or allow the startup sequence to proceed incrementally. During debug, for example, you might set up the configuration environment just enough to let the pROBE+ debugger take control and execute. If the pROBE+ debugger is present, the boot code passes control to it. The pROBE+ debugger then passes control to the pSOS+ component to initialize pSOS+/pSOS+m and other configured components, and starts execution of the pSOS+ application. You can start the pSOS+ kernel in one of the following ways:

- By passing control to the pSOS+ startup entry, as follows:

  branch to `START_of_PSOS` + 0x40

  NOTE: This assumes the CPU is in the supervisor state. If this is not the case, you must set the system call vector, to point to this pSOS+ startup address and use the `SC` instruction to enter it.

- From the pROBE+ debugger, enter the `gs` command (stand alone mode).
- Specify the *silent* startup mode in the pROBE+ configuration table, so that the pROBE+ debugger initializes itself then passes control to pSOS+ startup without stopping.

Upon entry, pSOS+ startup first uses the Node Anchor to locate the pSOS+ configuration table. The pSOS+ kernel then takes a segment of memory from the beginning of memory Region 0 for its data area. Within this area, it uses the bottom part for its key data structures. The next memory segment contains the system stack. Above the system stack, the pSOS+ kernel builds the required number of TCBs, QCBs, SCBs, MGBs, TMCBs, and the Object Tables.

Next, the pSOS+ kernel checks the Node configuration table. If other runtime components are present, the pSOS+ kernel locates and automatically calls the start up functions for those components to allow them to set up and initialize.

The last step in pSOS+ startup is to create and activate the IDLE system daemon task and the user ROOT task. The pSOS+ startup function then dispatches ROOT, which takes over and executes.

NOTE: The pSOS+ kernel treats any error it encounters during startup as a fatal error. For a description of fatal error handling, see *pSOSystem System Concepts* manual. Startup error codes and their meanings are described in this manual. For a complete description of the startup flow and configuration refer to the *pSOSystem Advanced Topics* manual.

6 Configuration Tables

The pSOSystem components configure themselves at system 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. See Chapter 5, *pSOSystem Configuration File*, for details on 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.

This chapter describes the structures for the following Configuration Tables:

- Node (See page 6-4)
- Multiprocessor (See page 6-6)
- pSOS+ (See page 6-10)
- pROBE+ (See page 6-17)
- pHILE+ (See page 6-23)
- pREPC+ (See page 6-28)
- pNA+ (See page 6-32)
- pMONT+ (See page 6-43)
- pRPC+ (See page 6-45)

- pLM+ (See page 6-30)
- pSE+ (See *OpEN User's Guide*)
- pSKT+ (See *OpEN User's Guide*)
- pTLI+ (See *OpEN User's Guide*)

The structure definitions for these configuration tables reside in the `include` 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 through the *Node Anchor*, which is the only 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 6-1 on page 6-3 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.

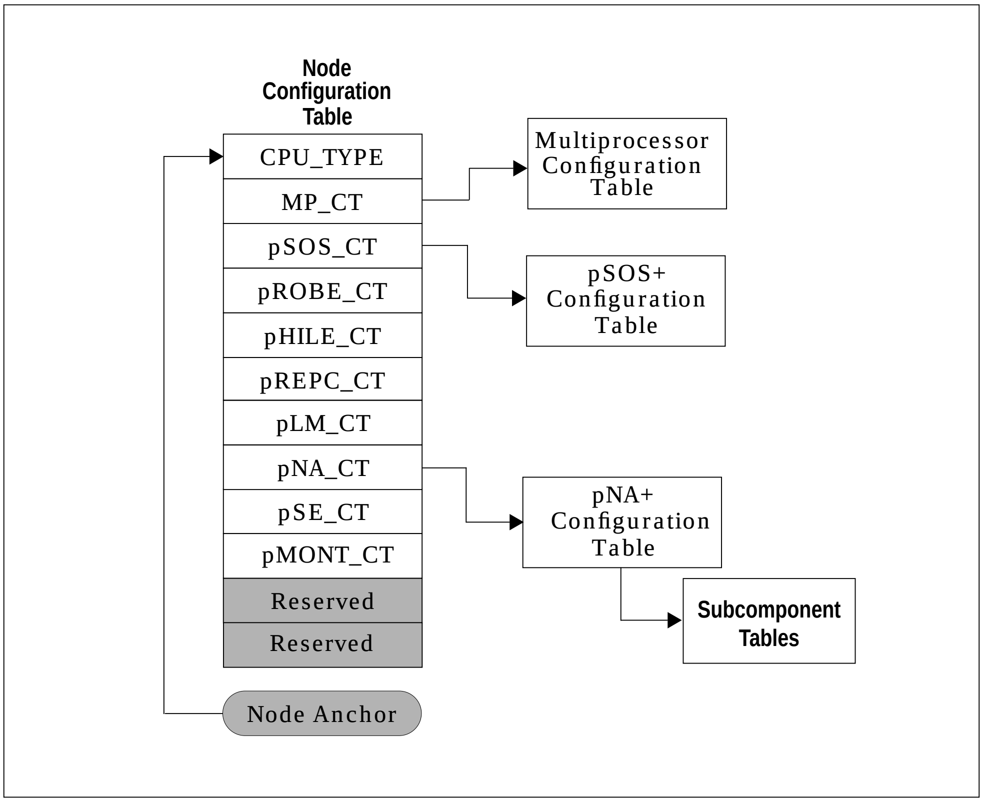


FIGURE 6-1 Node Configuration Table

Node

```
ttypedef struct NodeConfigTable
     {
     unsigned long                cputype;/* CPU type                  */
     struct MultiProcConfigTable  *mp_ct;  /* ptr to Multi-proc cfg tab */
     struct pSOSConfigTable       *psosct; /* ptr to pSOS+ config table */
     struct pROBEConfigTable      *probect;/* ptr to pROBE+ cfg table   */
     struct pHILEConfigTable      *philect;/* ptr to pHILE+ cfg table   */
     struct pREPCConfigTable      *prepct; /* ptr to pREPC+ cfg table   */
     struct pLMConfigTable        *plmct;  /* ptr to pLM+ config table  */
     struct pNAConfigTable        *pnact;  /* ptr to pNA+ config table  */
     struct pSEConfigTable        *psect;  /* ptr to pSE+ config table  */
     struct pMONTConfigTable      *pmct;   /* ptr to pMONT+ cfg 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:**	
	<u>BITS</u>	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 = PPC601
		1 = PPC603
		2 = PPC604
		3 = PPC603e

cputype	**CPU type and has the following meaning:**	
		4 = PPC403GA
		5 = PPC821
		6 = PPC604p, PPC604e
		7 = PPC603p
		8 = PPC860
		9 = PPC604r
		10 = MPC740, PPC740
		11 = MPC750, PPC750
		12 = MPC505, PPC505

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.
`probect`	Starting address of the pROBE+ Configuration Table: it should be 0 if pROBE+ is not installed.
`philect`	Starting address of the pHILE+ Configuration Table: it should be 0 if pHILE+ is not installed.
`prepct`	Starting address of the pREPC+ Configuration Table: it should be 0 if pREPC+ is not installed.
`plmct`	Reserved for future use and should be set to 0.
`pnact`	Starting address of the pNA+ Configuration Table: it should be 0 if pNA+ is not installed.
`psect`	Starting address of the pSE+ Configuration Table: it should be 0 if pSE+ is not installed.
`pmct`	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.

Multiprocessor

