the log is "-", then the LLOGTTY option is enabled and no file is actually opened. When determining the actual name of the file to use, a "%d" in the name will be replaced by the process-id of the program opening the log. On failure, the manifest constant NOTOK is returned. Otherwise, the log is opened (and left open, regardless of the presence of the LLOGCLS option), and the manifest constant OK is returned.

To close a log, the routine ll_close is used:

```
int     ll_close (lp)
LLog   *lp;
```

The parameter to this routine is:

lp: a pointer to a LLog structure.

This routine returns the manifest constant OK on success (even if the log was already closed), or NOTOK otherwise.

7.1.1 Timestamps

One of the characteristics of a log is that it contains an informational timestamp for each entry. This timestamp contains the date and time of the log and also two "header" strings, a static header and a dynamic header. Normally, these strings are constructed from the name of the program or subsystem using the log. The routine ll_hdinit is used to initialize the static header:

```
void   ll_hdinit (lp, prefix)
LLog   *lp;
char   *prefix;
```

The parameters to this routine are:

lp: a pointer to a LLog structure; and,

prefix: the name of the program or subsystem using the log.

This routine will form a header consisting of the program name, the process-id, and the user-name.

The routine ll_dbinit is similar, but also enables debugging features:

```
void   ll_dbinit (lp, prefix)
LLog   *lp;
char   *prefix;
```

The parameters to this routine are:

`lp:` a pointer to a `LLog` structure; and,

`prefix:` the name of the program or subsystem using the log.

This routine will form a header identical to the one formed by `ll_hdinit`. It will then set the name of the file associated with the log to be relative to the current working directory. Finally, it turns on all event logging and logging to the user's terminal.

7.1.2 Making Log Entries

At the lowest level, the `ll_log` routine is used to append an entry to a log:

```
int     ll_log (lp, event, what, fmt, args ...)
LLog   *lp;
int     event;
char   *what,
       *fmt;
```

The parameters to this routine are:

`lp:` a pointer to a `LLog` structure;

`event:` the event type being logged (e.g., `LLOG_NOTICE`);

`what:` some text associated with a system call error (use the manifest constant `NULLCP` if the entry is not associated with an error in a system call); and,

`fmt/args:` an argument list to *printf*(3s).

The entry is only made if the log is enabled (in the `ll_events` field of the `LLog` structure) for the event listed as a parameter to `ll_log`. If there was a problem in writing to the log, `ll_log` returns the manifest constant `NOTOK`. Otherwise, `OK` is returned.

The `ll_log` routine is actually a simple wrapper around the `_ll_log` routine:

```
int     _ll_log (lp, event, ap)
LLog   *lp;
int     event;
va_list ap;
```

The parameters to this routine are:

`lp:` a pointer to a `LLog` structure;

`event:` the event type being logged; and,

`ap:` an argument pointer to a variable-length argument list as described in *varargs*(3).

It may be necessary to have multi-line log entries. In this case, the first line of the entry should be made with `ll_log`. The remaining lines should be made with the `ll_printf` routine:

```
int     ll_printf (lp, fmt, args ...)
LLog   *lp;
char   *fmt;
```

The parameters to this routine are:

`lp:` a pointer to a `LLog` structure; and,

`fmt/args ...:` an argument list to *printf*(3s).

As with `ll_log`, this routine returns either `OK` on success or `NOTOK` on error. Unlink `ll_log` however, `ll_printf` will ignore the setting of the `LLOGCLS` option. As such, when the last line of a multi-line entry has been made, the routine `ll_sync` should always be called to synchronize the log:

```
int     ll_sync (lp)
LLog   *lp;
```

The parameter to this routine is:

`lp:` a pointer to a `LLog` structure.

7.1.3 More About Making Log Entries

Although the `ll_log` routine has a basic functionality, programmers often prefer a slightly simpler interface. A few macros have been defined for this purpose.

The `SLOG` macro is the most commonly used:

```
SLOG (lp, event, what, args)
```

The parameters to this macro are:

`lp:` a pointer to a `LLog` structure;

`event:` the event type being logged;

`what:` some text associated with a system call error (use the manifest constant `NULLCP` if the entry is not associated with an error in a system call); and,

`args:` a parenthesized argument list for *printf*(3s).

The `SLOG` macro compares the event enabled for a log to the event being logged to see if `ll_log` should be called.

Since, the need for a `what` parameter is not common in many applications, the `LLOG` macro has been supplied. It is essentially the `SLOG` macro but with a value of `NULLCP` supplied for the `what` parameter of `ll_log`:

```
LLOG (lp, what, args)
```

Further, even though logging is contingent on an event type being enabled, a programmer may still wish that calls to logging package still be conditionally compiled. The `DLOG` macro has been supplied for this purpose. If the pre-processor symbol `DEBUG` is defined, then `DLOG` is equivalent to `LLOG` otherwise it compiles no code whatsoever:

```
DLOG (lp, what, args)
```

Finally, it may be useful to log PDUs (protocol data units), again under conditional compilation. The `PLOG` macro takes the address of a pretty-printer function generated by *pepy*(1) (see Section 6.5 on page 75 in *Volume Four*) along with a presentation element (as described in Chapter 5 in *Volume One*) and a brief textual title, and directs the pretty-printer to output to the log:

```
PLOG (lp, fnx, pe, text, rw)
```

The `rw` parameter is an integer saying wheter the PDU was read from the network (non-zero) or written to the network (zero-valued). As with the `DLOG` macro, if the `DDEBUG` symbol is not defined, then no code is generated.

