The Zephyr Programmer's Manual

Protocol Version ZEPH0.2

Robert S. French
John T. Kohl

April 5, 1989

Contents

1. Introduction

Zephyr is a notice transport and delivery system developed at MIT Project Athena in 1987. It is an integrated system that provides the ability to send notifications, such as personal messages and system warnings, from one user or system service to another user or group of users.

This manual describes the Zephyr library, the programmer's interface to Zephyr. The Zephyr library consists of a collection of C language functions which allow the programmer to send and receive notices. Additional functions are provided in the library to modify how notices are distributed and to retrieve various pieces of information.

Because this manual is primarily concerned with describing how to write an application which uses Zephyr, the internal workings of the Zephyr system are not discussed in detail. The interested reader can find additional information about Zephyr in the Zephyr design document [1].

This manual is organized into four main sections as follows:

Overview of the Zephyr System. This section describes the various components of Zephyr, and briefly describes how they interact.

General Concepts. This section describes the subscription and user location services.

Programming Standard Applications. This section describes the Zephyr include files, the functions available in the Zephyr library, and many concepts that are relevant to programming Zephyr applications.

Advanced Programming Topics. This section describes more advanced topics, including how to send control messages to Zephyr servers and HostManagers.

In addition, the following appendices are provided:

Additional Examples. Listings of **zwrite**, **zlocate**, and **zstat**, three standard Zephyr applications, are provided.

Error Codes. All Zephyr error codes are listed with a brief description of each.

Function Templates. Contains the templates for all of the functions mentioned in this manual.

2. Manual Conventions

The following typographical conventions are used in this manual:

- A combination of class, class instance, and recipient (used for subscriptions) is written as <CLASS, INSTANCE, RECIPIENT>.
- A function template is written as follows:

  Function template for function:

  ```
  int function (arg1, arg2)
          int     arg1;
          char    *arg2;
  ```

  Prerequisite functions: Any functions that must be called before this one.

  Possible errors: All possible error codes that could be returned.
- During the discussion of a function, arguments are written in bold type, like **arg1**.
- Explicit members of a structure are also written in bold type, like **member**.
- Filenames are written in slanted type, like *filename*.
- Symbols that are defined in an include file are written in the normal type face, like ZERR_NONE.
- Strings that should be entered explicitly are written between quotes, like ‘‘rfrench’’.

3. Overview of the Zephyr System

3.1. Major Divisions

The Zephyr system consists of three primary sections:

- **The Zephyr clients:** These are the applications which actually use Zephyr to accomplish a task. Examples of clients are **zwrite**, which allows a user to send messages to other users, **syslogd**, which can send system warnings to users, and **zwgc**, the WindowGram client, which is the standard way for users to receive incoming notices. Clients are generally written using the Zephyr library, which is the primary subject of this manual.
- **The HostManager:** The HostManager is the intermediary between the clients and the rest of the Zephyr system. There is one HostManager running on every host which supports Zephyr programs. All clients send their outgoing notices to the HostManager. The HostManager then redistributes them to Zephyr servers for final delivery. The HostManager is in charge of determining if a particular server is still operational, and choosing a different server if necessary.
- **The Zephyr servers:** These are the core of the Zephyr system. They are in charge of receiving notices from the clients (via the HostManagers), figuring out which clients or users should receive them, and distributing them. They are also in charge of keeping an up-to-date user location database. There can be any number of servers spread throughout a workstation environment. The servers keep in constant communication, sharing information about changed subscriptions and changed user locations.

The communication between these three components is indicated schematically in Figure 1. One server and two hosts are shown. Each host consists of a HostManager and two clients. The sending host is sending a message which is received by a client on the receiving host. This is a simplified view of the computing environment. Normally there would be hundreds or thousands of hosts, and several servers which keep in constant communication with each other.

When a notice is sent from a client, many events happen ‘‘behind the scenes’’ that are not normally seen by the user. However, these events are very important to an applications programmer, and are listed in simplified form below. Note that many of these actions occur in parallel so this should not be construed as an absolute order of events.

1. The source client program calls a Zephyr library routine which sends a notice.
2. The Zephyr library sends the notice to the HostManager on the same host, and then waits for an acknowledgment.
3. The HostManager receives the notice, and sends back an acknowledgment (HMACK) to the originating client.
4. The Zephyr library receives the acknowledgment and returns to the calling program.
5. The HostManager forwards the notice to a Zephyr server, and appends the notice to its queue of unacknowledged notices.
6. The Zephyr server receives the notice, determines its recipients, and sends back an acknowledgment (SERVACK) to the HostManager.
7. The HostManager receives the acknowledgment, removes the notice from its queue of unacknowledged notices, and forwards a copy of the acknowledgment (SERVACK) to the client program.
8. The client program receives and disposes of the acknowledgment.

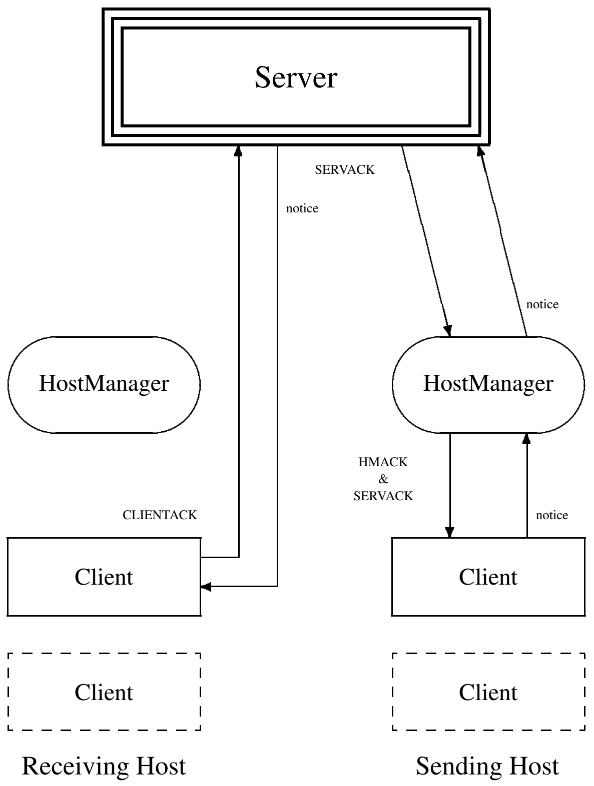

Figure 1: Interaction between the various parts of a Zephyr system.

9. The server forwards the notice to all recipients, and each time appends the notice to its queue of unacknowledged notices.
10. A destination client receives the notice, and sends back an acknowledgment (CLIENTACK) to the server.
11. The server receives the acknowledgment and removes the notice from its queue of unacknowledged notices.

At each stage except the initial client to HostManager communication, if an acknowledgment is not received after a certain length of time, the notice is retransmitted. The initial client to HostManager communication is not retransmitted because there is little chance of a notice being lost when transmitted to a program on the same machine, and thus the lack of an acknowledgment indicates that something is very wrong.

Flags may be set in the notice to indicate how much acknowledgment should be done. The possibilities range from no acknowledgments to the full acknowledgment scenario listed above. A discussion of the different levels of acknowledgment can be found in §5.2.2, and a discussion of the structure of acknowledgments can be found in §5.3.

4. General Concepts

The following pages describe two concepts that may be useful to an applications programmer: The *subscription* paradigm for notice distribution and the *user location* system. The subscription paradigm is used to determine who will receive a given notice. The user location system is used to locate users.

