How to Design a Networking Trace/Log Subformatter and a Sample Subformatter
Subsystems are typically an individual program or set of programs that act in concert. Each subsystem requires an associated subformatter, however several subsystems may use the same subformatter. Subformatter design depends on how logging and tracing are used in the subsystem.

Subsystems also have the capability to provide filtering or formatting options.
Designing a Subformatter

This section deals with the design of the actual function that is called in response to the formatter reading in a record containing data for a specific subsystem. If the data passes the global filters and appears to be a good message, the subformatter for the subsystem is called. The subformatter information is held in a table containing the subsystem ID, mnemonic, subformatter function, options function, message catalog, group name, and the subsystem options data structure.

The entry point for the subsystem options and formatter functions must follow a standard interface, defined in the `subsys_N_get_option` and `subsys_N_format` manpages. This interface provides the subformatter with a pointer to the buffer containing the complete message, including header and body. Additionally, parameters that are intended to be passed through to the formatter utility functions are included. These make up the bulk of the call. Most subformatter developers need only be interested in the few parameters.

The subformatter called for the subsystem must be able to handle both trace and log data. These can be separated into separate functions once the subformatter has been invoked, but there is no provision for the formatter to call more than one function for a given specific subsystem ID.

Formatting requirements for tracing are often different from logging. The developer should take this into consideration in designing the subformatters.

A subformatter developer should view the action of tracing or logging as a communication from the subsystem to the user, a user who sees only a message from the subsystem and not the medium that carried the message. The subformatter developer should consider the design of the subformatter in relation to the types of information that come from the subsystem. For logging, providing a few pieces of information, such as logging event and a couple of data items may be adequate. The subformatter can assemble the formatted output from a message in its message catalog based on the event ID, and the additional data can be inserted into the message. This method is employed by the ARPA logging subsystems.
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

Tracing information can be more of a problem, partly because it usually contains much more data, especially in the case of link-level packet tracing (PDU in or out tracing). The subformatter may have to know how the packet was constructed, which layer sits on top, and so on.

For tracing or logging, the subsystem should pass only as little data as possible to output complete, useful information. Flags are available for subformatters to control the format of the output data. For example, some flags request that output be limited to one line, and other flags request that every possible piece of information be decoded. Subformatters are expected to tailor their output according to these flags.

Alternative Subformatter Implementations

As shown in Figure 7-8 on page 282, Chapter 7, device driver developers must also provide for HP-UX network trace/log data formatting in the lib_N_fmt.sl file. Two developer-provided subsystem formatter functions required in this file must be named subsys_N_get_option and subsys_N_format. At the option of the developer, these routines can accomplish their formatting by using the basic calls of netfmt (see the following “HP-UX Subsystem Formatter Functions” section) or by using the well-developed subformatter components in libnsfmt.sl (see the following “HP-UX Subsystem Formatter Functions” section).

Subformatter Responsibilities

The subformatter has few responsibilities, other than transforming the data, as follows:

1. Perform subsystem filtering (if this feature is provided)
2. Print the header.
3. If console logging is on, format a terse message; or otherwise, format a message in accordance with the format flags.
4. Write the formatted message.
These responsibilities can be performed with the help of the utility functions provided for subformatters. In fact, for consistency and efficiency, all subformatters should use these functions:

- `tl_banner_check()`
- `tl_trace_kind()`
- `tl_log_class()`
- `tl_header_format1()`

Use of the functions above ensures that all headers look alike, thus helping users find useful landmarks to guide them through the output.

- `tl_format_write()`
- `tl_format_fprintf()`

Subformatters are free to call the output functions above for each line or for several lines of output, as appropriate.

- `tl_raw_format()`

Use of the above function ensures that all hexadecimal dumps look alike and behave consistently.

By using the common functions, the underlying implementation of I/O may be changed more easily, thus allowing easier porting, further performance enhancements, alternative output schemes, etc.

Console logging is determined by a flag passed in the subformatter call. When this flag is enabled, console logging is in effect, which means that the subformatter should use a terse, one-line message instead of a more verbose explanation. The `tl_format_write()` and `tl_format_fprintf()` functions print messages on the console.

*Terse* (one-line) trace formatting is determined by a flag passed in the subformatter call. When this flag is enabled, the subformatter should only print one line per trace message. This mode of formatting is used to get a summary of the trace file contents. Additional flags also control the behavior of terse formatting.

*Nice* (detailed) trace formatting is determined by a flag passed in the subformatter call. When this flag is enabled, the subformatter should attempt to identify and label every piece of data in the trace.

If neither *terse* nor *nice* is enabled, raw formatting should be used.
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

NOTE Each subformatter should follow Hewlett-Packard standards in formatting the data output. These standards are implemented in the HP-UX subformatters of trace and log data for Hewlett-Packard networking interfaces. Study the output of these HP-UX subformatters as templates for the data output of any new subformatter. Simple examples are provided in “HP-UX Formatting Library Routines” on page 594, for the format_link_nice, format_link_terse, and format_link_raw routines.

HP-UX Subsystem Formatter Functions

The following routines are supplied by the subsystem for the formatter to call when subsystem specific actions need to take place. Subsystem specific actions include parsing filter files and formatting the subsystem's trace or log data. These functions must be placed in a shared library.

Shared libraries are usually created by compiling all modules with the “+z” option to cc and linking them together using the options to ld:

```
-b +e subsys formatter function +e subsys get options function
```

which is then configured in the /etc/nettlgen.conf file using the nettlconf command. The +e option to ld must be used to prevent symbol collisions among the different subformatter libraries. For further explanation and details, refer to the ld(1) manpage. The formatter and user interface commands use the configuration information each time they are invoked.

Development and support of subsystem subformatters are responsibilities of the device driver developer for that subsystem.

subsys_N_format() The subsystem developer must provide a subsys_N_format() routine to format a single trace or log message from the N subsystem.

This routine, along with the shared library that contains it, is configured with the nettlconf command (see nettlconf(1M)) into the nettlgen.conf configuration file. subsys_N_format() is the default name; the value of N is the subsystem ID number assigned by HP (see “Assign Subsystem ID” on page 285, Chapter 7). The actual function name can be redefined with the nettlconf command.
At run time, the `netfmt` command loads the library and calls the routine whenever data from the subsystem is encountered. The `subsys_N_format()` routine may discard the message based on filter information supplied by the user in the options file, as determined by the `subsys_N_get_options()` routine associated with the subsystem. It returns 0 if no errors are encountered, otherwise it returns a −1.