7.1.4 Miscellaneous Routines

In order to support some of the more esoteric log capabilities, there are a few utility routines.

The routine `ll_preset` evaluates a *printf*(3s) argument list and returns a pointer to static buffer containing the result:

```
char   *ll_preset (fmt, args ...)
char   *fmt;
```

The routine `ll_check` determines if a log has exceeded its size, and if so, if a correction can be made:

```
int     ll_check (lp)
LLog   *lp;
```

This routine returns `OK` if the log is within its bounds.

7.2 Use of Logging in Programs

From the perspective of applications programmers, there are three kinds of styles for using the logging package, depending on the kind of program being written.

In all three cases, `LLog` structures are usually declared statically in the main module of a program and a pointer to the structure is made available for general use:

```
static LLog _pgm_log = {
    "myname.log", NULLCP, NULLCP,
    LLOG_FATAL | LLOG_EXCEPTIONS | LLOG_NOTICE,
    LLOG_FATAL,
    -1,
    LLOGCLS | LLOGCRT | LLOGZER,
```

```
    NOTOK
};

LLog *pgm_log = &_pgm_log;
```

Where the `myname` in "`myname.log`" is replaced with the name of the program.

Note that in all cases, in order to ensure consistent logging it is **critical** that the call to `isodetailor` be the first call made to *any* of the ISODE routines.

For static responders, two routines are called in the initialization code:

```
isodetailor (argv[0], 0);
ll_hdinit (pgm_log, argv[0]);
```

Later on, after argument parsing, if a debug option is enabled, then

```
ll_dbinit (pgm_log, argv[0]);
```

is called.

For dynamic responders, a similiar code sequence is used:

```
isodetailor (argv[0], 0);
if (debug = isatty (fileno (stderr)))
    ll_hdinit (pgm_log, argv[0]);
else
    ll_dbinit (pgm_log, argv[0]);
```

For user-interfaces, the code is simply:

```
isodetailor (argv[0], 1);
ll_hdinit (pgm_log, argv[0]);
```

which will ask the tailoing system to read both the standard tailoring file and a user-specific tailoring file and then to initialize the program log.

Part IV

Configuration

Chapter 8

The Transport Switch

As of this writing, the concept of ubiquitous OSI has not yet been realized. In particular, there continues to be disagreement on the transport/network protocols to be used to provide end-to-end service. As a result, interworking between sites in many cases is the *exception*, not the rule.

In order to facility communication, the ISODE has a powerful abstraction, the *Transport Switch*. The goal of the transport switch is to hide the complex underpinnings of the "real world" of OSI from the user of the transport service. In order to make effective use of this abstraction, it may be necessary to tailor the ISODE (using the *isotailor*(5) file described in Chapter 6).

To explain how to use the transport switch effectively, it is necessary to introduce some terminology.

8.1 Transport Stacks

A *TS-stack* refers to a combination of transport protocol and network service that is used to provide end-to-end transport service. The ISODE, depending on how you configure it at compile-time, supports any combination of these

stacks:

Mneumoic	TS-Stack
`tcp`	RFC1006 over TCP/IP
`x25`	TP0 over X.25
`x25-84`	TP0 over X.25(1984)
`bridge`	TP0 over the TP0–bridge
`tp4`	TP4 over CLNP

Internally, the ISODE uses "typing" on all network addresses to one of these choices. From a practical perspective, usually only the `tcp`, `x25-84` and `x25` are of interest.

The run-time tailoring variable `ts_stacks` keeps track of the TS-stacks available on your host. This defaults to the TS-stacks that were configured at compile-time. However, if you are sharing executables between machines with different network attachments, you will want to change this. Generally, you compile-time configure the ISODE for all stacks available at your site. Then, for each host you install the executables on, you set the variable `ts_stacks` accordingly. So, suppose you have a host with both TCP and X.25 services, but that all the other hosts at your site have only TCP. In this case (assuming binary compatibility between the hosts at your site), you generate the ISODE on the host with both TCP and X.25 services configured. When you move the executables to the TCP only hosts, you add the line

```
ts_stacks: tcp
```

to the *isotailor* file on those hosts.

8.2 OSI Communities

An OSI *community* is a collection of hosts which share a common TS-stack along with basic connectivity. Simply put, a community consists of end-systems that all interwork with each other. In a perfect world, there would be but a single community. `realns`. But it isn't a perfect world, my favorite Rock group, *Bangles*, broke up in September of '89, and I am very upset about this. But that is another story.