```
typedef struct MultiProcConfigTable
     {
     ULONG mc_nodenum;            /* this node’s node number */
     void  (*mc_kicode)();        /* addr of this node’s kernel interface */
     ULONG mc_nnode;              /* max number of nodes in system */
     ULONG mc_nglbobj;            /* size of global obj table in each node */
     ULONG mc_nagent;             /* number of RPC agents in this node */
     ULONG mc_flags;              /* operating mode flags */
     void  (*mc_roster)();        /* address of user roster change callout */
     void *mc_dprext;             /* dual-port RAM external starting address */
     void *mc_dprint;             /* dual-port RAM internal starting address */
     ULONG mc_dprlen;             /* dual-port RAM length in bytes */
     ULONG mc_kimaxbuf;           /* maximum KI packet buffer length */
     void  (*mc_asyncerr)();      /* asynchronous calls error callout */
     ULONG 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 `mp_ct` 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 Power PC calling conventions are used for the call. The syntax of the call is as follows:

```
void rst_change (unsigned long change, void *roster,
unsigned long parm1, unsigned long parm2, unsigned
long parm3);
```

Change = Tyoe of Change

0	This is the initial roster. roster points to the internal pSOS+m roster.
1	A node has joined. parm1 and parm2 contain, respectively, the node number and sequence number of the new node.
2	A node has failed. parm1, parm2, and parm3 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,
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 Chapter 2, *Interfaces*, for a description of the this value. Also, note the following: 1. As explained in Chapter 2, *Interfaces*, 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.

pSOS+

```
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;               /* max number of tasks */
   unsigned long kc_nqueue;              /* max number of message queues */
   unsigned long kc_nsema4;              /* max number of semaphores */
   unsigned long kc_nmsgbuf;             /* max number of message buffers */
   unsigned long kc_ntimer;              /* max number of timers */
   unsigned long kc_nlocobj;             /* max 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;                 /* num of I/O devices in system */
   struct iojent *kc_iojtable;           /* addr of I/O switch table */
   unsigned long kc_sysstk;              /* pSOS+ system 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 */
   unsigned long kc_reserved3;           /* reserved */
   unsigned long kc_idlestk;             /* IDLE task stack size */
   unsigned long kc_rootpri;             /* ROOT task priority */
   unsigned long kc_nmutex;              /* # of mutexes */
   unsigned long kc_ncvar;               /* max number of condition variables */
   unsigned long kc_maxio;               /* maximum number of I/O Devices */
   unsigned long kc_ntvar;               /* max number of task variable */
   unsigned long kc_maxdntent;           /* max number of device names in DNT */
   unsigned long kc_dnlen;               /* max length of device name in DNT */
   unsigned long kc_ntsd;                /* max number of task-specific-data
                                            entries */
   void          *kc_tsdanchor;          /* Address of task-specific-data
                                            anchor */
   unsigned long kc_maxscmajor;          /* max major no. of service call
                                            extension */
   unsigned long kc_ncocb;               /* max number of callouts */
   void         (*kc_sysstartco)();      /* System startup callout */
   unsigned long kc_reserved2[10];       /* 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.

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 `t_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.

kc_switchco

4. 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.
5. 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.

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:
kc_idleco	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. 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_idlestk	Define the size (in bytes) of the IDLE task's stack.
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_nmutex	Defines the number of Mutex Control Blocks that will be statically pre-allocated by pSOS+ at startup.
kn_ncvar	Defines the number of conditional variables that will be statically pre-allocated by pSOS+ at startup.
kc_maxio	Defines the maximum number of IO devices in the system. It corresponds to the size of the IO Switch table, pre-allocated by pSOS+ at startup.
kc_ntvar	Defines the number of Task Variable Control Blocks that will be statically pre-allocated by pSOS+ at startup.

kc_maxdntent	Defines the number of DNT entries that will be statically pre-allocated by pSOS+ at startup, to construct the pSOS+ Device Name Table.
kc_dnlen	Defines the maximum length of a device name in the pSOS+ DNT. This is used by pSOS+ to calculate the size of a DNT entry when pre-allocating the DNT at startup.
kc_ntsd	Defines the number of task specific data (TSD) objects control blocks that will be statically pre-allocated by pSOS+ at startup.
kc_tsdanchor	Defines the address of the global task specific data (TSD) anchor which will be stored internally by pSOS+.
kc_maxscmajor	The value of the maximum major number (a value between 16 and 255, inclusive) of a service call extension table.
kc_ncocb	Defines the number of Callout Control Blocks that will be statically pre-allocated by pSOS+ at startup.
kc_sysstartco	Supplies the address of the system startup callout routine that is called in pSOS+ before tasking begins (only IDLE task has been created and started). This callout is provided so that system initialization can be done by various driver, bsps, libraries, etc, in case of a *warm restart* of the system. The callout function is defined in `<sysinit.c>` as the `PssSysStartCO()` function. It in turn invokes the callout function provided by the various components and the `DrvSysStartCO()` function in `<drv_conf.c>`.
kc_reserved2	Should be all zeros for upward compatibility.

pROBE+