The routine is defined as:

```c
int subsys_N_format(ss_N_fmt_flag_type Flags,
    char *BinaryMsgPtr,
    char *OptionsPtr,
    int32_t MsgCatFD,
    int32_t ErrorFD,
    int32_t OutputFileCount,
    fp_result Outputfiles[],
    char *TimeBuffer,
    int32_t TimeBufferLength,
    int32_t PrintOp,
    int32_t UserCount,
    user_acct_result Users[],
    err_num *Status)
```

### Flags

The type of `Flags` is defined as:

```c
typedef struct {
    unsigned verbosity_bit: 1;
    unsigned console_logging: 1;
    unsigned highlight_bit: 1;
    unsigned nice_mode_bit: 1;
    unsigned terse_mode_bit: 1;
    unsigned terse_link_mode_bit: 1;
    unsigned terse_time_mode_bit: 1;
    unsigned map_to_names_bit: 1;
    unsigned reserved: 24;
} ss_N_fmt_flag_type;
```

- **verbosity_bit**  When this bit is set, a high level of verbosity has been selected (high verbosity is the default).
- **console_logging**  This bit is set if console logging is enabled, in which case the subformatter should only call the `tl_header_format1()` routine and provide very minimal additional information (to be kept to one line)
Designing a Subformatter

**highlight_bit**  This bit is set if highlighted output is enabled (highlighted output enabled is the default).

**nice_mode_bit**  This bit is set when nice formatting has been enabled (nice output not enabled is the default). Nice formatting is the most descriptive mode of formatting. All possible information should be displayed in this mode of output. *Nice* mode is not usually used for log messages.

**terse_mode_bit**  This bit is set when terse formatting has been enabled (terse output not enabled is the default). Terse formatting should be limited to one line of output per trace record. *Terse* mode is not usually used for log messages.

**terse_link_mode_bit**  If *terse* mode is enabled then *terse_link_mode_bit* is a flag that should cause the link name to be included in the output.

**terse_time_mode_bit**  If *terse* mode is enabled then *terse_time_mode_bit* is a flag that should cause the timestamp to be included in the output.

**map_to_names_bit**  This bit is set when numbers should be resolved into names whenever possible (mapping numbers to names is enabled by default). For example, an IP address should be displayed as a hostname if the *map_to_names_bit* flag is set.

**BinaryMsgPtr**  Pointer to a buffer that contains the binary trace/log message to be formatted. The buffer contains the trace/log header, struct `tl_msg_hdr` (as follows) from the `/usr/include/ntl.h` file, followed by the trace/log data (from `ktrc_write` or `klogg_write`):

```c
typedef struct {
    unsigned short  hdr_len;
    short           subsystemid;
    int             device_id;
    tl_msg_flag_type  flags;
    set_of_32      kind;
    set_of_32      class;
} tl_msg_hdr;
```
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

### Designing a Subformatter

```c
set_of_32 version;
unsigned int dropped_events;
unsigned int dropped_data;
unsigned int data_len;
unsigned int orig_data_len;
struct timeval time;
int invoke_id;
int path_id;
unsigned short log_instance;
uid_t uid;
unsigned int connection_id;
} tl_msg_hdr_type;
```

**NOTE**

For tracing, the data may be truncated by the nettl command facilities. Check the `tl_msg_hdr > data_len` field to find out how much data was captured.

---

**OptionsPtr**

Pointer to a data structure defined by the subsystem for communication between the `subsys_N_get_options()` routine and the `subsys_N_format()` routine. If no options are used, then this pointer is null. The actual type of the structure pointed to by `OptionsPtr` is entirely up to the subsystem developer.

**MsgCatFD**

File descriptor of the subsystem message catalog configured in `nettlgen.conf`. Subsystems should not open their own message catalog files.

**ErrorFD**

File descriptor that refers to be file that will receive any error messages.

**OutputFileCount**

Number of output files to receive the formatted trace/log messages. For HP-UX, this parameter must have a value of 1.

**OutputFiles**

Array of structures, each of which contains a file pointer and a result.

```c
typedef struct {
    int fd;
    int result;
} fp_result;
```
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

This file receives the formatted trace/log messages. Only one output file is used for HP-UX, `OutputFiles[0].fd` (`.OutputFiles[0].result` is ignored.) This output file will have been opened by the formatter driver. Fatal errors on HP-UX should be reported through the return code and status parameters. Fatal and nonfatal error messages should be written to the file referenced by `ErrorFD`.

`TimeBuffer`  
String containing the formatted timestamp from the trace/log header.

`TimeBufferLength`  
Length of the `TimeBuffer` string, not counting the null terminator.

`PrintOp`  
For HP-UX, this parameter must have a value of 0.

`UserCount`  
For HP-UX, this parameter must have a value of 0.

`Users`  
For HP-UX, this parameter must have a value of NULL.

`Status`  
Contains the error value if the routine returned a −1.

`susys_N_get_options()`  
This routine is supplied by the subsystem developer to process options for the N subsystem.

The `nfmt` command calls this routine whenever a filter configuration file is encountered that contains lines beginning with the subsystem name. It is `susys_N_get_options()` routine's responsibility to read the subsystem specific options information from the filter command file and store any necessary information. It returns a −1 in the event of a fatal error.

This routine is defined as:

```c
int subsys_N_get_options(get_opt_parms_type *Get_Opt_Parms_Ptr)
```

`get_opt_parms_type`

**Defined in** `/usr/include/fmt.h` as:

```c
typedef struct {
  int *status_ptr;
  FILE *subsys_strm;
  FILE *error_strm;
  FILE *log_strm;
  int ss_id;
  char *ss_name;
}
```
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

```c
nl_catd  ss_msg_cat;
get_opt_flag_type  ss_n_get_opt_flag;
char   **ss_options_ptr;
int    ss_output_fd;
char   *options_file_name;
int    *options_filename_printed;
}
```

**status_ptr** Contains the error code of the routine if the returned value is \(-1\).

**subsys_strm** FILE pointer to the file that refers to the temporary file containing the options specifically for the N subsystem. This file is created by the caller prior to invoking `subsys_N_options()` routine, and each line has been converted to lower case. All comments, blank lines and lines for other subsystems are already removed. In addition, the keyword identifying this subsystem has been stripped off each line, so only the options for this particular subsystem are in the file. Due to a special encoding of line number and other data, the `tl_get_line()` routine must be used to get option lines from this stream file.

**error_strm** FILE pointer to the file that will receive error messages.

**log_strm** FILE pointer to the file that will receive a summary of all options and files in effect for the subsystem, generated by `subsys_N_get_options()` routine. The `nett1` command reports the contents of this file after all the subsystems have finished reading their respective filter command files.