4.1. The Subscription Service

Since the primary purpose of Zephyr is to deliver notices from one user or service to another, an important consideration is the means of specifying the recipient of a notice. An application must be able to send a given notice to a particular recipient, a known group of recipients, or an arbitrary, dynamically changing group of recipients. Zephyr accomplishes this by using a ''subscription'' service. Each Zephyr notice contains three fields which determine its recipients: *class*, *instance*, and *recipient*. Users subscribe to specific triples of class, instance, and recipient, as described below, and these subscriptions are used to determine whether or not a given user will receive a notice.

4.1.1. The class field

The class is the top-level characteristic of a notice. It serves two primary purposes:

- It is used as the first indicator of who might be able to receive the notice
- It is used to determine if the sender is authorized to send a notice of the particular class (see §4.1.5, ''Subscription Authorization'').

For example, a ''MESSAGE'' class might be used to indicate a generic user-to-user message, and a ''FILSRV'' class might be used to indicate a file server message.

4.1.2. The instance field

The *instance* is a subdivision of the *class*. Its primary purpose is to narrow down the subject of the notice. For example, a notice with class ''FILSRV'' might contain the name of the fileserver as its instance. By itself, the instance is not very useful. It is simply an extra string like the class field that is used to determine possible recipients. However, the instance field allows wildcarding at subscription time. This means that a person could subscribe to file server messages from only a particular server by specifying the file server's name as the instance, or all file server messages by specifying instance ''*''.

The only wildcard instance allowed is ''*''. More complicated regular expressions (such as ''*.MIT.EDU'' to match all hosts in the MIT.EDU domain) are not allowed.

4.1.3. The recipient field

The *recipient* is the actual username of the person the notice is intended for. On systems which support the Kerberos authentication system [2], the recipient is the Kerberos principal of the recipient. A Kerberos principal is usually of the form *username@realm*, where *realm* is the name of the Kerberos realm controlling the user's host.

The recipient field may be wildcarded on both the sending and receiving ends. Once again ''*'' is the only valid wildcard. These limitations apply:

- If a user is subscribing to the triple <*class*, *instance*, **username**>, where **username** is the username of the user, only notices with the user's explicit username in their recipient field will be sent to the user.

- If a user is subscribing to the triple <*class*, *instance*, *>, only notices with a recipient of ''*'' will be sent to the user.

Thus, if a user is subscribing to <MESSAGE, PERSONAL, rfrench@ATHENA.MIT.EDU>, and a message is sent to <MESSAGE, PERSONAL, *>, he will not receive it. Likewise, if a person is subscribing to <MESSAGE, PERSONAL, *>, and one is sent to <MESSAGE, PERSONAL, rfrench@ATHENA.MIT.EDU>, he will not receive it. Subscriptions can be combined. Thus if a person is subscribing to both of these triples, he will receive both messages. Note also that a person cannot subscribe to messages destined for users other than himself. These limitations combine to prevent a user from receiving another user's personal messages.

4.1.4. Examples

A notice sent to <MESSAGE, PERSONAL, rfrench@ATHENA.MIT.EDU> will be received by user ''rfrench'' if he subscribes to:

<MESSAGE, PERSONAL, rfrench@ATHENA.MIT.EDU>

<MESSAGE, *, rfrench@ATHENA.MIT.EDU>

But he would not receive it if he subscribed to:

<FOOBAR, PERSONAL, rfrench@ATHENA.MIT.EDU>

<MESSAGE, FOOBAR, rfrench@ATHENA.MIT.EDU>

<MESSAGE, PERSONAL, *>

Likewise, a notice sent to <FILSRV, PARIS.MIT.EDU, *> would be received by someone subscribing to:

<FILSRV, PARIS.MIT.EDU, *>

<FILSRV, *, *>

But would not be received by someone subscribing to:

<FILSRV, PARIS.MIT.EDU, user@ATHENA.MIT.EDU>

<FILSRV, *, user@ATHENA.MIT.EDU>

4.1.5. Subscription Authorization

It is possible for a notice to be authenticated using the Kerberos authentication system [2]. The method used to do this is described in §5.7.5, ''Sending Authenticated Notices.'' When a notice is authenticated, a Zephyr server can perform a number of tests to determine if a user is allowed to send notices to or subscribe to messages from a particular class. The lists which determine these restrictions are kept on the server machines, and may not be updated by users. They are not available for inspection by users.

Some of the restrictions that can be placed on a class are:

- Only specified users can send notices to this class
- Only specified users can subscribe to notices from this class

- Any notice sent to this class must be authenticated
- Any notice sent to this class must have an instance equal to the sender of the notice

A complete list of restrictions and how they are implemented is available in the Zephyr design document [1].

4.1.6. Default Subscriptions

The Zephyr servers maintain a list of default subscriptions which are automatically added to all subscriptions at the first subscription request for a given port. These subscriptions are intended to make sure that the Operations staff can easily notify all users. If you wish to disable these subscriptions, see the section on ZUnsubscribeTo, §5.9.2.

4.2. The User Location Service

In addition to storing subscription information about each user, the Zephyr servers maintain a database of the location of each currently logged-in user. This information is not used by the servers to determine where to send notices—a client can subscribe to notices without being registered in the location database—but is made available to other users for personal use.

Normally a user is registered by a standard client application (such as the WindowGram client) upon login, and is deregistered at logout. However, an application may occasionally desire to modify or remove user location information from the database. The ZSetLocation function (§6.1.1) will do this.

4.2.1. Location Information

The following information is stored by the Zephyr servers for each registered user:

- The name of the host the user is registered from.
- The name of the terminal the user is using; on a workstation that supports the X window system, this may be the display name instead.
- The date and time the user was registered.

4.2.2. Exposure Levels

A user can register at any of a number of *exposure levels*. An exposure level tells the servers who will be able to access information about the user's location, and whether or not the registration will be announced to other users. When a registration can be announced to other users, the Zephyr server sends it to users subscribing to <LOGIN, *user*, *>, where *user* is a fully qualified Kerberos principal (*user@realm*). If Kerberos is not in use, the user field will be *user@host*.

The following exposure levels are defined:

- **None:** The user's location information is completely hidden, and the registration is not announced.
- **Opstaff:** Only members of the site's operational staff can retrieve the user's location information, and the registration is not announced.
- **Realm-Visible:** Only users authenticated in the local Kerberos realm can retrieve the user's location information, and the registration is not announced.

- **Realm-Announced:** Only users authenticated in the local Kerberos realm can retrieve the user's location information, and the registration is announced to all interested users in the local realm.
- **Net-Visible:** All users can retrieve the user's location information. The registration is announced to all interested users in the local realm.
- **Net-Announced:** All users can retrieve the user's location information, and the registration is announced to all interested users.

When Kerberos is not enabled, each host is interpreted as a separate realm for purposes of exposure levels and login notices.

5. Programming Standard Applications

The following sections describe the concepts involved in using the Zephyr library, and the function calls that would be used in a standard Zephyr application.

5.1. The Zephyr Library and Include Files

Applications that want to use the features of Zephyr must link against the Zephyr library and the com_err library, and possibly the Kerberos library and DES library. The Zephyr library contains the functions defined in the later sections of this document. It is usually called `libzephyr.a`, and may be included in an application by specifying `-lzephyr` on the compile or link line.

The com_err library contains error-reporting functions (See the com_err design document [3].). It is usually called `libcomerr.a`, and may be included in an application by specifying `-lcomerr` on the compile or link line.

The Kerberos library contains Kerberos functions. It is usually called `libkrb.a`, and may be included in an application by specifying `-lkrb` on the compile or link line.

The DES library contains DES functions. It is usually called `libdes.a`, and may be included in an application by specifying `-ldes` on the compile or link line.

The main Zephyr include file, `zephyr.h`, must also be included in all source files that use Zephyr functions. It contains many Zephyr-related definitions, and will automatically include zephyr_err.h, the file that contains the error code definitions, plus the system include files <errno.h>, <sys/types.h>, <netinet/in.h>, <sys/time.h>, and <stdio.h>, and if Kerberos is enabled, <krb.h>. `zephyr.h` may be included by specifying the line `#include <zephyr/zephyr.h>` in the source file.