```
typdef struct pROBEConfigTable {
   void (*td_code) ( );             /* Address of pROBE+ code module */
   unsigned long *td_data;          /* Address of pROBE+ data area */
   unsigned long *td_stack;         /* Top address of pROBE+ stack area */
   void (*td_ce_code) ( );          /* Address of console interface executive */
   void (*td_rd_code) ( );          /* Address of debug interface executive */
   void (*td_qs_code) ( );          /* pSOS+ component query */
   void (*td_ds_code) ( );          /* Disassembler component */
   void (*td_reserved1[4] ( );      /* Each must contain 0's (4 words long) */
   unsigned long td_brkopc;         /* Instruction break opcode */
   unsigned long td_flags;          /* Initial flag settings */
   unsigned long td_dbgpri;         /* Priority of debugger tasks */
   long (*td_drv0) ( );             /* Communications driver 0 */
   long (*td_drv1) ( );             /* Communications driver 1 */
   void (*td_statechng) (ULONG)     /* Debugger state change callout */
   long (*td_urcom) ( );            /* Address of URCOM */
   long (*td_urwrite) ( );          /* Controls data transfer to application */
   long (*td_cacflsh) ( );          /* Cache flush routine */
   unsigned long td_reserved2[5];   /* Each must contain 0's (6 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:

The user-supplied pROBE+ Configuration Table specifies hardware and application-specific parameters that the pROBE+ debugger uses. This table can be anywhere in RAM or ROM. Its starting address is specified by the `nc_probect` entry in the Node Configuration Table. The C language template for the pROBE+ Configuration Table is in `include/probecfg.h`.

NOTE: For a more detailed explanation of the pROBE+ Configuration Table entries, refer to the *pROBE+ User's Guide*.

Definitions for the pROBE+ Configuration Table entries are as follows:

td_code	Contains the starting address of the pROBE+ code.
td_data	Defines the lowest RAM address that the pROBE+ debugger uses for data. This field should be 0 for processor types that require a static linked data area for components.
td_stack	Used to define the beginning RAM address that the pROBE+ debugger uses for stack space. If interrupt activity can occur while the pROBE+ debugger is running, the stack size must be large enough to accommodate the worst-case stack requirements for all nesting ISRs.
td_ce_code	Contains the address of the Console and Debug Interface Executives.
td_rd_code	Required by the user for the debug environment. The appropriate executive is selected based on the RBUG bit in `td_flags`.
td_qs_code	Contains the address of the pSOS+ Query Services module of the target debugger. If no query services are installed, `td_qs_code` must be 0.
td_ds_code	Contains the address of the pSOSystem or user-supplied disassembler.

The pROBE+ debugger passes a pointer to the instruction to be disassembled (`inst_buf`) and a pointer to a 64-byte character buffer (`dis_buf`). `td_ds_code` places a string containing the disassembled instruction in `dis_buf` and returns the length of the string.

```
long (*td_ds_code)(inst_buf, dis_buf)
OPCODE_SIZE *inst;
char *dis_buf;
```

reserved1[4]	Must be set to 0.
td_brkopc	Must be 0.
td_flags	Specifies the initial settings for the pROBE+ flags. Unused bits should be set to 0. One bit in `td_flags` corresponds to each pROBE+ flag, as shown in the following table:

TD_FLAGS	Bit	Meaning
RBUG:	0	0 = Disabled
		1 = Enabled
Reserved:	1	0 = Disabled
		1 = Enabled
NODOTS:	2	0 = Disabled
		1 = Enabled
NOMANB:	3	0 = Disabled
		1 = Enabled
NOPAGE:	4	0 = Disabled
		1 = Enabled
PROFILE:	5	0 = Disabled
		1 = Enabled
NOTUPD	6	0 = Disabled
		1 = Enabled
Reserved:	7	0 = Disabled
		1 = Enabled
SMODE:	8	0 = Disabled
		1 = Enabled
Reserved:	9	0 = Disabled
		1 = Enabled
Reserved:	10	0 = Disabled
		1 = Enabled
Reserved:	11	0 = Disabled
		1 = Enabled

TD_FLAGS	Bit	Meaning
ILEVEL:	12	0 = Disabled
		1 = Enabled
Reserved:	13	0 = Disabled
		1 = Enabled
Reserved:	14	0 = Disabled
		1 = Enabled
Bits 15 - 31 are reserved.		

If you want to initialize the NOMANB and NOPAGE bits to be on, for example, set `td_flags = 0x00000018`. The default state for all flag settings is off. So, unless you want one or more to start in the on state, set `td_flags = 0`. You can interactively change any flag with the pROBE+ FL (flag) command. For a description of each flag and the FL command, refer to the *pROBE+ User's Manual*.

dbgpri Defines the priorities for the four debugger tasks: the manager, input, output, and the command processor. This entry sequences the priorities of the four tasks by assigning values, with the lowest number assigned to the lowest priority task. If `dbgpri` is zero, all debugger tasks take the default values for their priorities. If it is non zero, `dbgpri` specifies the highest priority of the debugger tasks - the priority of the manager task. The priorities of other three tasks are dbgpri-1, dbgpri-2, and dbgpri-3.

This entry is used primarily for remote debugging.

td_drv0
td_drv1 Contains the address of the Interface Communications Drivers used by the Executives defined in `td_rd_code` and `td_cd_code`. The drivers have a common interface, as follows:

```
long (*rc_drv)(mode, byte_cnt, MsgBuffer)
unsigned long mode; /* INIT, GET, PUT, STATUS */
long *byte_cnt;     /* Number of bytes to transfer
                       or */
                   /* residing in buffer */
char *MsgBuffer;   /* Pointer to buffer with
                      message */
```

td_statechng Contains the address of an optional user supplied routine that is called whenever a debugger state change occurs. A debugger state change occurs when the debugger needs complete access to an aspect of system control. Currently, the pROBE+ debugger makes this callout only when it assumes or releases control over either, or both, of two system aspects: the MMU and pSOS+ tasking. The following bits are passed as an argument to indicate which state change(s) is taking place:

```
#define MMU_NORM         0x1
#define MMU_PROBE        0x2
#define TASKING_NORM     0x4
#define TASKING_PROBE    0x8
void (*td_statechng)(ULONG flags);
```

pROBE+ expects no return value from `td_statechng()`.

td_urcom Specifies the address of an optional, user-supplied procedure. The pROBE+ debugger calls the procedure when it encounters an unrecognized command and thus allows you to extend the pROBE+ command set. If no user-supplied procedure is installed, `td_urcom` must be 0.

The interface to this procedure is as follows:

```
long (*td_urcom)(cmd_ptr)

char *cmd_ptr;
```

RETURNS: 0 = command accepted, -1 = command unrecognized

td_urwrite

Contains an optional, user-supplied procedure that controls data transfers to and from application program space memory. This procedure's usage includes the following:

- Setting and resetting breakpoints
- Patching memory
- Downloading code

It is called by the following interface:

```
long (*td_write)(address,byte_cnt, bufptr, nbytes)
unsigned long address; /* where to write to */
long byte_cnt;         /* how many bytes to write */
char *bufptr;          /* what to write */
unsigned long *nbytes; /* number of bytes actually
                          written */
```

RETURNS: 0 = buffer written correctly, -1 = error occurred on write.