**ss_id** Subsystem ID number for the subsystem as found in the configuration file.

**ss_name** Subsystem name for the subsystem as found in the configuration file.

**ss_msg_cat** File descriptor pointing to the message catalog for the subsystem as found in the configuration file.
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

ss_n_get_opt_flag
Type of flag is defined as:

```c
typedef struct {
    u_int trace_log_bit: 1;
    u_int parse_only_bit: 1;
    u_int reserved: 30;
} get_opt_flag_type;
```

- **trace_log_bit** This flag is not needed and should not be used by `subsys_N_get_options`.
- **parse_only_bit** This flag is set when the `subsys_N_get_options()` routine does not need to process the information in the file, only parse the input and check for syntax and semantic errors.

- **ss_options_ptr** Pointer to a pointer to a data structure containing the specific information processed by the `subsys_N_get_options()` routine and passed on to the `subsys_N_format()` routine to handle special formatting. This structure should be allocated and initialized by `subsys_N_get_options()` routine.

- **ss_output_fd** File descriptor referring to the file receiving the formatter output

- **options_file_name** To be filled in.

- **options_filename_printed** To be filled in.

**HP-UX Formatting Library Routines**

The 802.3/Ethernet LAN product provides the ability to format information from upper layer protocols such as IP, TCP, UDP, ARP, DUX, and NFS, from traces taken at the link layer. This capability makes it much easier to analyze networking dialogs than examining raw hex data and manually determining what the protocols were sending.

In addition, the formatter also provides the ability to filter the trace output so that only dialogs taking place with a particular TCP port would be displayed. The filters include Ethernet type, 802.2 SAPs, IP addresses, UDP ports, and RPC information.
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

This existing base of capabilities makes desirable the leveraging of this code to support other link products as they are released. This section describes a set of routines available in the base netfmt product that new link products can and should take advantage of.

Link subformatters may take advantage of the ARPA decoding routines to format link level packets, as follows:

1. The link subformatter calls the appropriate set_up_*() function to prepare the decoder for filtering.
2. The link subformatter then calls filter_packet() to see if the value in the packet will pass the user-specified filters.
3. If the filters pass, then the link subformatter may call the format_link_*() functions to produce formatted output.

NOTE

These decoding routines are the only supported case where routines in one shared library may call those in another. Subsystems should not depend on the load order of subsystem shared libraries.

By using these routines, a link product trace formatter needs to format only the information in its link header, not including the 802.2 information. (Other routines take care of the rest.) Note also that the trace formatter does not directly perform I/O, which is performed through the three provided formatting routines. Using these provided routines allows future changes to be made to the look of the formatted output without modifying the link format code. Using these routines also promotes consistency among links.

The following basic algorithm, for PDU_IN and PDU_OUT trace kinds only, is consistent with these objectives. (Formatting other kinds of trace messages must be done by using the routines described in the “Example: Using HP-UX Formatting Library Routines.” section.

1. Extract local link information from the packet (do not print it).
2. Call set_up_8022() to extract information for 802.2 and upper layer protocols. This routine also handles SNAP.
3. Call filter_packet() to determine if the packet meets any filter criteria. Return without printing if it fails.
4. Based on the setting of the global variables nice_fmt and terse_fmt, call one of the following routines:

- format_link_nice()
- format_link_terse()
- format_link_raw()

When calling these routines, include string buffers containing any link-specific information extracted in step 1.

Figure 7-9 on page 301, Chapter 7, identifies these subroutines and shows their relationship.

**set_up_8022() Routine Description**  This routine sets up global information used by both the filter function and the three formatting functions.

This routine walks through the buffer and copies protocol header information to appropriate global variables used by the filter and formatter. Call this routine for each PDU_IN or PDU_OUT trace event.

```c
set_up_8022(buf_ptr, len, dst_addr, src_addr)
```

- `buf_ptr`  Pointer to beginning of the 802.2 information. Should not include MAC info.
- `len`  Length of the buffer (excluding MAC header).
- `dst_addr`  Pointer to the 6-byte destination MAC address (extracted by local methods from the MAC header).
- `src_addr`  Pointer to the 6-byte source MAC address (extracted by local methods from the MAC header).

**set_up_link() Routine Description**  This routine sets up global information only for the link layer and does not attempt to extract any upper layer information from the traced packet.
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

NOTE

Use this routine only if the packet being formatted cannot be handled by set_up_8022.

```c
set_up_link(buf_ptr, len, dst_addr, src_addr)
    u_char *buf_ptr;
    int len;
    u_char *dst_addr;
    u_char *src_addr;
```

buf_ptr Pointer to the beginning of the Data Link information. Should not include MAC info. The routine does not currently use this parameter, for future extensions.

len Length of the buffer (excluding MAC header).

dst_addr Pointer to the 6-byte destination MAC address (extracted by local methods from the MAC header).

src_addr Pointer to the 6-byte source MAC address (extracted by local methods from the MAC header).

set_up_ip() Routine Description This routine walks through the buffer and copies protocol header information to the appropriate global variables (that it sets up) for use by the filter function and the three formatting functions.

NOTE

Link products should not use this routine. Call this routine only when no link information is available for output formatting (for example, NS_LOOPBACK).

```c
set_up_ip(buf_ptr, len)
    u_char *buf_ptr;
    int len;
```

buf_ptr Pointer to beginning of the 802.2 information, which should not include MAC information.

len Length of the buffer, excluding MAC header.
Designing a Subformatter

**set_up_ether() Routine Description**  This routine sets up global information used by both the filter function and the three formatting functions.

This routine should be called for each PDU_IN and PDU_OUT trace event that contains Ethernet packets.

- **buffer**  Pointer to the beginning of the Ethernet data. It should not include the destination address, source address, or Ethernet type information. This routine will then walk through the buffer and copy protocol header information to appropriate global variables used by the filter and formatter.

- **len**  Length of the buffer, excluding destination, source, and Ethernet type.

- **dst_addr**  Pointer to the 6-byte destination MAC address, extracted by local methods from the MAC header.

- **src_addr**  Pointer to the 6-byte source MAC address, extracted by local methods from the MAC header.

- **ether_type**  Ethernet-type field from the MAC header.

**filter_packet() Routine Description**  filter_packet() examines the globals set up by one of the preceding set_up_xxx() routines and returns 0 if the packet should not be displayed, and the subformatter should return without producing any output.

If the packet meets the filter criteria, a non-zero value is returned.