In order for your programs to interact properly with the Zephyr library provided on Project Athena, you *must* specify the `-DKERBEROS` option to the C compiler when compiling your code.

5.1.1. Naming conventions

All routines internal to the Zephyr library are named beginning with "Z_". All routines intended to be used by applications programmers are named beginning with "Z" and do not contain any underscores (_).

5.2. The ZNotice_t Structure

The ZNotice_t structure is the central data object of the Zephyr library. Outgoing notices are first represented in a ZNotice_t structure, and incoming notices are returned in one. A Zephyr notice consists of two primary parts: a header containing information about the notice and a data area. The ZNotice_t structure contains information about both portions of the notice.

5.2.1. Components of the Header

All fields in the ZNotice_t structure are filled in with valid data when a notice is received. However, only some of the fields need to be initialized when a notice is sent. The rest are either filled in automatically by the library, or are filled in with default values if they are NULL. The ZNotice_t structure contains the following fields:

char *z_packet: If the notice has been formatted, or the notice was received, this field points to a buffer containing the formatted version of the notice.

char *z_version: The protocol version the notice was formatted with in the form ZEPH*n.m* where *n* is the major version number and *m* is the minor version number. This manual discusses the functions related to version ZEPH0.2. If a notice is received by a client that supports a different major version, the client will refuse to parse the notice. (This field is filled in automatically when the notice is sent.)

ZNotice_Kind_t z_kind: The type of the notice (see §5.2.2). (This field must be initialized by the client before the notice is sent.)

ZUnique_Id_t z_uid: The unique ID of the notice. This ID is actually a per-transaction unique ID instead of a per-notice unique ID. This is described in more detail in §5.3. (This field is filled in automatically when the notice is sent.)

struct timeval z_time: The time the notice was sent. (This field is not actually part of the notice that is sent, but is derived from the unique ID when the notice is received. Thus it does not need to be filled in at all when the notice is sent.)

unsigned short z_port: The port number on the client from which the notice was sent. (This field must be filled in before the notice is sent. If it is 0, it is automatically filled in with the client's port number.)

int z_auth: An indication of how authentic the notice claims to be. 0 means not authenticated, 1 means authenticated by Kerberos. Note this is an indication of the claim to authenticity, not an indication of the actual authenticity. The function ZCheckAuthentication (§5.10.3.1) verifies or rejects this claim. (This field is filled in automatically when the notice is sent.)

int z_authent_len: The length of the authenticator in **z_ascii_authent**. (This field is filled in automatically when the notice is sent.)

char *z_ascii_authent: The authenticator. This data is used to determine the authenticity of the notice as it is passed from the client program to the server. (It is filled in automatically when the notice is sent.)

char *z_class: The class of the notice. (This field must be filled in before the notice is sent.)

char *z_class_inst: The class instance of the notice. (This field must be filled in before the notice is sent.)

char *z_opcode: The opcode of the notice. This is an extra field with no specified purpose. It may be used to provide extra information between a client and recipient about a notice's contents. The name "opcode" is derived from its initial creation as a place to put an "operation code" about what to do with the notice once it was received. Some internal Zephyr functions use this field. Very few applications actually use it for this purpose, however. (It must be filled in before the notice is sent.)

char *z_sender: The sender of the notice. (If this field is NULL, it will be filled in with the current user's username. It must be set to either a string or NULL before the notice is sent.)

char *z_recipient: The recipient of the notice. (This field must be filled in before the notice is sent.)

char *z_default_format: The default display format for the notice. See the Zephyr design document's chapter on the WindowGram client for more information about the default format string. (This field must be filled in before the notice is sent.)

char *z_multinotice: An indication of what part of a fragmented notice this notice constitutes. (This field will be filled in automatically.)

ZUnique_Id_t z_multiuid: An identification of which fragmented notice this notice is a part of. (This field will be filled in automatically.)

ZChecksum_t z_checksum: The cryptographic checksum of various header fields, used to check the authenticity of the notice as it is passed from the server to the recipient. (This field will be filled in automatically.)

int z_num_other_fields: The number of extra user-defined or unknown fields present in this notice.

char *z_other_fields[Z_MAXOTHERFIELDS]: An array of the extra fields (if any) for this notice.

int z_message_len: The length, in bytes, of the message in **z_message**. (This field must be filled in before the notice is sent. It may be 0 if no message is being included in the notice body.)

char *z_message: The body of the message. (This field must be filled in before the notice is sent. It may be NULL if no message is being included in the message body.)

5.2.2. Notice Kinds

A ZNotice_Kind_t is an enumerated type of one of the following kinds. These kinds may be used in the **z_kind** field of a ZNotice_t structure. Only the first three kinds, UNSAFE, UNACKED, and ACKED, will normally be used by an application program. They are used to deliver a notice to a recipient. The other kinds are included for internal communication between the various portions of a Zephyr system, and the notice is not delivered to a recipient.

UNSAFE: The notice should be delivered. No user acknowledgments will be performed. No acknowledgment is sent from the HostManager to the source client, and the HostManager will not forward the server acknowledgment to the client. Note that the server still sends the HostManager an acknowledgment, and the destination clients still send acknowledgments to the server. UNSAFE is used when a notice needs to be sent, but the client can't afford to wait around for acknowledgments (for example, a system shutdown message).

UNACKED: The notice should be delivered. The HostManager will send an acknowledgment to the client, but will not forward the server's acknowledgment. This is used by clients which wish to make sure that the notice has actually reached the HostManager safely, but don't care if the server ever received it.

ACKED: The notice should be delivered. The HostManager will send an acknowledgment to the client, and will forward the server's acknowledgment. This is used by clients which want to make sure the notice got to the server safely, and perhaps want to know if the server redistributed the notice to any recipients.

HMACK: The notice is an acknowledgment from the HostManager to a client.

HMCTL: The notice is a HostManager control packet, which may be sent to a HostManager to change its state. See §6.7 for more information.

SERVACK: The notice is an acknowledgment from a server to a HostManager, and possibly to a client if the HostManager forwards it.

SERVNAK: The notice is a negative acknowledgment from a server to a HostManager, and possibly to a client if the HostManager forwards it. This notice means that something failed and the server was unable to distribute the original notice.

CLIENTACK: The notice is an acknowledgment from a receiving client to a server. These notices are sent automatically when a notice is received by the Zephyr library.

STAT: The notice contains a request for statistics information from a HostManager. See the section on HostManager communication (§6.7) for more information.

5.2.3. Field Structure of the Notice Body

The **z_message** field of a notice can contain information in any format. The data may be ASCII text, or it may be raw binary data. However, if a notice is to be displayed (by the WindowGram client or a similar application), it should conform to the following standard:

- All data in the message body should be in printable ASCII form. Newline and tab characters are allowed.

- If the information can be conveniently broken into fields, the fields should be concatenated with a NULL between each field and after the last field.
- The **z_message_len** should include all NULLs, including the one after the last field.

The WindowGram client is capable of accepting a notice in this format and displaying it in a user-friendly manner. See the Zephyr design document [1] for more information on the WindowGram client.

5.3. Acknowledgment Structure

An acknowledgment is a notice sent by either the HostManager, a server, or a receiving client to indicate that a notice has been received at some step in the notice relay process. The acknowledgment is identical to the original notice, except for the following differences:

- The **z_kind** field of the notice will be changed as follows:
  - If the notice is a HostManager-to-sending-client acknowledgment, **z_kind** will be set to HMACK.
  - If the notice is a server-to-HostManager acknowledgment, **z_kind** will be set to either SERVACK or SERVNAK depending on whether the notice was handled successfully.
  - If the notice is a receiving-client-to-server acknowledgment, **z_kind** will be set to CLIENTACK.
- Since the only purpose of an acknowledgment is to indicate that a notice has been received successfully, it does not need to contain all of the information included in the original notice. Specifically, the message field and the authenticator fields are left empty in most cases.