If no user-supplied procedure is installed, `td_urwrite` must be 0.

td_cacflsh

Contains an optional, user-supplied procedure that is called whenever the pROBE+ debugger has modified the application program space. The interface to this procedure is as follows:

```
void (*td_cacflsh)(addr, num0fBytes);

void*addr:      /* the start address of the memory
                   area to be flushed and
                   invalidated */

unsigned long:  /* the size of the memory area in
                   the unit of byte */
```

This function should flush the data cache and invalidate the instruction cache of the specified memory area. If `num0fBytes` is zero, it should flush the whole data cache and invalidate the whole instruction cache.

reserved2[5]

Must be set to 0.

pHILE+

```
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 # opened files per system */
  unsigned long fc_ncfile;          /* Max # opened files per task */
  unsigned long fc_ndnlc;           /* Max # cached dir. entries */
  unsigned long res[2];             /* Must be 0 */
  pHILE_SCT *fc_sct;                /* add of sub-component cfg. tables */
  pHILE_err_call_out_t fc_errco;    /* Error call-out */
  unsigned long res2[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 two aspects of the block size for pHILE+ formatted volumes. First, it defines the block size for all pHILE+ format volumes initialized by `init_vol()`. Second, it defines the maximum block size pHILE+ format volume that can be mounted by `mount_vol()`. Unlike pHILE+ version 2.x.x and 3.x.x, mounted pHILE+ format volumes can have different block sizes. The value is adjusted for the second purpose, but not for the first, as follows. If the value is smaller, the value is increased to 512 byte block size, and 2048 byte block size if MS-DOS FAT file system format, and CD-ROM file system format, respectively, are included.

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` is 8 through 15 (the smallest possible block size is 256 bytes, and the largest is 32 Kbytes.) The disk that contains the volume must have a logical block size of less than or equal to the pHILE+ volume block size. Most disks in use today have a logical block size of 512. Therefore on most disks the range is 9 through 15, i.e. 8 can not be used.

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 configured or not less than 2048 if CD ROM volumes are configured. The minimum number of cache buffers needed to ensure proper operation of pHILE+ is given in the table below. Add up all rows for the final number of cache buffers needed. If less than this are configured, deadlock may occur within pHILE+.

File Format	Cache Buffers Needed
pHILE+	# of mounted pHILE+ volumes + (2 x number of concurrent tasks using pHILE+ volumes)
MS-DOS FAT	(2 x # mounted MS-DOS FAT volumes) + 1
CD-ROM	2 x number of concurrent tasks using CD-ROM volumes
NFS Client	0

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_ndnlc	Defines the number of CD-ROM directory entries that can be cached. This speeds up file name processing for CD-ROMs. This parameter is ignored if CD-ROM file system format is excluded. If CD-ROM is included and this parameter is 0, it is changed to 2 x `fc_nmount`.
res[2]	Should be 0 for upward compatibility.

fc_sct

Pointer to a table that contains pointers to configuration tables for pHILE+ subcomponents. The table is defined as follows:

```
typedef struct pHILESubCompTables
{
  pHILE_ST *fc_phile;   /* pHILE+ (pHILE+ real-time
                          file sys) */
  pHILE_ST *fc_msdos;   /* pHILE+ (MS-DOS FAT file
                           system) */
  pHILE_ST *fc_cdrom;   /* pHILE+ (ISO 9660 CD-ROM
                           file sys) */
  pHILE_ST *fc_nfs;     /* pHILE+ (NFS client) */
  pHILE_ST *res[6];     /* Must be 0 */
} pHILE_SCT;
```

The parameter definitions are as follows

`fc_phile`	Points to the pHILE+ file system format configuration table.
`fc_msdos`	Points to the MS-DOS FAT file system format configuration table.
`fc_cdrom`	Points to the CD-ROM ISO 9660 file system format configuration table.
`fc_nfs`	Points to the NFS client configuration table.

For now, the structure of all the above subcomponent configuration tables are the same. They contain only a pointer to the subcomponent code, i.e. `fcs_code`. However, there are other fields which should all be set to zero for upwards compatibility.The structure of all the subcomponent tables follows:

```
typedef struct pHILESubconfigTable
{
  void (*fcs_code)();         /* Subcomponent code
                                 address */
  void *fcs_data;             /* Address of sub-
                                 component data area */
  unsigned long fcs_datasize; /* Size of subcomponent
                                 data area */
  unsigned long res[10];      /* Must be 0 */
} pHILE_ST;
```

fc_errco	Defines the I/O call out routine. If nonzero pHILE+ calls this whenever an I/O error occurs after a volume is mounted. The callout routine can retry the operation, or terminate the operation with an error code, either the original I/O error code or a new error code substituted by the callout routine.
res2[5]	Should be 0 for upward compatibility.

pREPC+

```
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;      /* Obsolete */
   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.
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 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	Obsolete. This configuration parameter is no longer used by pREPC+.
reserved	Should be 0 for upward compatibility.

pLM+

Hardware and application specific parameters required by pLM+.

Syntax

```
typedef struct pLMConfigTable
    {
    void (*lm_plm)();               /* Address of pLM+ module */
    void *lm_data;                  /* Address of pLM+ data area */
    unsigned long lm_datasize;      /* Size of pLM+ data area */
    unsigned long lm_maxreg;        /* Max # of registered libraries */
    lm_loadco_t   lm_loadco;        /* sl_acquire() Load call-out */
    lm_unloadco_t lm_unloadco;      /* sl_release() Unload call-out */
    unsigned long res[8];           /* Must be 0 */
    } pLM_CT;
```

Description

The pLM+ Configuration Table is a user supplied table that provides hardware and application specific information required by pLM+. It can reside anywhere in RAM or ROM. The starting address of the pLM+ Configuration Table must be specified as the `plmct` entry in the Node Configuration Table. The C language template for the pLM+ Configuration Table is located in `include/plmcfg.h`.