**format_link_nice() Routine Description**  This routine formats a packet using nice formatting to display upper layer information.

```
format_link_nice(tl_msg_hdr_type *hdr, 
    u_char *buffer, 
    int len, 
    char *linktype, 
    char *linel, 
    char *addlinfo, 
    char *upperinfo) 
```

- **hdr**  Pointer to the standard nettl message header.
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

`buffer`  
Pointer to data beginning at the 802.2 level. The upper layer routines typically will not format data straight from this buffer, but the uppermost layers may display data at an appropriate offset into the buffer.

`len`  
Length of the buffer (including 802.2, excluding any lower layer data).

`linktype`  
String describing the type of link this information is carried over (for example, FDDI, 802.5, ETHER; 802.3 in the following example).

`line1`  
Short string (less than 23 bytes) giving more information to be displayed on the same line as the source address, for example, “TYPE: 0x800” for Ethernet packets (NOT SNAP), or “LENGTH: 26” for 802.3 packets (as in the following example); may be left blank by passing “”.

`addlinfo`  
Additional lines of information pertaining to data in the MAC header. (Blank for 802.3 and Ethernet, but could include formatted flags or other information in the MAC header for other link types). Should be terminated with a newline (\n).

`upperinfo`  
Other lines of information pertaining to data beyond the MAC header. Will be displayed only if the packet does not have 802.2 or Ethernet information present, that is, as in conjunction with `set_up_link()`). Ordinarily should be left blank. If present, you may wish to include a separator (for example, -).

`ReturnValue`  
If any part of the formatting encounters a problem (like packet truncated or an unexpected value in a protocol header), a value of 0 is returned. If the formatting is successful a nonzero value is returned.
Designing a Subformatter

The following shows an example output:

[The linktype parameter goes here]-vvvvv

====================== 802.3 ============================

Source : 00-00-0c-00-06-31 [I] [Cisco ] LENGTH: 26

Dest : 09-00-09-00-00-01 [M] [HP Probe ] TRACED LENGTH: 60

< The addlinfo parameter info goes here

Date : Mon Dec 02 09:22:04:33390 PST 1991

< The upperinfo parameter info goes here

====================== 802.2 ============================

DSAP : 0xfc SSAP : 0xfc CONTROL : 0x03[U-FORM AT]

DXSAP: 0x503 SXSAP: 0x503

===================== PROBE VNA REQ (inbound [ICS]) ======

version: 0 length: 16 seq: 0x6dc1
domain: 1 version: 0 rep len: 8 domrep len: 6
Source: 00-00-0c-00-06-31 Requesting: 15.13.106.63

format_link_terse() Routine Description This routine formats a packet using terse formatting to display upper layer information in a single line.

format_link_terse(tl_msg_hdr_type *hdr, u_char *buffer, int len, char *linktype, char *addlinfo)

hdr Pointer to the standard nettl message header.

buffer Pointer to data beginning at the 802.2 level. The upper layer routines typically will not format data straight from this buffer.

len Length of the buffer (including 802.2, excluding any lower layer data).

linktype String describing the type of link this information is carried over, plus any other MAC layer information appropriate for terse mode. For 802.3 the linktype is simply “8”; for Ethernet (not SNAP) it is “E”.

addlinfo String giving other MAC or upper layer information to be displayed (blank for 802.3 and Ethernet).
The following shows and explains an example output:

8m probe vna request for: 15.13.106.63 from: 00-00-0c-00-06-31
seq: 6dcl

|   |   | Any addlinfo string would appear beginning here.
|+ The linktype is placed here (the second character is placed by
  the function and describes what type of MAC address is used
  (m)ulticast, (b)roadcast, (l)oopback, (i)ndividual.

**format_link_raw() Routine Description**  This routine formats a packet using raw formatting to display upper layer information as hex/ASCII data.

```c
format_link_raw(tl_msg_hdr_type *hdr,
    u_char  *buffer,
    int len,
    int offset,
    char  *linktype,
    char  *interface,
    char  *line3,
    char  *addlinfo)
```

- **hdr**  Pointer to the standard nett1 message header.
- **buffer**  Pointer to entire traced packet (including MAC) use the “offset” parameter to control where the data actually begins printing.
- **len**  Length of the entire buffer.
- **offset**  Offset to actually begin displaying the data; that is, if the MAC information is not to be shown. 802.3 and Ethernet do not display until the beginning of the 802.2 information or the Ethernet data (because the Source and Dest information are formatted out).
- **linktype**  String describing the type of link this information is carried over, such as FDDI, 802.3, 802.5, or Ethernet.
- **interface**  String appended to the device ID and printed out in the interface=[interface] field. Pass a NULL pointer to suppress displaying any interface information.
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

<table>
<thead>
<tr>
<th><strong>field</strong></th>
<th><strong>description</strong></th>
</tr>
</thead>
</table>
| `line3`   | Short string, less than 14 bytes, giving information to be displayed on the same line as the addresses: “Type: 0x800” for Ethernet packets (NOT SNAP) or “Length: 00-1a” for 802.3 packets; may be left blank by passing “”.
| `addlinfo`| (Blank for 802.3 and Ethernet, but may include formatted flags or other information in the MAC header for other link types). Terminated with a newline (\n).

The following shows an example output:

```
vvvv[The linktype parameter goes here]
Received 60 bytes via 802.3 Mon Dec 02 09:22:04:33390 PST 1991
vvvv-[The interface parameter goes here]
        pid=[ICS] interface=[lan0]
        [The line3 parameter goes here]-vvvvvvvvvvvvv
Dest: 09-00-09-00-00-01 Source: 00-00-0c-00-06-31 Length: 00-1a
< [The addlinfo parameter info goes here]
14: fc fc 03 00 00 00 05 03 05 03 03 00 11 00 10 6d c1........m.
30: 00 08 00 06 00 01 0f 0d 6a 3f d8 68 fd f1 0c 20..j?.h...
46: e3 ff 07 50 18 80 00 00 00 00 00 00 00 00 0c 02...P........
```

**Example: Using HP-UX Formatting Library Routines**

The code sample below shows HP-UX formatting library routines including complete description of each line of code.

```c
my_formatter(....)

/* Call for PDU_IN and PDU_OUT trace kinds ONLY */
{
    struct local_hdr_type local_mac_hdr, *my_hdr;
    tl_msg_hdr_type *hdr;
    char temp_buf[80];
    char *buffer, orig_buffer;
    int size, orig_size;
    int ret;

    /* extract TL message header and data */
    /* set buffer and orig_buffer to point to data and */
    /* size and orig_size to the length of traced info */
    ....

    /* now extract MAC specific information from trace info */
```