The preferred method for a client to determine which original notice the acknowledgment is referencing is to use the **z_uid** field. The **z_uid** of the acknowledgment will be the same as the **z_uid** of the original notice. Thus this field is not a per-packet unique ID, but a per-transaction unique ID.

When a server acknowledges a notice to the HostManager, it includes some additional information in the message body in place of any message that was there before. This information takes the form of a single, NULL-terminated string, as follows:

- **ZSRVACK_SENT**: The notice was sent to at least one recipient. This does not necessarily indicate that the recipient received the notice, but is a good indication that *someone* is subscribing to the class and instance of the notice.
- **ZSRVACK_NOTSENT**: The notice was not sent to any recipients. This means that no one is subscribing to the class and instance of the notice.
- **ZSRVACK_FAIL**: The processing of the notice failed for some reason, most likely because of a lack of proper authentication. This may also be caused by an attempt to send a notice to a restricted class which the user is not authorized to send to.

More information about processing acknowledgments can be found in §5.8.

5.4. Error Handling

Almost all routines in the Zephyr library return an error code of type **Code_t**. This error code may be a UNIX error, a Kerberos error (if Kerberos authentication is enabled), or a Zephyr error. A **Code_t** may be treated as an integer. Each type of error is in a different numeric range (so that they do not interfere with each other). This is accomplished using the **com_err** library. If a routine succeeds, it will return the status ZERR_NONE, which is defined to be zero. A complete list of Zephyr error codes and their meanings can be found in §B, ‘‘Error Codes.’’

The following routines provided by the com_err library may be useful. A full description of the features of the com_err library may be found in the com_err design document [3].

Function template for error_message:

```
char * error_message (code)
        int     code;
```

Return a string containing the error message associated with error **code**.

Function template for com_err:

```
void com_err (whoami, code, message)
        char    *whoami;
        int     code;
        char    *message;
```

Print an error message of the form:

```
<whoami>: <error> <message>
```

Thus if **whoami** were ‘‘zwrite’’, **code** were the error code for ‘‘Internal error’’, and **message** were ‘‘while sending notice’’, the error message printed would be:

```
zwrite: Internal error while sending notice
```

5.5. Initialization

5.5.1. ZInitialize

Function template for ZInitialize:

```
Code_t ZInitialize ()
```

Prerequisite functions: None

Possible errors: ZERR_HMPORT

The Zephyr library must be initialized before it can be used. ZInitialize performs this function. It caches information such as the user’s username, the host’s name, and the HostManager’s port number to speed up future operations, and initializes other internal state variables. If the port for the HostManager is not found in */etc/services*, ZERR_HMPORT is returned.

5.5.2. ZOpenPort

Function template for ZOpenPort:

```
Code_t ZOpenPort (port)
        unsigned short    *port;
```

Prerequisite functions: ZInitialize

Possible errors: UNIX errors, ZERR_PORTINUSE

Before the Zephyr library can perform any operations that require the sending or receiving of notices, a *port* must be allocated for these transactions. The ZOpenPort function accomplishes this. It should be called before any of the functions which send or receive notices. There are three ways to call ZOpenPort:

- With **port** NULL: A port is allocated randomly.
- With **port** pointing to an integer containing 0: A port is allocated randomly, and ***port** is set to the port number.
- With **port** pointing to a a non-zero integer: The port ***port** is allocated, if possible. If ***port** cannot be allocated, ZERR_PORTINUSE is returned.

5.6. Cleaning Up

5.6.1. ZClosePort

Function template for ZClosePort:

```
Code_t ZClosePort ()
```

Prerequisite functions: None

Possible errors: None

The ZClosePort function closes the Zephyr port that was opened with the ZOpenPort function (§5.5.2). If no port was opened with ZOpenPort, or a new file descriptor was set with ZSetFD (§6.2.2), no action is performed. It is not really necessary to call ZClosePort before an application exits, since UNIX will automatically close all open file descriptors when a program exits.

5.6.2. ZCancelSubscriptions

Before an application exits, it should cancel any subscriptions it has made. The ZCancelSubscriptions function (§5.9.3) performs this task.

5.7. Sending Notices

There are a number of functions that will take a notice described by a ZNotice_t structure and transmit it to the current destination address (usually the HostManager). Each of the functions requires the following fields to be initialized in the ZNotice_t structure:

z_kind

z_port (may be 0)

z_class

z_class_inst

z_opcode

z_sender (may be 0)

z_recipient

z_default_format

The rest of the structure should be zeroed out, with code similar to this:

```
ZNotice_t notice;

bzero((char *)&notice, sizeof(notice));
/* initialize the notice after zeroing the entire structure */
notice.z_kind = acked;
```

In addition to the above fields, ZSendNotice requires the **z_message** and **z_message_len** fields to be initialized. The contents of each of these fields, and the default values for those that have them, are described in detail in the ZNotice_t section (§5.2).

5.7.1. ZSendNotice

Function template for ZSendNotice:

```
Code_t ZSendNotice (notice, cert_routine)
        ZNotice_t    *notice;
        int          (*cert_routine)();
```

Prerequisite functions: ZInitialize

Possible errors: Kerberos errors, UNIX errors, ZERR_HEADERLEN, ZERR_ILLVAL, ZERR_HMDEAD, ZERR_BADPKT, ZERR_VERS, ZERR_QLEN

The ZSendNotice function takes the notice described by **notice** and sends it to the current destination address (usually the HostManager). **cert_routine** is a function that will authenticate the message, or is NULL if no such authentication is requested. See §5.7.5 for a description of notice authentication. The fields **z_message** and **z_message_len** must be set in **notice**. If no port has yet been allocated, ZOpenPort(0) (§5.5.2) is automatically called.

If the total header size plus the length of the message exceeds the maximum packet size, the notice will be fragmented into multiple packets; thus there is no limit [1] on notice size. This fragmentation is transparent to the application program.

If the **z_kind** field of **notice** is UNACKED or ACKED, and server mode has not been set (see ZSetServerState, §6.5.1), the library will automatically wait for an acknowledgment from the HostManager. If the acknowledgment does not arrive after thirty seconds, ZERR_HMDEAD is returned.

5.7.2. ZSendList

Function template for ZSendList:

```
Code_t ZSendList (notice, list, nitems, cert_routine)
        ZNotice_t    *notice;
        char         *list[];
        int          nitems;
        int          (*cert_routine)();
```

Prerequisite functions: ZInitialize

Possible errors: Kerberos errors, UNIX errors, ZERR_HEADERLEN, ZERR_ILLVAL, ZERR_HMDEAD, ZERR_BADPKT, ZERR_VERS, ZERR_QLEN

[1] Well, almost no limit. Sending very large notices (> 50K) will be very inefficient using this method. It would be much better to use Zephyr as a rendezvous service (see §6.4).

The ZSendList function performs identically to the ZSendNotice function above, except that the message body is taken from the **list** argument instead of the **z_message** field of **notice**. **nitems** is the number of items (**nitems** ≥ 0) in the **list** array. The items in **list** are concatenated in order with a NULL placed after each one.

5.7.3. Useful Information to Include in a Notice

The following functions provide information that it may be useful to include in the header of a notice.