Definitions for the pLM+ Configuration Table entries are as follows:

lm_plm	Defines the starting address of the pLM+ code.
lm_data	Defines the starting address of the pLM+ data area, which must be located in RAM. You must reserve enough space for the memory requirements of pLM+. If the `lm_data` and `lm_datasize` entries are both 0, pLM+ automatically allocates the required amount of memory from pSOS+ region 0 during initialization. In this case, `lm_data` is ignored. pLM+ calculates the amount of memory it requires by examining entries in its Configuration Table.
lm_datasize	Defines the size of the pLM+ data area. The value of `lm_datasize` depends on various pLM+ Configuration Table entries.

lm_maxreg	Specifies the maximum number of shared libraries that can be registered simultaneously. It defines the number of entries in the registered shared library table.
lm_loadco	Defines the load callout used to load shared libraries. Unlike most pSOSystem callouts, which are optional, this callout is usually required. It is called whenever either acquiring or calling via a stub any shared library that is not already registered, or registering with `sl_register()` any shared library whose dependents are not already registered. It need not necessarily load anything. If the shared library is part of the pSOSystem image, it need only look up the address of the shared library in a table, and return that value. If it does load something the pSOSystem loader or any other loader can be used. (See Chapter 1, *System Services*.)
lm_unloadco	Defines the unload callout used to unload shared libraries. Unlike most pSOSystem callouts, which are optional, this callout is usually required. It is called whenever a shared library is either unregistered with `sl_unregister()` or fully released by all tasks by `sl_release()` and that shared library or one of its dependents has an unloadmode of `SL_AUTOUNLOAD`.
res[8]	Should be 0 for upwards compatibility.

pNA+

```
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*/
   long nc_nmc_socs;                  /* Number of multicast sockets*/
   long nc_nmc_memb;                  /* Number of multicast group
                                         memberships*/
   long nc_nnode_id;                  /* Network node ID or router ID*/
   unsigned long nc_dtask_sstack;     /* pNAD daemon task running stack
                                         size */
   unsigned long nc_dtask_ustksz;     /* pNAD daemon task running stack
                                         size */
   unsigned long nc_dtask_prio;       /* pNAD task running priority level */
   unsigned long nc_new_multitask_sync; /* New multitasking sync scheme */
   unsigned long nc_use_mutex;        /* Use pSOS MUTEX primitive */
}  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 source-level 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
IFF_BROADCAST	0:	0 = Disabled
		1 = Enabled
IFF_NOARP	1:	0 = Enabled
		1 = Disabled
IFF_POINTTOPOINT	4:	0 = Disabled
		1 = Enabled
IFF_UP	7:	0 = Disabled
		1 = Enabled

IFF_PROMISC	8:	0 = Disabled
		1 = Enabled
IFF_MULTICAST	11:	0 = Disabled
		1 = Enabled
IFF_UNNUMBERED	12:	0 = Disabled
		1 = Enabled
IFF_RAWMEM	13:	0 = Disabled
		1 = Enabled
IFF_EXTLOOPBCK	14:	0 = Disabled
		1 = Enabled
IFF_POLL	15:	0 = Disabled
		1 = Enabled
IFF_INITDOWN	16:	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 addr */
 unsigned long gwipadd; /* Gateway internet addr */
 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, RT_MASK, 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. The gateway must be reachable through one of the initial network interfaces.
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{
  long arp_ipadd;   /* Internet addr for NI */
  char *arp_hadd;   /* Hardware addr for NI */
  long 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	signal number
+ 8	tid / 0
+ 12	socket descriptor / interface number

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{
  unsigned 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. This parameter is retained for compatibility and must not be used.

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_bufcfg {
  unsigned long pna_nbuffers; /* No 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. For optimal results it is recommended that the data buffer size be a multiple of 4 bytes.
`reserved`	Used internally by pNA+.

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 `IP_MAX_MEMBERSHIPS` (defined in `pna.h`) group memberships (an internal constant) can be joined per multicast socket. Adding a group membership address that matches an existing group membership address on the same interface is not counted as a new membership and is only the associated reference count of the membership increment.

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.

`nc_dtask_sstksz`	Determines the pNAD daemon task supervisor stack size.
`nc_dtask_ustksz`	Determines the pNAD daemon task user stack size. It is not used and should be set to zero.
`nc_dtask_prio`	Determines the pNAD daemon task running priority level.
`nc_new_multitask_sync`	Defines the new multitasking synchronization scheme. Refer to the *pSOSystem System Concepts* manual for details on various synchronization schemes.
`nc_use_mutex`	Defines the use of the pSOS mutex primitive. Refer to the *pSOSystem System Concepts* manual for details on various locking schemes.

pMONT+

The pMONT+ configuration table is defined in the sys_conf.h file. The sys_conf.h parameter settings become assignments in the following typedef structure located in the pmontcfg.h file:

```
typedef struct
    {
void (* code)();               /* Address of pMONT+ module */
long data;                     /* start of pMONT data */
long dataSize;                 /* size of pMONT data */
long cmode;                    /* comm.mode:NETWORK_TYPE_CONN,PSOSDEV_..*/
long dev;                      /* IO dev maj/minor# in form pSOS expects */
char *traceBuff;               /* Buffer for logging trace events */
long traceBuffSize;            /* trace events buffer size */
unsigned long (* tmFreq)();    /* returns second timer frequency */
void (*tmReset)();             /* resets second timer */
unsigned long (* tmRead)();    /* reads counter value of second timer */
long res1;
long res2;
long res3;
long res4;
}
 pMONT_CT;
```

where the parameters are defined as follows:

code	Starting address of pMONT+ code.
data	Starting address of pMONT+ data area. If data is 0, the data area is allocated from Region 0.
dataSize	The size of the pMONT+ data area. If you specify the address with data, you must also specify dataSize.
cmode	Specifies the communication that pMONT+ uses: ■ cmode=1 means Ethernet communication through the pNA+ network manager. ■ cmode=2 means serial communication through a pSOS+ device.
dev	The pSOS+ I/O major :minor device number if cmode is 2. If cmode is 1, dev is not used.

`traceBuff`	Address of the buffer for logging trace data. If `traceBuff` is 0, `traceBuffSize` defines the size, and the pSOSystem environment supplies the buffer. pMONT+ does not allocate `traceBuff` from Region 0 because the buffer should remain intact. If pMONT+ allocated `traceBuff` from Region 0, system initialization could result in unreliable buffer content.
`traceBuffSize`	The size of `traceBuff` in bytes, 1 kilobyte minimum.
`tmFreq`	Pointer to a user-supplied routine to return the frequency (counts per second) of an extra timer for finer timekeeping during resolution a data collection run.
`tmReset`	Pointer to a user-supplied routine to reset the extra timer and start counting.
`tmRead`	Pointer to a user-supplied routine to return the current count of the timer: the returned count must be between 0 and `tmFreq` and must indicate a sequence counted up from 0. The count must not exceed 24 bits within the span of 1 pSOS+ tick. If you do not use timers and are not running under pSOSystem, then all three of the preceding timer entries must be 0.
`res[0-3]`	An array reserved for pMONT+ use. Each element of `res[]` should be initialized to zeroes (0000).

If you are configuring pMONT under the pSOSystem environment, you can specify a macro in the `sys_conf.h` file to set or disable the extra timer automatically by setting `PM_TIMER` to `YES` or `NO`, respectively.

pRPC+