---

602 Appendix B
memcpys(&local_mac_hdr, buffer, sizeof(struct local_hdr_type));
    my_hdr = &local_mac_hdr;
    
    /* now bump buffer and size to reflect beginning of 802.2 */
    /* information... */
    buffer += LOCAL_HDR_SIZE;
    size -= LOCAL_HDR_SIZE;
    
    /* call setup routine to set up structures reflecting the */
    /* 802.2 and above level headers. Handles SNAP as well */
    set_up_8022(buffer, size, my_hdr->dst_addr, my_hdr->src_addr);
    
    /* the routine filter packet will indicate whether the */
    /* current packet meets the user specified filter criteria */
    /* filter uses the global info setup by set_up_8022. */
    /* (i.e. IP address, 802.2 SAP, Ether type, TCP port... ) */
    if (!filter_packet())
        /* display no info if filter fails */
        return;
    
    /* call the terse formatter if flag set */
    if (terse_fmt) {
        format_link_terse(hdr, buffer, size, "Z", ");
    }
    
    /* always return after terse, the caller only wants 1 line */
    /* of information, so never fall through to format_link_raw */
    return;
}

/* set up "temp_buf" with any short link specific info.*/
/* If we had longer info to pass about the link hdr, */
/* we pass it as the last "addl_info" parameter to */
/* fmt_link_nice() or fmt_link_raw(), in this case we */
/* just pass ",", blank. */
sprintf(temp_buf, "FLAGS: %4x", my_hdr->flags);

/* otherwise call the nice formatter if nice flag set */
if (nice_fmt) {
    ret = format_link_nice(hdr, buffer, size, "802.2", temp_buf,
                           ",", ");
}
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

```c
/* if the nice formatting failed, fall through to */
/* raw formatting. Otherwise return. */
if (ret) return;

format_link_raw(hdr, orig_buffer, orig_size, orig_size-size,
    "802.Z", "zan", temp_buf, "");
return;
```

HP-UX Subsystem Formatter Calls

In case the HP-UX formatting library routines are inadequate for the formatter developer (infrequent), HP-UX also provides a full library of low-level formatting calls for developing a formatter “from the ground up.” This section details these calls.

A formatter developer wishing to use the HP-UX network trace and log data formatting calls to develop a subsystem formatter must include the following in the source code.

```c
#include <fmt.h>
```
This file contains the necessary data structure for the format support calls.

```c
#include <ntl.h>
```
This file contains the necessary data structure for the trace and log data.

```c
#include <subsys_id.h>
```
This file contains subsystem identification information and definitions for log classes and trace kinds.

The following function calls are provided to subsystems for formatting trace and log data and are provided to subsystem formatters in the format library libfmt.sl.

**NOTE**

Subsystems should not link with the libfmt.sl library. All externals are resolved during dynamic loading at run time.
tl_header_format1() This routine is called to format a single trace or log header.

The format of the output will conform to the standard HP-UX network tracing and logging recommendations. The formatted header will be written to the output file referenced by output_file[0].fd. The tl_header_format1() routine must be called by every subformatter. At a minimum, this may be the only output generated by the subformatter. If an error occurs inside the tl_header_format1() routine, −1 is returned. Otherwise, if no errors occurred, 0 is returned. Fatal errors are reported through the return value and the status_ptr parameter. All error messages are written to the file pointed to by the error_fd parameter. The error codes are:

FMTERR_INV_HDR_PTR Trace/log header pointer is invalid.
FMTERR_INV_HDR Trace/log header is invalid (corrupt).
FMTERR_INV_OUT_FD Output file descriptor is invalid.
FMTERR_INV_MC_FD Message catalog descriptor is invalid.
FMTERR_SYS_ERROR An error was returned from a system call within tl_header_format1().

The tl_header_format1() routine is defined as:

```c
int tl_header_format1(char *header_ptr,
                      int error_fd,
                      ss_N_fmt_flag_type flags,
                      char *kind_str,
                      char banner_char,
                      int output_file_count,
                      fd_result output_files[],
                      char *time_buffer,
                      int time_buffer_length,
                      int print_op,
                      int user_count,
                      user_acct_result users[],
                      int location, err_num *status_ptr)
```

**header_ptr** Points to a buffer that contains the header of the trace/log message to be formatted.

**error_fd** File descriptor that refers to the file that will receive error messages.
flags Type of flag is defined as:

```
typedef struct {
    unsigned verbosity_bit: 1;
    unsigned console_logging: 1;
    unsigned highlight_bit: 1;
    unsigned nice_mode_bit: 1;
    unsigned terse_mode_bit: 1;
    unsigned terse_link_mode_bit: 1;
    unsigned terse_time_mode_bit: 1;
    unsigned map_to_names_bit: 1;
    unsigned reserved: 24;
} ss_N_fmt_flag_type;
```

This structure is defined in `/usr/include/fmt.h`.

kind_str May indicate a text message (typically the result of the `tl_log_class` or `tl_trace_kind` function) to be displayed for the kind field from the trace/log header. This string must be null-terminated. The kind message is truncated to 16 characters. If `kind_str` is NULL, the kind field from the header is displayed as a decimal value.

banner_char Character to use in the banner header line (typically the result of the `tl_banner_check` function). The subformatter may use this character to indicate differences in messages, such as inbound or outbound messages. For example, inbound messages could use the character “v”, while outbound messages could use the character “^”.

output_file_count Number of output files to receive the formatted trace/log header output. For HP-UX only one output file is used, and so this value is always 1.

output_files Array of structures consisting of a file descriptor and result variable for each file to receive the formatted trace/log header output. For HP-UX, only one output file is used: `output_file[0].fd`.

time_buffer Contains a string depicting the formatted time stamp from the trace/log header.
Designing a Subformatter

- **time_buffer_length**: Contains the length of `time_buffer` not counting the null terminator byte.

- **print_op**: For HP-UX, this parameter must have a value of 0.

- **user_count**: For HP-UX, this parameter must have a value of 0.

- **users**: For HP-UX, this parameter must have a value of NULL.

- **location**: Value which can be used to locate the source of the message in the code. This parameter is set by the subsystem and may be used to represent any information the subsystem desires.

- **status_ptr**: Contains the error value if the routine returns a −1.