5.7.3.1. ZGetSender

Function template for ZGetSender:

```
char * ZGetSender ()
```

Prerequisite functions: ZInitialize

Possible errors: None

The ZGetSender function returns the name of the current user. If Kerberos is enabled, ZGetSender will return the Kerberos principal (in the form *username@realm*). If Kerberos is not enabled, or the user is not authenticated, the user's username from */etc/passwd* concatenated with '@' and the official name of the current host (in the form *username@hostname*) will be returned.

5.7.3.2. ZGetRealm

Function template for ZGetRealm:

```
char * ZGetRealm ()
```

Prerequisite functions: ZInitialize

Possible errors: None

The ZGetRealm function returns the realm of the current host. If Kerberos is enabled, ZGetRealm will return the current Kerberos realm. Otherwise, the official name of the current host is returned.

5.7.4. Sending Binary Data

Occasionally it is useful to send binary data in a notice, but to represent the binary data in printable form (for debugging or auditing purposes). The ZMakeAscii function makes this task easy.

5.7.4.1. ZMakeAscii

Function template for ZMakeAscii:

```
Code_t ZMakeAscii (buffer, buffer_len, field, field_len)
        char            *buffer;
        int             buffer_len;
        unsigned char   *field;
        int             field_len;
```

Prerequisite functions: None

Possible errors: ZERR_FIELDLEN

The ZMakeAscii function converts **field_len** bytes of data stored in **field** into printable ASCII and stores them in the **buffer** of length **buffer_len**. If the converted data is longer than **buffer_len** bytes, ZERR_FIELDLEN is returned. The data generated by ZMakeAscii can be converted back to binary with ZReadAscii (§5.10.2.1).

The data generated by ZMakeAscii is in the format "`0xaabbccdd 0xeeffgghh...`", where `aa` is the hexadecimal representation of the first byte in the buffer, `bb` is the second byte, etc.

5.7.5. Sending Authenticated Notices

A notice may be *authenticated* by the Kerberos authentication system [2]. When this is done, the recipient of the notice is guaranteed that the Kerberos principal in the **z_sender** field is the actual sender of the notice. Note that true authentication only works when Kerberos is enabled. If Kerberos is not in use, any notice claiming to be authentic will appear authentic, and there is no guarantee of actual authenticity.

ZMakeAuthentication is the function in the Zephyr library which authenticates a packet under Kerberos. Since this is an internal function and should never be called directly, its template is not given here. The symbols ZAUTH and ZNOAUTH are defined to be ZMakeAuthentication and NULL, respectively. In this way one of these abbreviations may be placed in the **cert_routine** parameter for ZFormatNotice, ZSendNotice, ZSendList or ZSrvSendList to either enable or disable authentication. As new forms of authentication are developed, new abbreviations will be provided.

Authenticated notices should be used only when necessary because of their greater overhead. The time required to compute the DES encryptions is usually around $\frac{1}{4}$ second at each step in the transmission: sender, server, and recipient.

A notice can be checked for authentication with the ZCheckAuthentication function (§5.10.3.1).

5.7.6. Sample Application

This application demonstrates use of ZSendNotice. It initializes the Zephyr library, fills in a notice structure (with the recipient set to the user executing the application), and calls ZSendNotice twice to send the notice, first unauthenticated and then authenticated.

```
#include <zephyr/zephyr.h>

main()
{
    int status;
    ZNotice_t notice;

    /* Initialize the library */
    if ((status = ZInitialize()) != ZERR_NONE) {
        com_err("sample", status, "while initializing");
        exit(1);
    }

    bzero((char *)&notice, sizeof(notice));
    /* initialize the notice after zeroing the entire structure */
    notice.z_kind = UNACKED;    /* We don't care about acknowledgments */
    notice.z_port = 0;  /* Will be filled in by HostManager */
    notice.z_class = "MESSAGE";
    notice.z_class_inst = "PERSONAL";
    notice.z_opcode = "";
    /* The sender will get filled in by the library if 0 */
```

```
    notice.z_sender = (char *)NULL;
    notice.z_recipient = ZGetSender(); /* Send to myself */
    notice.z_default_format = "";

    notice.z_message = "Hello - This is an example!";
    /* Make sure we include the trailing NULL in the length */
    notice.z_message_len = strlen(notice.z_message)+1;

    /* First send the notice unauthenticated */
    /* ZSendNotice will automatically open a port for us */
    if ((status = ZSendNotice(&notice, ZNOAUTH)) != ZERR_NONE)
        com_err("sample", status, "while sending notice");

    /* Send it again, authenticated */
    if ((status = ZSendNotice(&notice, ZAUTH)) != ZERR_NONE) {
        com_err("sample", status, "while sending notice");
        exit(1);
    }
    exit(0);
}
```

5.8. Receiving Acknowledgments

As mentioned in previous sections (§3 and §5.2.2) different kinds of notices trigger various levels of acknowledgment. The acknowledgment from the HostManager to the sending client is handled internally. However, any other acknowledgments that have been requested must be handled by the application program. These will normally consist solely of acknowledgments from the server.

The structure of the returning acknowledgment is described in §5.3. Acknowledgments have the same unique ID as the original notice. Thus, it is desirable to have some way to check or wait for incoming notices with known characteristics (such as a certain unique ID). A simple application may perform the following sequence of tasks repeatedly:

- Send out a notice
- Wait for a server acknowledgment for that notice
- Take action based on the information contained in the acknowledgment
- Repeat

The following functions make this a painless process.

5.8.1. Using Predicates

A *predicate* is a C function which takes a series of arguments and returns a true or false value indicating some relationship (or lack of relationship) between the arguments. All predicates which can be used by the functions described below must take the format:

int *predicate* (notice, arg)
    ZNotice_t   *notice;
    char   *arg;

The function should return 1 if the **notice** satisfies some prespecified relationship with **arg**, 0 otherwise.

5.8.1.1. ZCompareUIDPred and ZCompareMultiUIDPred

The only predicates supplied by the Zephyr library are ZCompareUIDPred and ZCompareMultiUIDPred. Both take a notice and a pointer to a ZUnique_Id_t as arguments. ZCompareUIDPred returns 1 if the unique ID of the notice is the same as the supplied unique ID, or 0 if they are different. ZCompareMultiUIDPred returns 1 if the z_multiuid field of the notice is the same as the supplied unique ID, or 0 if they are different.

5.8.2. ZCheckIfNotice

Function template for ZCheckIfNotice:

```
Code_t ZCheckIfNotice (notice, from, predicate, args)
        ZNotice_t               *notice;
        struct sockaddr_in      *from;
        int                     (*predicate)();
        char                    *args;
```

Prerequisite functions: ZInitialize, ZOpenPort

Possible errors: UNIX errors, ZERR_BADPKT, ZERR_VERS, ZERR_PKTLEN, ZERR_NOPORT, ZERR_QLEN, ZERR_NONOTICE

The ZCheckIfNotice function scans the notice input queue. For each notice, it calls the **predicate** function with the notice and **arg**. If the **predicate** returns 1, ZCheckIfNotice allocates a packet buffer and copies the notice data into this buffer, then parses the packet into **notice**, removes the notice from the queue, and returns ZERR_NONE. If no notice is found which is accepted by the **predicate**, ZCheckIfNotice returns ZERR_NONOTICE. ***from** is filled in with the address of the host which sent the notice.

After a successful return from the ZCheckIfNotice function, ZFreeNotice (§5.8.3) should be called with argument **notice** when the caller has finished using **notice**.

5.8.3. ZFreeNotice

Function template for ZFreeNotice:

```
void ZFreeNotice (notice)
        ZNotice_t    *notice;
```

Prerequisite functions: None

Possible errors: None

ZFreeNotice frees up the storage allocated for a notice by other library routines. ZFreeNotice should only be used for notices returned by library functions whose documentation instructs the programmer to use ZFreeNotice.