```
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 */

   int nr_opensubcomp;               /* If set, pRPC is OpEN sub-component */
                                     /* Default pRPC is pNA sub-component */

   int (*nr_gethostname)(char *hostname, int hostnamelen);
                                      /* User callout(): Get host name
                                        function */

   int  (*nr_get_hentbyname)(char *name, void *paddr);
                                      /* User callout(): Name resolver
                                        function. */

   struct nr_pmap_info *nr_pmap_info; /* PMAP task parameters.
                                        Default if null */
   long nr_debug_flag;                /* PMAP task will print debug messages
                                        upon failure if this flag is set */
   long reserved[5];                 /* 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.
nr_opensubcomp	Set to 1 if pRPC+ is configured as subcomponent of OpEN. Should be set to 0 if pRPC+ is subcomponent of pNA+.
nr_gethostname	Contains the address of the user function that returns the local host name. The calling syntax is: nr_gethostname(char *hostname, long hostnamelen); The first parameter `hostname` points to a buffer and second parameter `hostnamelen` points to the length of buffer `hostname`. This entry should be 0 if no handler is present. If this entry is set, pRPC+ uses it for getting the host name. If the entry is not provided and pRPC+ is a subcomponent of pNA, the *gethostname()* function from pNA is used. When pRPC+ is a subcomponent of OpEN, this entry must be set since OpEN does not provide this function. For forward compatibility with future revisions of pNA and pRPC+, this entry must be set for both pNA and OpEN.

nr_get_hentbyname	Contains the address of user function to retrieve the IP address of a host.

The calling syntax is:

```
  nr_get_hentbyname(char *name, void *paddr);
```

The input parameter name points to the host name. The function should return the IP address in hex notation in the return buffer pointed by `paddr`.

If this entry is set, pRPC+ uses it for getting the host name. If the entry is not provided and pRPC+ is a subcomponent of pNA, the `get_hentbyname` function from pNA is used.

When pRPC+ is a subcomponent of OpEN, this entry must be set since OpEN does not provide this function.

For forward compatibility with future revisions of pNA and pRPC+, this entry must be set for both pNA and OpEN.

nr_pmap_info

Points to structure containing portmapper task configuration information. If this field is set to 0, default values will be used.

```
struct nr_pmap_info
    {
    unsigned long pmap_pri;         /* task
priority */
    unsigned long pmap_sstack;   /* supervisor
stack size */
   unsigned long pmap_ustack;  /* user stack size
*/
    unsigned long pmap_flags;     /* t_create
flags */
    unsigned long pmap_mode;    /* t_start mode */
   long reserved[5];       /* Reserved entries of
pmap_CT */
    };
```

`pmap_pri:  contains the priority of portmapper task.`
`pmap_sstack:  contains system stack size for portmapper task.`
`pmap_ustack:  contains user stack size for portmapper task.`
`pmap_flags:  contains create flag value for portmapper task.`
`pmap_mode:  contains start mode for portmapper task.`
`reserved: these fields must be 0.`

nr_debug_flag

Indicates that the `nr_debug_flag` is set to 1. The Portmapper task will print error messages if an error occurs during it's initialization.

reserved

Should all be 0 for upward compatibility.

7 Memory Usage

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 7-2)
- pHILE+ File System Manager (See page 7-7)
- pREPC+ Run-Time C Library (See page 7-10)
- pNA+ TCP/IP Network Manager (See page 7-12)
- pRPC+ Remote Procedure Call Library (See page 7-16)
- pMONT+ (See page 7-17)
- pSE+ (See *OpEN User's Guide*)
- pSKT+ (See *OpEN User's Guide*)
- pTLI+ (See *OpEN User's Guide*)
- pLM+ (See page 7-18)

pSOS+

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
- Task Specific Data Memory

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)
Global Data	3812
System stack	`kc_sysstk`
Task Control Blocks (TCBs)	(`kc_ntask` + 2) x 260
Queue Control Blocks (QCBs)	`kc_nqueue` x 84
Semaphore Control Blocks (SCBs)	`kc_nsema4` x 48
Message Buffers	`kc_nmsgbuf` x 20
Timer Control Blocks (TMCBs)	`kc_ntimer` x 48
Mutex Control Blocks (MCBs)	kc_nmutex x 36

Usage	Size (bytes, decimal)
Condition Variable Control Blocks (CVCB)	kc_ncvar x 24
Task Variable Entries	kc_ntvar x 12
TSD Objects Control Blocks (TSDCBs)	kc_ntsd x 12
Service Call Extension Control Blocks (SVCCB)	(kc_maxscmajor - 15) x (4 + 8 x 16) + 36 + 24
Callout Control Blocks (COCBs)	kc_ncocb x 28
Local Object Table	((`kc_nlocobj` + 3) 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_maxio` x 36) + 36
Device Name Table	kc_maxdntent x (20 + long word sized ceiling of (kc_dnlen + 1)

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 *stack. The memory for all stacks is allocated from Region 0. The sizes of a task’s stack is defined by parameters passed in the* `t_create()` *system call. The following paragraphs describe issues that must be considered when sizing these stacks.*

First, sizing this task requires a determination of the worst case, nested procedure stack usage.

Secondly, both pSOS+ kernel and interrupts are involved in sizing a task’s stack. Any of the following can use a task’s supervisor stack:

- The code of the task code (including its ASR and callouts)

- The pSOS+ kernel, if the task makes pSOS+ system calls (The worst case use within any pSOS+ system call is 256 bytes.)
- Device drivers, if the task makes pSOS+ I/O calls (The worst case use within an I/O call is 264 bytes plus the additional use within the user's drivers.)
- ISRs (the use because of interrupt activity is 184 bytes)

Since interrupt activities result in a switch from the stack of the running task to the system stack, the system 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.

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:

> 80 + ((*length-80)*/(*unit_size* + 6) x 6) bytes, rounded up to the next 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 formula for the amount of memory reserved for a Partition Header is derived as follows:

rbsize = *PTbsize* rounded to next multiple of long word size.