**tl_format_write**() This routine is called to write the decoded buffer to stdout.

The `tl_format_write()` routine prints a buffer pointed to by `output_file[0].fd`. The buffer may be created by one or more calls to the `sprintf()` C library function. If an error occurs inside the `tl_format_write()` routine, −1 is returned. Otherwise (no error occurred), 0 is returned. Fatal errors are reported through the return value and `status_ptr` parameter. All error messages are written to the file pointed to by `error_fd`. The error codes are:

- **FMTERR_FORMAT_WRITE**: An error has occurred in writing to the output files.
- **FMTERR_INV_OUT_FD**: Invalid output file descriptor.
- **FMTERR_INV_L_STR**: Invalid line pointer string.
- **FMTERR_SYS_ERROR**: An error has been returned from a system call within `tl_format_write()` routine.

The `tl_format_write()` routine is defined as:

```c
int tl_format_write(u_char *input_line_ptr,
                    int input_line_byte_count,
                    int error_fd,
                    fmt_wrt_flag_type flags,
                    int output_file_count,
                    fd_result output_files[],
                    int print_op,
                    int user_count,
```
Designing a Subformatter

```c
user_acct_result  users[],
err_num  *status_ptr)
```

`input_line_ptr`  Character string that contains the message to be printed on the outfile files. `input_line_ptr` need not be null-terminated or end with a newline.

`input_line_byte_count`  Byte-count of `input_line_ptr` message string.

`error_fd`  File descriptor pointing to a file to receive error messages from `tl_format_write()` routine.

`flags`  Controls output behavior of the `tl_format_write()` routine. The value must be set before calling `tl_format_write()`.

```c
typedef struct
{
    unsigned  highlight: 1;
    unsigned  wait_to_write: 1;
    unsigned  reserved: 30;
} fmt_wrt_flag_type;
```

`highlight`  Write the `input_line_ptr` data in inverse video.

`wait_to_write`  Reserved for future use.

`output_file_count`  Number of output files to receive the formatted trace/log header output. For HP-UX only one output file is used, and the value is always 1.

`output_files`  Array of structures consisting of a file descriptor and result variable for each file to receive the formatted trace/log header output. For HP-UX only one output file is used; `output_file[0].fd` refers to the file receiving the formatter output.

`print_op`  For HP-UX, this parameter must have a value of 0.

`user_count`  For HP-UX, this parameter must have a value of 0.

`users`  For HP-UX, this parameter must have a value of NULL.

`status_ptr`  Contains the error value if the routine returns a -1.
Designing a Subformatter

**tl_format_fprintf()** This routine is called to convert, format, and print its arguments under control of the format.

It prints the formatted buffer to stdout and is defined as:

```c
int tl_format_fprintf(FILE *stream,
    fmt_wrt_flag_type flags,
    error_num *status_ptr,
    char *format, /* [,arg] */ ...)
```

- **stream** One of the FILE streams contained in the ss_N_fmt_parms_type structure returned by tl_get_parms.
- **flags** Controls the output behavior of the tl_format_fprintf() routine. The value must be set before calling tl_format_fprintf().

```c
typedef struct {
    unsigned highlight : 1;
    unsigned wait_to_write : 1;
    unsigned reserved : 30;
} fmt_wrt_flag_type;
```

- **highlight** Write the format data in inverse video.
- **wait_to_write** Reserved for future use.
- **status_ptr** Contains the error value if the routine returns a −1.
- **format** The format character string contains two types of objects: plain characters that are copied to the output stream, and conversion specifications. Each string results in fetching 0 or more arguments. The results are undefined if there are insufficient args for the format. If the format is exhausted while arguments remain, the excess arguments are ignored.

This routine behaves like printf(). For detail see tl_format_fprintf(NET3).
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

**tl_raw_format()** This routine is called to format a trace or log message into both hexadecimal and printable ASCII characters.

The raw formatted output will appear as follows:

```
0 : 73 61 6d 70 6c 65 5f 6c 6f 67 5f 64 61 74 61 2e sample_log_data
16: 20 6d 6f 72 65 5f 64 61 74 61 20 61 73 64 66 6a more_data asdf
```

The left-most column gives the decimal byte offset. The center area is the hexadecimal display of the data. The right-most column is the printable ASCII display of the data. A period will be displayed for any non-printing character. If an error occurs inside the `tl_raw_format()` routine, a −1 is returned. Otherwise, if no errors occurred, 0 is returned. Fatal errors are reported through the return value and `status_ptr` parameter. All error messages are written to the file pointed to by `error_fd`. The `tl_raw_format()` routine is defined as:

```c
int tl_raw_format(unsigned char *data_ptr,
                  int num_bytes,
                  int start,
                  int error_fd,
                  raw_fmt_flag_type flags,
                  int output_file_count,
                  fd_result output_files[],
                  int print_op,
                  int user_count,
                  user_acct_result users[],
                  err_num *status_ptr)
```

- **data_ptr** Pointer to the buffer that contains the data to be dumped in hexadecimal form.
- **num_bytes** Number of bytes to dump from the buffer pointed to by `data_ptr`. There is no check to ensure that the number of bytes given does not exceed the actual buffer length. If `num_bytes` is zero, then no data will be dumped.
- **start** Offset into the buffer pointed to by `data_ptr` indicating where the dump should begin. If start is zero, the dump will begin at the byte pointed at by `data_ptr`.
- **error_fd** File descriptor that will receive error messages.
- **flags** Reserved for future use. Value should be set to 0 by the caller.
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

output_file_count
Number of output files to receive the raw dump. For HP-UX, this parameter must have a value of 1.

output_files
Array of structures, each of which contains a file descriptor and a result code for the last operation on the file. For HP-UX, only one output file is used; output_file[0].fd refers to the file receiving the formatter output.

print_op
For HP-UX, this parameter must have a value of 0.

user_count
For HP-UX, this parameter must have a value of 0.

users
For HP-UX, this parameter must have a value of NULL.

status_ptr
Contains the error value if the routine returns a −1.

tl_get_parms() This routine returns to the caller a pointer to an ss_N_fmt_parms_type data structure containing parameters that a subsystem subformatter needs in order to operate.

The core formatter builds and initializes this data structure before calling subsys_N_format().

The tl_get_parms() routine is defined as:

ss_N_fmt_parms_type *tl_get_parms()

The ss_N_fmt_parms_type type is defined as:

typedef struct {
    int *ss_status_ptr;
    FILE *ss_output_strm;
    int ss_output_fd;
    FILE *ss_error_strm;
    int ss_error_fd;
    nl_catd ss_msg_cat;
    char *ss_name;
    char *ss_binary_msg_ptr;
    char *ss_options_ptr;
    ss_N_fmt_flag_type ss_n_fmt_flags;
    char *time_buffer;
    int time_buffer_length;
    int output_file_count;
    fd_result output_files[1];
    int print_op;
    int user_count;
} ss_N_fmt_parms_type;
Designing a Subformatter

This data structure is defined in /usr/include/fmt.h, and the parameters are as follows:

`ss_status_ptr` Used by a subformatter to store an error code if it fails.