5.8.4. ZIfNotice

Function template for ZIfNotice:

```
Code_t ZIfNotice (notice, from, predicate, args)
        ZNotice_t               *notice;
        struct sockaddr_in      *from;
        int                     (*predicate)();
        char                    *args;
```

Prerequisite functions: ZInitialize, ZOpenPort

Possible errors: UNIX errors, ZERR_BADPKT, ZERR_VERS, ZERR_PKTLEN, ZERR_NOPORT, ZERR_QLEN

The ZIfNotice function performs identically to the ZCheckIfNotice function above, except that if there is no notice in the input queue that fits the desired criteria as determined by **predicate**, ZIfNotice will wait until a satisfying notice is received, and then perform as described above for ZCheckIfNotice.

After a successful return from the ZIfNotice function, ZFreeNotice (§5.8.3) should be called with argument **notice** when the caller has finished using **notice**.

5.8.5. ZCompareUID

Function template for ZCompareUID:

```
int ZCompareUID (uid1, uid2)
        ZUnique_Id_t    *uid1;
        ZUnique_Id_t    *uid2;
```

Prerequisite functions: None

Possible errors: None

There are times when the above routines may be inefficient. For example, an application may simply want to accept incoming notices, and decide on the fly whether a notice is an acknowledgment to a previously sent notice. The ZCompareUID function provides an easy means to determine if two unique ID's are the same. It returns 1 if they are the same, and 0 if they are different.

5.8.6. Sample Application

This application demonstrates use of ZSendNotice and ZIfNotice by sending a simple message to a username specified on the command line. It verifies that a username was specified, initializes the library, fills in the notice, sends it, and then waits for an acknowledgment. When the acknowledgment is received, it checks the response and prints a message indicating the response type.

```
#include <zephyr/zephyr.h>

main(argc,argv)
    int argc;
    char *argv[];
{
    ZNotice_t notice, retnotice;
    Code_t retval;

    /* verify username is specified */
    if (argc < 2) {
        fprintf(stderr, "No username specified!\n");
        exit (1);
    }

    /* Initialize the library */
    if ((retval = ZInitialize()) != ZERR_NONE) {
        com_err("sample", retval, "while initializing");
        exit (1);
    }
```

```
    bzero((char *)&notice, sizeof(notice));
    /* initialize the notice after zeroing the entire structure */
    /* Fill in the notice fields */
    notice.z_kind = ACKED;      /* ACKED since we want an acknowledgment */
    notice.z_port = 0;  /* Will be filled in by HostManager */
    notice.z_class = "MESSAGE";
    notice.z_class_inst = "PERSONAL";
    notice.z_opcode = "";
    /* The sender will get filled in by the library if 0 */
    notice.z_sender = (char *)NULL;
    /* Send to the person named on the command line */
    notice.z_recipient = argv[1];
    notice.z_default_format = "";

    notice.z_message = "Hi there!\n";
    /* Make sure we include the trailing NULL in the length */
    notice.z_message_len = strlen(notice.z_message)+1;

    /* Send the notice unauthenticated */
    /* ZSendNotice will automatically open a port for us */
    if ((retval = ZSendNotice(&notice, ZNOAUTH)) != ZERR_NONE) {
        com_err("sample", retval, "while sending notice");
        exit (1);
    }

    /* Wait for the server response. */
    if ((retval = ZIfNotice(&retnotice, (struct sockaddr_in *)0,
                            ZCompareUIDPred,
                            (char *)&notice.z_uid)) != ZERR_NONE) {
            com_err("sample", retval, "while waiting for ack");
            exit (1);
    }

    /* Was there an error? */
    if (retnotice.z_kind == SERVNAK) {
        printf("Received authentication failure while sending\n");
        exit (1);
    }

    /* Check the message body for the acknowledgment information */
    if (!strcmp(retnotice.z_message, ZSRVACK_SENT))
        printf("Message sent!\n");
    else
        printf("%s not receiving messages!\n", argv[1]);

    ZFreeNotice(&retnotice);
    exit (0);
}
```

5.9. Subscribing to Notices

The following functions allow an application to subscribe to notices. The subscription service is described in §4.1.

ZSubscribeTo and ZUnsubscribeTo use the ZSubscription_t structure, which has the following fields:

- **char *class**: The class of the subscription.
- **char *classinst**: The instance of the subscription.

- **char *recipient**: The recipient of the subscription.

5.9.1. ZSubscribeTo

Function template for ZSubscribeTo:

```
Code_t ZSubscribeTo (sublist, nitems, port)
        ZSubscription_t    sublist[];
        int                nitems;
        unsigned short     port;
```

Prerequisite functions: ZInitialize

Possible errors: Kerberos errors, UNIX errors, ZERR_PKTLEN, ZERR_ILLVAL, ZERR_HMDEAD, ZERR_BADPKT, ZERR_VERS, ZERR_QLEN

The ZSubscribeTo function attempts to inform the Zephyr servers that subscriptions for the indicated **port** on the current host should be added. The subscriptions are listed as class/instance/recipient triples in the **sublist** array. **nitems** is the number of entries in the **sublist** array. **port** will usually be the port number of the current application. If **port** is 0, the port number of the current application is substituted.

5.9.2. ZUnsubscribeTo

Function template for ZUnsubscribeTo:

```
Code_t ZUnsubscribeTo (sublist, nitems, port)
        ZSubscription_t    sublist[];
        int                nitems;
        unsigned short     port;
```

Prerequisite functions: ZInitialize

Possible errors: Kerberos errors, UNIX errors, ZERR_PKTLEN, ZERR_ILLVAL, ZERR_HMDEAD, ZERR_BADPKT, ZERR_VERS, ZERR_QLEN

The ZUnsubscribeTo function attempts to inform the Zephyr servers that the specified subscriptions for the indicated **port** on the current host should be deleted. The subscriptions are listed as class/instance/recipient triples in the **sublist** array. **nitems** is the number of entries in the **sublist** array. **port** will usually be the port number of the current application. If **port** is 0, the port number of the current application is substituted.

ZUnsubscribeTo may be useful to remove the server default subscriptions (§4.1.6), which are automatically recorded for every port which has been passed to the ZSubscribeTo function. See §5.15.2 to see how to examine the default subscriptions.

5.9.3. ZCancelSubscriptions

Function template for ZCancelSubscriptions:

```
Code_t ZCancelSubscriptions (port)
        unsigned short   port;
```

Prerequisite functions: ZInitialize

Possible errors: Kerberos errors, UNIX errors, ZERR_PKTLEN, ZERR_ILLVAL, ZERR_HMDEAD, ZERR_BADPKT, ZERR_VERS, ZERR_QLEN

The ZCancelSubscriptions function removes *all* of the subscriptions for the indicated **port**. If **port** is 0, the port number of the current application is substituted.

5.9.4. Subscribing for the WindowGram Client

The WindowGram client is the standard way for users to receive incoming notices. To accomodate this, applications will occasionally want to subscribe to notices on behalf of the WindowGram client; in this way an application can easily add to the types of notices that the user will receive. The **zctl** command is an example of a program that will subscribe on behalf of the WindowGram client.

5.9.4.1. ZGetWGPort

Function template for ZGetWGPort:

```
int ZGetWGPort ()
```

Prerequisite functions: None

Possible errors: -1 = No port number available

The ZGetWGPort function returns the port number associated with the user's WindowGram client. It does this by examining the WGFILE environment variable, and reading the file */tmp/$WGFILE*. If WGFILE is not set, */tmp/wg*.uid, where uid is the UNIX user ID of the user, is examined instead. If neither file could be found, -1 is returned.

The port number returned by ZGetWGPort can be cast to an unsigned short value and used as the **port** argument to ZSubscribeTo, ZUnsubscribeTo, or ZCancelSubscriptions.