Space till next rbsize-aligned address,

fbuf_size = (*rbsize* - ((*PTaddr* + 52) modulo *rbsize*)

End address of first buffer,

$fbuf\_end = PTaddr + 52 + fbuf\_size$

End address of last buffer,

$lbuf\_end = PTaddr + PTlength - ((PTaddr + PTlength) \% rbsize)$

```
$Nrem = ((lbuf\_end - fbufend) / rbsize) - (fbuf\_size \times 8)$
if ($Nrem > 0$) then
    $remsize = rbsize \times (Nrem - \text{integer ceiling of } (Nrem / (rbsize \times 8 + 1)))$
else
    $remsize = 0$;
```

Hence, the size of the Partition Header is ($52 + fbuf\_size + remsize$) bytes,

where *PTaddr*, *Ptlength*, and *PTbsize* 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, 264 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.

Task Specific Data Memory

At task creation, pSOS+ allocates at least (`kc_ntsd` x 4) bytes to a task, as its TSD pointer array. In addition, for all existing TSD objects that were created with the automatic allocation option, pSOS+ allocates the memory for the TSD areas of size defined during `tsd_create()`. The size is rounded to the next multiple of long word size.

pHILE+

Description

The pHILE+ file system manager needs RAM for the following elements:

- Data Area
- Stack
- TCB Extensions

Data Area

Data area requirements for the pHILE+ file system manager 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	CD-ROM	DOS FAT	pHILE+	NFS Client
Static pHILE+ variables	948+			24	200
Buffer headers	`fc_nbuf` x 48				
Cache buffers	`fc_nbuf` x BUFFSIZE†				
Directory name cache	`fc_ndnlc`‡ x 36				
Mounted volume table	`fc_nmount` x *`formatsel`*††	228	248	400	248
File control blocks (FCB)	(`fc_nfcb` + `fc_nmount`) x *`formatsel`*	52	68	72	160
Task extension area	132 + ((fc_ncfile + 1) x *`formatsel`*[c])	24	28	24	40
Directory name cache	`fc_ncfile` x 36				

†. BUFFSIZE is max(2^`fc_logbsize`, 512 if DOS FAT format included, 2048 if CD-ROM included).

‡.`fc_ndnlc` is changed to 0 if CD-ROM is excluded, or to 2 x `fc_nmount` if zero and CD-ROM is included.

††.*formatsel* = max(CD-ROM, DOS FAT, pHILE+, NFS Client)

NOTE: If fc_dnlc = 0, it is changed to fc_nmount x 2.

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

The pHILE+ file system manager 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 the pHILE+ file system manager in a system, pSOS+ kernel allocates a pHILE+ TCB extension for each task at task creation. Memory for a TCB extension comes from Region 0, and the following table gives its sizes:

Usage	Size in bytes	CD-ROM	DOS FAT	pHILE+	NFS Client
Task extension area	132 + ((fc_ncfile + 1) x *formatsel*[†])	24	28	24	40

†. *formatsel* = max(CD-ROM, DOS FAT, pHILE+, NFS Client)

`fc_ncfile` is the entry in the pHILE+ Configuration Table that specifies the maximum number of open files allowed per task.

Example Memory Usage Computation

An example memory usage computation is below.

TABLE 7-1 Value Calculation

Parameter	Value	Parameter	Value
`fc_logbsize`	9	`fc_ndnlc`	64
`fc_nbuf`	6	`SC_PHILE_CDROM`	0
`fc_nmount`	3	`SC_PHILE_MSDOS`	1
`fc_nfcb`	10	`SC_PHILE_NFS`	0
`fc_nfile`	2	`SC_PHILE_PHILE`	1

TABLE 7-2 Memory Usage Computation

Usage	Computation
Static pHILE+ variables	1,008 = 984 + 24
Buffer headers	288 = 6 x 48
Cache buffers	3,072 = 6 x 2^9
Directory name cache	0
Mounted volume table	1,200 = 3 x max(248, 400)
File control blocks (FCB)	936 = (10 + 3) x max(68, 72)
Total data area	6,504
Task extension area	216 = 132 + ((2 + 1) x max(28, 24))

pREPC+

Description

The pREPC+ library needs RAM for the following elements:

- Data Area
- Stack
- TCB Extensions

Data Area

The pREPC+ library requires a fixed-size data area of 256 bytes. Memory for the pREPC+ 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

The pREPC+ library uses the caller's stack for temporary storage and automatic variables. The pREPC+ library requires a maximum of 1 Kbyte of stack space.

TCB Extensions

- Per Task Data Area—pREPC+ allocates 60 bytes of memory as per-task data storage at the time of task creation. This memory is freed when the task is deleted.

Dynamic Memory Requirements

The pREPC+ library allocates memory dynamically on an as needed basis, during different stages of system operation. These requirements and the conditions under which the allocation occurs are detailed below.

- Memory for maintaining the bookkeeping information associated with memory allocation—This memory is allocated from Region 0 at system startup time and the size of the memory is given by the following formula:

```
if (kc_rn0usize >= 128) then
  mem_allocated = ($\log_2$ (kc_rn0usize) - $\log_2$(32)) x 16 +
  (kc_rn0len/kc_rn0usize) x 4
```

where:

`kc_rn0usize` and `kc_rn0len` are the pSOS+ configuration table parameters for Region 0 unit size and Region 0 length, respectively.

- Per Task Data for maintaining bookkeeping information associated with I/O Streams—The first time any of the standard I/O functions are invoked by a task, pREPC+ allocates memory. The memory allocation size is given by the following formula:

  ((`lc_numfiles` + 3) x 36 + 4)

  where:

  `lc_numfiles` is the pREPC+ configuration table entry for maximum number of simultaneously open I/O Streams that a task can have (excluding `stdin`, `stdout`, and `stderr`). This memory is freed when the task is deleted.

- Memory for Data Buffers for I/O Streams—pREPC+ allocates `lc_bufsiz` bytes of memory as a data buffer, every time an I/O Stream is opened. Note that the `stdin`, `stdout`, and `stderr` streams are opened by pREPC+ automatically when the first I/O operation is performed on these streams. These streams are never opened for tasks that do not do any I/O operation on them.

  The memory allocated for Data Buffers for I/O Streams is not freed when the stream is closed. This expedites the opening of another stream that reuses the stream control block. The memory for Data Buffers is freed when the task is deleted.

pNA+

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	4856
Network Interface Table	`nc_nni` x 120
Routing Table	`nc_nroute` x 100
ARP Table	`nc_narp` x 40
Host Table	`nc_nhentry` x 44
Socket Control Blocks	`nc_nsockets` x 156
Protocol Control Blocks	`nc_nsockets` x 76
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 28
Data Block Table	Number of different buffer sizes x 24
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

The pNA+ network manager uses the caller's stack for temporary storage and automatic variables. The worst case stack usage by the pNA+ network manager is 2 Kbytes plus the worst case stack usage for network interface drivers. The interrupt stack size must be at least 2 Kbytes.