`ss_output_strm` FILE pointer that will receive the formatted trace/log message. This field must be initialized before calling the `tl_get_parms()` routine.

`ss_output_fd` File descriptor that will receive the formatted trace/log messages.

`ss_error_strm` FILE pointer that will receive any fatal or nonfatal error messages.

`ss_error_fd` File descriptor that will receive any fatal or non fatal error messages.

`ss_msg_cat` Message catalog descriptor to be used in `catgets`.

`ss_name` Pointer to the subsystem name.

`ss_binary_msg_ptr` Pointer to a buffer containing log/trace messages to be formatted.

`ss_options_ptr` Pointer to a buffer containing information to be passed between the `subsys_N_format()` routine. See OptionsPtr in “subsys_N_format()” on page 588.

`ss_n_fmt_flag` Options flags: the type is defined as:

```c
typedef struct {
    unsigned verbosity_bit: 1;
    unsigned console_logging: 1;
    unsigned highlight_bit: 1;
    unsigned nice_mode_bit: 1;
    unsigned terse_mode_bit: 1;
    unsigned terse_link_mode_bit: 1;
    unsigned terse_time_mode_bit: 1;
    unsigned map_to_names_bit: 1;
    unsigned reserved: 24;
} ss_N_fmt_flag_type;
```
Designing a Subformatter

See TimeBuffer in the subsys_N_format function call.

**tl_check_cat_version()** This routine checks that the subsystem message catalog has a compatible version with the subsystem formatter library.

It returns 0 if the versions match, and −1 if they don't, or the file descriptor of the message catalog is invalid.

The *tl_check_cat_version()* routine is defined as:

```c
int tl_check_cat_version(int MsgCatFd, int SetNum, int MsgNum, char *ExpectedVersion, FILE *ErrStrm)
```

*MsgCatFd* File descriptor of the message catalog which contains the version string.

*SetNum* Set number in the message catalog.

*MsgNum* Message number in the message catalog.

*ExpectedVersion* Version string that the message catalog is expected to contain.

*ErrStrm* FILE pointer to a stream that will receive error messages.

**tl_banner_char()** This routine obtains the character to be used when printing a header banner with the *tl_header_format1()* function.

The character is based on the type of log class or trace kind. This function helps to ensure that banners are consistent for all subsystems.

```c
char tl_banner_char(unsigned int kind_class)
```

*kind_class* Trace kind or log class of the message.

**tl_trace_kind()** This routine returns a text interpretation of a trace kind.

The trace kind is stored as an integer. This function converts that number into a string that can be used in the formatted output. For example, passing in a trace kind of 0x80000000 causes the return value to be *HDR_IN_TRACE*. The result of *tl_trace_kind()* is typically used as a parameter to *tl_header_format1()* when printing a header.
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

**Designing a Subformatter**

```c
char *tl_trace_kind(unsigned int kind)
```

`kind` Trace kind of the message.

**tl_log_class()** This routine returns a text interpretation of a log class.

The log class is stored as an integer. This function converts that number into a string that can be used in the formatted output. For example, passing in a log class of 8 causes the return value to be DISASTER. The result of `tl_log_class()` is typically used as a parameter to `tl_header_format1()` when printing a header.

```c
char *tl_log_class(unsigned int class)
```

`class` Log class of the message.

**Subformatter Option**

The formatter options file contains additional information to control the operation of the subformatter. Each line represents the setting of an option. The lines consist of the identifier, which is the same as the subsystem mnemonic, and the arguments recognized by that subformatter.

Options processing is performed as that of the formatter itself; that is, when it recognizes a subsystem mnemonic it passes that line to the subsystem options function. The subsystem options function is responsible for parsing and determining the contents of the line. By the time the options function receives the line, the mnemonic has been stripped off and all strings have been converted to lower case. The only restriction on the contents of the line are that it cannot exceed 2048 bytes and must contain only printable characters. The `tl_get_line()` function (see the *HP-UX Driver Development Reference*) must always be used to read options lines from the options file.

Subsystems may adopt this technique to alter the level of information (beyond terse and verbose), to include extra kinds of data, to provide extra filtering (events or certain trace or log messages for data not covered by the global filtering functions), and so on.

**tl_get_line()** This routine obtains a line from a filter command file according to the following steps:

1. The core formatter reads the filter command file, collects the lines specific to a subsystem and then edits and stores them into a temporary file.
2. It then calls the `subsys_N_get_options()` routine with parameter set as a pointer to this temporary file.

3. The `subsys_N_get_options()` routine can call `tl_get_line()` routine to extract one line at a time from this temporary file for processing.

4. The `tl_get_line()` routine returns 0 for EOF, a negative number for failure, and 2 for success.

It is defined as:

```c
int tl_get_line(
    FILE *stream,
    char *buf,
    int bufsize,
    char *org_buf,
    int LineNo,
    FILE *ErrorStream
);
```

- **Stream** — FILE pointer that points to the temporary filter command file containing a single subsystem's filter commands.
- **buf** — Stores the “cleaned” filter command line in this buffer.
- **bufsize** — Size of Line (no more than 2048).
- **orgbuf** — Stores the original filter command line as it appeared in the filter command file in OrigLine.
- **LineNo** — Stores the line number of OrigLine in LineNo as it appeared in the filter command file.
- **ErrorStream** — FILE pointer to a stream that will receive error messages.

### Internationalization and Message Catalog Support

The formatter provides the subformatters the capability to use the National or Native Language Support (NLS) facilities in HP-UX. When registering the NLS subsystem with the tracing and logging system at installation, the name of the message catalog to be used by the subformatter must also be provided.
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

The message catalog is called as follows:

1. netfmt opens and closes the message catalog by using the catopen() and catclose() calls.

2. The file descriptor returned by the catopen() is passed to the subformatter.

3. If no message catalog is registered, or if the message catalog cannot be opened, a special file descriptor of −1 meaning “no file” is passed in.

The subformatter should perform the appropriate catgets() calls to retrieve their messages from the message catalog. Subsystems should not open their own message catalogs or use multiple message catalogs.

The commonly accepted method of using message catalogs is to use the catgets() call, providing the English language string as the default to the call if the message catalog read fails. This should be the same string the call would retrieve from the default message catalog, located typically in /usr/lib/nls/msg/C/name.cat, where name is the name registered with netfmt.