5.10. Receiving Notices

The following functions are used to receive and interpret incoming notices.

5.10.1. ZReceiveNotice

Function template for ZReceiveNotice:

```
Code_t ZReceiveNotice (notice, from)
        ZNotice_t           *notice;
        struct sockaddr_in  *from;
```

Prerequisite functions: ZInitialize, ZOpenPort

Possible errors: UNIX errors, ZERR_NOPORT, ZERR_PKTLEN, ZERR_QLEN, ZERR_BADPKT, ZERR_VERS

The ZReceiveNotice function reads and removes the next notice from the input queue. If there are no notices in the queue, it blocks until a notice arrives. A buffer to contain the notice is allocated automatically. **notice** contains the parsed packet. All fields of **notice** are filled in as appropriate. The address from which the notice was received is placed in ***from** if **from** is non-NULL.

After a successful return from the ZReceiveNotice function, ZFreeNotice (§5.8.3) should be called with argument **notice** when the caller has finished using **notice**.

5.10.2. Receiving Binary Data

5.10.2.1. ZReadAscii

Function template for ZReadAscii:

```
Code_t ZReadAscii (buffer, buffer_len, field, field_len)
        char    *buffer;
        int     buffer_len;
        char    *field;
        int     field_len;
```

Prerequisite functions: None

Possible errors: ZERR_BADFIELD

The ZReadAscii function reverses the action of the ZMakeAscii function (§5.7.4.1). It takes the **field_len** bytes in **field**, and converts them back into binary form in **buffer**. **buffer_len** is the number of bytes available in **buffer**. ZERR_BADFIELD is returned if **field** contains data in an improper format or the data will not fit into the buffer.

5.10.3. Receiving Authenticated Notices

When a notice is received, there is no way to immediately discern if it is authenticated or not (the **z_auth** field in the ZNotice_t structure is not necessarily accurate when a notice is received). The ZCheckAuthentication function is provided to verify or deny the authenticity of a notice.

5.10.3.1. ZCheckAuthentication

Function template for ZCheckAuthentication:

```
Code_t ZCheckAuthentication (notice, from)
        ZNotice_t               *notice;
        struct sockaddr_in      *from;
```

Prerequisite functions: ZInitialize, ZReceiveNotice

Possible errors: None

The ZCheckAuthentication function verifies the Kerberos authentication in **notice**. It returns ZAUTH_NO if the notice is not authentic, ZAUTH_FAILED if the notice claimed to be authentic but the authenticity could not be verified, and ZAUTH_YES if the notice was verified to be authentic. ***from** should contain the the address from which the notice was received.

When Kerberos is enabled, this verification uses Kerberos authentication information. When Kerberos is disabled, notices claiming to be authentic cause ZCheckAuthentication to return ZAUTH_YES, and notices not claiming to be authentic cause ZCheckAuthentication to return ZAUTH_NO.

5.10.4. Sample Application

This application demonstrates use of ZOpenPort, ZSubscribeTo, ZReceiveNotice, ZCheckAuthentication and ZFreeNotice. It initializes the library, opens a port, subscribes to messages, and then loops, printing every notice it receives. The program never terminates normally, so it does not cancel its subscriptions. **NOTE**: any messages which contain NULL's in them will not be printed in their entirety, due to the way printf() handles **%s** arguments.

```
#include <zephyr/zephyr.h>

main()
{
    ZSubscription_t sub;
    ZNotice_t notice;
    struct sockaddr_in from;
    Code_t retval;

    /* Initialize the library */
    if ((retval = ZInitialize()) != ZERR_NONE) {
        com_err("sample", retval, "while initializing");
        exit (1);
    }

    /* Open a port, so that ZSubscribeTo has a port to use */
    if ((retval = ZOpenPort((int *) 0)) != ZERR_NONE) {
        com_err("sample", retval, "while opening port");
        exit (1);
    }

    /* Subscribe this client to personal messages */

    sub.class = "MESSAGE";
    sub.classinst = "PERSONAL";
    sub.recipient = ZGetSender();

    if ((retval = ZSubscribeTo(&sub, 1, 0)) != ZERR_NONE) {
        com_err("sample", retval, "while subscribing");
        exit (1);
    }

    /* Loop and accept incoming messages and print them out */

    printf("Now accepting messages...\n");

    for (;;) {
        if ((retval = ZReceiveNotice(&notice, &from)) != ZERR_NONE) {
                com_err("sample", retval, "while receiving notice");
                exit (1);
        }
        retval = ZCheckAuthentication(&notice, &from);
        switch (retval) {
        case ZAUTH_YES:
            printf("** Authenticated message! **\n");
            break;
        case ZAUTH_NO:
            printf("** Unauthenticated message! **\n");
            break;
        case ZAUTH_FAILED:
            printf("** Forged message! **\n");
            break;
        }
        printf("Message from: %s\n", notice.z_sender);
        printf("%s\n", notice.z_message);
        ZFreeNotice(&notice);
    }
```

```
    /* We really should cancel subscriptions when we're finished,
       but this program never finishes, so... */
}
```

5.11. Using the Input Queue

The Zephyr library keeps a queue of notices as they are received. When a packet arrives, it is placed into the queue. If the packet is part of a fragmented notice, the notice is reassembled as the pieces arrive. When all fragments have been received, the count of complete messages is incremented. If no fragments of an incomplete notice arrive during a fixed, short interval after the last fragment was received, all the stored fragments of that notice are discarded. In this way resources are reclaimed if a failure occurs.

The following functions allow an application to test the status of the input queue.

5.11.1. ZPending

Function template for ZPending:

int ZPending ()

Prerequisite functions: ZInitialize, ZOpenPort

Possible errors: UNIX errors, ZERR_MAXQLEN, ZERR_EOF

The ZPending function places any newly received packets into the input queue, and then returns the total number of complete notices available in the queue. The value -1 is returned if an error occurs, with the error code placed in the global variable **errno**.

5.11.2. ZQLength

Function template for ZQLength:

int ZQLength ()

Prerequisite functions: ZInitialize, ZOpenPort

Possible errors: None

The ZQLength function returns the total number of complete notices in the input queue. No new packets are placed into the input queue.

5.11.3. ZPeekNotice

Function template for ZPeekNotice:

```
Code_t ZPeekNotice (notice, from)
        ZNotice_t            *notice;
        struct sockaddr_in   *from;
```

Prerequisite functions: ZInitialize, ZOpenPort

Possible errors: UNIX errors, ZERR_NOPORT, ZERR_PKTLEN, ZERR_QLEN, ZERR_BADPKT, ZERR_VERS

The ZPeekNotice function ''peeks at'' the next notice in the input queue, and returns that notice. It functions identically to ZReceiveNotice (§5.10.1), except that the notice is left in the input queue.

After a successful return from the ZPeekNotice function, ZFreeNotice (§5.8.3) should be called with argument **notice** when the caller has finished using **notice**.

5.11.4. ZPeekIfNotice

Function template for ZPeekIfNotice:

```
Code_t ZPeekIfNotice (notice, from, predicate, args)
        ZNotice_t           *notice;
        struct sockaddr_in  *from;
        int                 (*predicate)();
        char                *args;
```

Prerequisite functions: ZInitialize, ZOpenPort

Possible errors: UNIX errors, ZERR_BADPKT, ZERR_VERS, ZERR_PKTLEN, ZERR_NOPORT, ZERR_QLEN

The ZPeekIfNotice function acts identically to the ZCheckIfNotice function (§5.8.2), except that the notice is left in the input queue.

After a successful return from the ZPeekIfNotice function, ZFreeNotice (§5.8.3) should be called with argument **notice** when the caller has finished using **notice**.