If the pROBE+ debugger is using the pNA+ network manager for communication purposes, the pROBE+ stack size must be increased by 2 Kbytes plus the worst-case stack usage for network interface drives. The stack size is configurable.

TCB Extensions

With the pNA+ network manager in a system, the pSOS+ kernel 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.

The pNA+ network manager 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 the pNA+ network manager 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 */
   short whichp;                 /* Used internally */
   short reserved;               /* Future use */
   };
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 the pNA+ network manager.
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 the pNA+ network manager.
db_debug	Used internally by the pNA+ network manager.
db_frtn	Free function and argument.

Data Buffers

A data buffer is a contiguous block of memory used for storing packets/messages.

pRPC+

RAM requirements

Description

pRPC+ requires:

- 4 Kbytes of supervisor stack (pRPC+ executes in supervisor mode only, uses the calling task's supervisor stack, and requires no user stack space.).
- 76 bytes for pRPC+ extensions to each task control block.
- 672 bytes for PowerPC variables.

pMONT+

pMONT+ requires memory for two reasons:

- To keep track of information about creation and deletion of system objects.

  This memory buffer is allocated from region 0 and the size is 96 * `KC_NLOCOBJ` bytes. `KC_NLOCOBJ` is the maximum number of pSOS+ kernel objects which is set in the `sys_conf.h` file. In case of a multi-processor system with pSOS+m, the equivalent number is 96 * (`KC_NLCOBJ` + `MC_NGLBOBJ`). `MC_NGBOBJ` is the maximum number of global objects.

- To Log the events occurring when an ESp experiment is on.

  This memory buffer is know as the trace buffer can be specified in the `sys_conf.h` file. If you want to allocate the memory for the trace buffer, the variable `PM_TRACE_BUFF` should be set to the starting address of such memory. The `PM_TRACE_SIZE` should be set tot he size of this memory.

  If `PM_TRACE_BUFF` si zero and `PM_TRACE_SIZE` is non-zero, then pMONT+ allocates this memory from FreeMemPtr during system startup.

  `PM_TRACE_SIZE` should be at least 1000 bytes for an ESp experiment to be configured.

pLM+

Description

The pLM+ shared library manager needs RAM for the following elements:

- Data Area
- Stack
- TCB Extensions

Data Area

Data area requirements for the pLM+ shared library manager depend on user supplied entries in the pLM+ Configuration Table. The size of the data area is the sum of the values generated by incorporating the relevant table entries (each of which begins with `lm`) in the following formulas:

Usage	Size in bytes
`Static pLM+ variables`	112
`Table of registered libraries`	40 x `lm_maxreg`
`Bit maps`	(`lm_maxreg` + 1) x Bit map size
`Bit map size`	4 x (1 + `floor`(`lm_maxreg`/32))

For example, if `lm_maxreg` is 8 the bit map size is 4 x (1 + `floor`(8/32)) which is 4. The data area requirement is 112 + (40 x 8) + ((8 + 1) x 4) which is 468.

Memory for the data area of pLM+ can be allocated from Region 0, or it can be allocated from a fixed location. The location depends upon the pLM+ Configuration Table entry `lm_data`.

Stack Requirements

The pLM+ shared library manager executes in supervisor mode and uses the supervisor stack of the caller for temporary storage and automatic variables. The worst case supervisor stack usage of calls to `sl_acquire()`, `sl_bindindex()`, and `sl_register()` that register new shared libraries depends upon the depth of the dependency tree of shared libraries not already registered. The worst case caller's supervisor stack usage of all other pLM+ system calls, and of these system calls if no new shared libraries are registered, is fewer than 4096 bytes. In most cases this should be enough for system calls that register new shared libraries. If not, and due to deep shared library dependencies, use more. Therefore, a task that uses pLM+ should be created with at least that much stack space.

Task Extension Areas

With the pLM+ shared library manager in a system, the pSOS+ kernel allocates a pLM+ TCB extension for each task at task creation. Memory for a TCB extension comes from Region 0, and the following formula gives its size:

4 + (2 x `lm_maxreg`)

For example, if `lm_maxreg` is 8 the TCB extension size is 4 + (2 x 8) which is 20.

8 pNET: Ethernet Debugging Without Using pNA

8.1 Overview of pNET

pNET is derived from pNA, the networking component of the pSOS real-time operating system. pNET is designed and is developed to provide the target debug solution based on the UDP/IP protocol over various transmission medium, for example, Ethernet and serial line. pNET is highly portable, efficient, and exists in the minimum amount of main memory throughout its operational lifetime. pNET is designed mainly to service network service requests made by pROBE, for its communication with the host debugger over various types of network; and by TFTP, for fast downloading of the application code to the remote target. Communication through pNET assumes UDP as the transport protocol. As such, it is optimized for pROBE and TFTP operation. pNET, unlike pNA, is a non-reentrant library. There does not exist a daemon task for packet and IP/ARP timers related processing. pNET is independent of pNA. TCP transport protocol is not supported in this version of the pNET. Therefore, it is not possible for pMONT to operate over pNET.

8.2 Configuration of PNET

pNET is intended to fulfill the responsibilities of pNA to pROBE and TFTP in the absence of pNA. pNET configuration parameters map directly into pNA configuration table, however, not every parameter from pNA configuration table is applicable to pNET. Communication through pNET allows the host debugger to reside on networks different from that of the target. This is achieved through the default gateway mechanism. Routing IP traffic through pNET seems irrelevant when pNET is intended for host debugger to target communication. Therefore, pNET does not maintain any routing information internally. pNET maintains the default information when default gateway is enable, and regardless of the value of the `NCNROUTE` parameter configured. Consequently, `NC_NNODEID` which specifies the router ID is ignored as well. Parameters relating to IP multicasting are ignored during pNET configuration process. In pNA the ARP table provides the dynamic mappings of IP addresses to corresponding hardware addresses. Again, as pNET emphasizes communication between the host debugger and the remote target, only one ARP entry is cached by pNET. Normally, this ARP cache contains the mapping of the host debugger IP address to its corresponding hardware address. The ARP cache is invalidated when the IP address of the outgoing packet does not match that of the ARP cache entry. The ARP cache entry is permanent otherwise. The value of the NCNARP parameter is ignored.

pNET has the same memory configuration procedure as pNA.

pNET and pNA cannot both be active at the same time. If both are selected, then pNA takes precedence over pNET. To select pNET, set the SC_PNET parameter in application sys_conf.h to YES.

Index