One recommendation for using message catalogs effectively is to have each logging event correspond to a message number, which makes processing and retrieval simpler. Different message sets or an offset can be used for terse (console) and verbose messages.

Because the message catalogs can be altered for a given location, the subformatter should also put some kind of identifying tag (such as “FTAM 489”) on the message that is not localized. Support personnel in a different location will then be able to understand what is being logged without trying to translate the text of the message.

Due to the subformatter's dependency on message catalogs to provide the correct text for a log event, the version of the catalog is highly dependent on the version of the subsystem. The tl_check_cat_version() function (see tl_check_cat_version(NET3)) is provided to facilitate checking of message catalog versions.

Configuring Developed Subsystems into the System

The process for getting the tracing and logging facility to know about developed subsystems is somewhat complex. Subsystems must inform the tracing and logging facility of their existence at install/update time.
Each fileset is required to have an SDU configure script. Tracing and logging take advantage of this independence to facilitate the configuration of subsystems into the nettl and netfmt commands.

The nettlconf script has the capability to configure the subsystem (see the nettlconf(1M) manpage). nettlconf should be called from within the configure script during an SDU update or installation. nettlconf configures the subsystem information and puts it into the /etc/nettlgen.conf data base file.

The nettl and netfmt commands use the information in the /etc/nettlgen.conf data base file to configure themselves at run time. For the netfmt command, the subsystem's subformatters (in shared libraries) are dynamically loaded so all symbols can be resolved. The nettl command uses the subsystem names and initial log classes to build the tables necessary to control subsystem operations.

The information that the subsystems need to configure include:

- **Subsystem ID:** Assigned to your subsystem by Hewlett-Packard (see “Assign Subsystem ID” on page 285, Chapter 7).
- **Subsystem Mnemonic:** This is the name by which the subsystem will be identified in nettl and in the formatted header printed by netfmt. It is a string that may consist of alphanumerics (beginning with a letter) and may contain underscores. Blanks are not allowed.
- **Default Logging Class:** This is a mask containing the level of logging to be enabled when the logging facility starts up. This level may be changed by subsequent calls to nettl.
- **Subsystem Space Type:** This is a flag that identifies user-space subsystems and kernel-space subsystems. The two types of subsystems are handled differently within the nettl command.
- **Subsystem Formatting Function:** This is the C function name used to call the function that supports formatting for the subsystem. This function must be contained in the subsystem formatter shared library.
- **Subsystem Options Function:** This is the C function that is called to process options specified by the user in the netfmt options file. Only the OTS, LAN, ARPA, and X.25 subsystems use this feature. The formatter uses this function to set up global filtering and formatting information as well. This function must be contained in the subsystem formatter shared library.
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

- Subsystem Group Name: Each subsystem belongs to some logical group, usually a product. This group name is included on the banner printed during formatting. Although this group name can be any ASCII string, it should definitely contain the subsystem product name. For example, all X.25 subsystems use the group name “X.25/9000 Networking”.

- Subsystem Formatter Message Catalog: This is the name of the message catalog used by the subsystem formatter functions. This is typically an unqualified name, that is, the base name of the catalog with no path or .cat extensions. For example, the default message catalog for the formatter is netfmt.cat, and it resides in the default NLS directory, /usr/lib/nls/msg/C. This could be specified simply as netfmt. However, if the message catalog does not reside in the default directory, the message catalog name must contain NLSPATH path constructors described in the environ(5) manpage. For example, for product xyz, the abc message catalog, /opt/xyz/lib/nls/msg/C/abc.cat, would be specified as /opt/xyz/lib/nls/%L/abc.cat. This is so the end user of the formatter can use other message catalogs and control them with the LANG and NLSPATH environment variables. This restriction requires subsystems to load their standard English catalog into the C directory under their nls paths (this is the standard place for the shipped message catalogs).

Sample Subformatter Configure Script  The fileset configure script should perform the configuration of all subsystems contained in the fileset. The following fragment is from an SDU control script to perform the configuration:

```bash
#!/usr/bin/posix/sh
#
# Product:
# Fileset: NETTL-MIN
# configure
# @(#) $Revision: 1.2 $
#
# (c) Copyright Hewlett-Packard Company 1993
#
set -a # Export all vars
exitval=0 # Anticipate success
```
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

```bash
# Extract the configuration file
# Test the configuration file
cmd_output=`$NETFMT -pc /dev/null 2>&1`
```

Appendix B 619
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

**Designing a Subformatter**

```bash
62   cmd_result=$?
63   if [ $cmd_result -ne 0 ]
64       then
65           # The configuration file caused an error
66           echo "ERROR The $NETFMT command produced following error"
67           echo " messages while verifying configuration:"
68           echo "$cmd_output"
69       fi
70   fi
71
72   exit 0
```
Network Trace/Log Subsystem Installation Testing

Subsystem developers must perform complete installation testing on their subsystems. As described in the previous section, the network trace/log facility is configured at installation time by a registration process that occurs in the subsystems configure script. This process tells the `netfmt` and `nettl` commands the IDs of the subsystems that exist on the system and gives information about how the subsystems are to be controlled and formatted. Only those subsystems that are registered are allowed to be turned on for logging and tracing or have their records formatted appropriately.

The `nettlconf` command does not check the parameters that are passed to it. The subsystem must check that the information to be stored in the configuration database is correct. Subsystems must test their installation for all possible environments, including multi-user systems, workstations, and diskless clusters.

NOTE

The registration scheme has the potential to break tracing and logging for all subsystems if the configuration becomes corrupt or if the information that is given is invalid. Subsystems should test and review the procedures used to configure their subsystems into the network trace/log facility.

The problems described in the following list are very common and can cause the configuration file to be unusable. All of these problems are preventable with proper understanding and testing of the subsystem configuration process:

- The subsystem subformatter library or message catalog cannot be found or opened except by superuser.
- A field in the `nettlgen.conf` subsystem configuration database file is corrupted.
- Symbols in the subformatter library conflict with symbols exported from other subformatter libraries of other subsystems. (This situation cannot occur if the +e option to ld is used when creating the subformatter library.)
- Symbols remain unresolved after `netfmt` has loaded the subformatter libraries of all configured subsystems.
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

- The function name of the `subsys_N_format` or `subsys_N_get_options` functions cannot be found in the specified subformatter library.

- The subsystem name or ID is in use by another subsystem. (This cannot happen if subsystems use the subsystem names and ID numbers assigned by the OpenConnect Team as described in “Assign Subsystem ID” on page 285.)