5.12. Using Packets

While notices, in the guise of ZNotice_t structures, are the objects most frequently manipulated by applications, another representation, the ''packet'', is available. A packet is a single buffer which contains all of the information of a ZNotice_t structure. The packet is what is transmitted and received over the network, and is broken down into its constituent fields to become a ZNotice_t structure.

The following functions are provided for applications that want to manipulate or store packets directly. This may be useful since packets are generally easier to store than ZNotice_t structures.[2]

See §5.7 for a description of what fields in a notice need to be initialized before using the following routines.

5.12.1. ZFormatNotice

Function template for ZFormatNotice:

```
Code_t ZFormatNotice (notice, buffer, ret_len, cert_routine)
        ZNotice_t   *notice;
        char        **buffer;
        int         *ret_len;
        int         (*cert_routine)();
```

Prerequisite functions: ZInitialize

[2] The strings in a ZNotice_t structure always contain pointers into the original packet; thus they must stay together. However, a packet can stand on its own and be parsed into a ZNotice_t structure when necessary.

Possible errors: Kerberos errors, UNIX errors, ZERR_PKTLEN, ZERR_ILLVAL

The ZFormatNotice function takes **notice** and formats it into a packet. The packet is stored in a newly allocated buffer. The address of this buffer is returned in ***buffer**. The length of the resulting packet is returned in ***ret_len**. **cert_routine** is called, if non-NULL, as described in §5.7.5, to authenticate the notice and place authentication information into the packet.

After a successful call to ZFormatNotice, the standard C library routine free() should be called with argument ***buffer** when the caller is finished using the packet.

5.12.2. ZFormatNoticeList

Function template for ZFormatNoticeList:

```
Code_t ZFormatNoticeList (notice, list, nitems, buffer, ret_len, cert_routine)
        ZNotice_t   *notice;
        char        *list[];
        int         nitems;
        char        **buffer;
        int         *ret_len;
        int         (*cert_routine)();
```

Prerequisite functions: ZInitialize

Possible errors: Kerberos errors, UNIX errors, ZERR_PKTLEN, ZERR_ILLVAL

The ZFormatNoticeList function is a hybrid between the ZFormatNotice function (§5.12.1) and the ZSendList function (§5.7.2). It formats the notice into a newly allocated packet like ZFormatNotice, but the message body comes from the **nitems** fields of **list**. The fields are concatenated together with a NULL after each field.

After a successful call to ZFormatNoticeList, the standard C library routine free() should be called with argument ***buffer** when the caller is finished using the packet.

5.12.3. ZSendPacket

Function template for ZSendPacket:

```
Code_t ZSendPacket (packet, len, waitforack)
        char   *packet;
        int    len;
        int    waitforack;
```

Prerequisite functions: ZInitialize

Possible errors: UNIX errors, ZERR_ILLVAL, ZERR_HMDEAD, ZERR_BADPKT, ZERR_VERS, ZERR_PKTLEN, ZERR_QLEN

The ZSendPacket function simply transmits the **len** byte long packet in **packet** to the current destination address. If the **waitforack** argument is non-zero, it will wait until it receives a HostManager acknowledgment before returning.

5.12.4. ZReceivePacket

Function template for ZReceivePacket:

```
Code_t ZReceivePacket (buffer, ret_len, from)
```

```
        ZPacket_t             buffer;
        int                   *ret_len;
        struct sockaddr_in    *from;
```

Prerequisite functions: ZInitialize, ZOpenPort

Possible errors: UNIX errors, ZERR_NOPORT, ZERR_PKTLEN, ZERR_QLEN

The ZReceivePacket function reads and removes the next complete packet from the input queue and returns it in **buffer**, which should be the standard Z_MAXPKTLEN bytes long. The actual size of the received packet is returned in ***ret_len**, and the address from which the packet was received is returned in ***from**. If the next complete packet is larger than Z_MAXPKTLEN, ZERR_PKTLEN is returned. If you need to use the packet representation of a notice larger than Z_MAXPTKLEN, use ZReceiveNotice (§5.10.1) and access the **z_packet** field of the notice.

5.12.5. ZPeekPacket

Function template for ZPeekPacket:

```
Code_t ZPeekPacket (buffer, ret_len, from)
        char                  **buffer;
        int                   *ret_len;
        struct sockaddr_in    *from;
```

Prerequisite functions: ZInitialize, ZOpenPort

Possible errors: UNIX errors, ZERR_NOPORT, ZERR_PKTLEN, ZERR_QLEN, ZERR_BADPKT, ZERR_VERS

The ZPeekPacket function ‘‘peeks at’’ the next packet in the input queue. The packet is placed in a newly allocated buffer. The address of the buffer is returned in ***buffer**. The size of the received packet is returned in ***ret_len**, and the address from which the packet was received is returned in ***from**. The packet is not removed from the input queue.

After a successful call to ZPeekPacket, the standard C library routine free() should be called with argument ***buffer** when the caller is finished using the packet.

5.12.6. ZParseNotice

Function template for ZParseNotice:

```
Code_t ZParseNotice (buffer, buffer_len, notice)
        char        *buffer;
        int         buffer_len;
        ZNotice_t   *notice;
```

Prerequisite functions: ZInitialize

Possible errors: ZERR_BADPKT, ZERR_VERS

The ZParseNotice function breaks a packet down into its constituent fields, and builds a ZNotice_t structure. **buffer** contains the **buffer_len** byte long packet to be parsed. The resulting notice is returned in ***notice**.

5.12.7. Sample Application

This application demonstrates use of ZFormatNotice and ZParseNotice. It initializes the library, fills in a notice, formats it, prints the formatted representation, parses the notice, prints the parsed notice, and frees the allocated storage.

```
#include <zephyr/zephyr.h>

main()
{
    ZNotice_t notice, newnotice;
    char *buffer;
    int buffer_len;
    int retval;

    /* Initialize the library */
    if ((retval = ZInitialize()) != ZERR_NONE) {
        com_err("sample", retval, "while initializing");
        exit(1);
    }

    bzero((char *)&notice, sizeof(notice));
    /* initialize the notice after zeroing the entire structure */
    notice.z_kind = ACKED;
    notice.z_port = 0;  /* Will be filled in by HostManager */
    notice.z_class = "MESSAGE";
    notice.z_class_inst = "PERSONAL";
    notice.z_opcode = "";
    /* The sender will get filled in by the library if 0 */
    notice.z_sender = (char *)NULL;
    notice.z_recipient = ZGetSender(); /* myself */
    notice.z_default_format = "This is the default format\n$1";

    notice.z_message = "Hello - This is an example!";
    /* Make sure we include the trailing NULL in the length */
    notice.z_message_len = strlen(notice.z_message)+1;

    /* Format the notice */
    if ((retval = ZFormatNotice(&notice, &buffer, &buffer_len, ZNOAUTH))
        != ZERR_NONE) {
        com_err("sample", retval, "while formatting notice");
        exit(1);
    }

    /* print the formatted packet.
       use write() here so that the entire buffer is printed.  printf() would
       stop at the first NULL.
       Unfortunately, however, the NULL's are normally invisible unless the
       output is piped through /bin/cat -v */
    write(1, buffer, buffer_len);

    /* Parse the notice */
    if ((retval = ZParseNotice(buffer, buffer_len, &newnotice)) != ZERR_NONE) {
        com_err("sample", retval, "while parsing notice");
        exit(1);
    }
    /* Print the fields in the notice */
    printf("\nversion = '%s'\nz_time = %d,%d\nz_port = %u\nz_auth = %d\n\
```

```
z_authent_len = %d\nz_ascii_authent = '%s'\nz_class = '%s'\n\
z_class_inst = '%s'\nz_opcode = '%s'\nz_sender = '%s'\nz_recipient = '%s'\n\
z_default_format = '%s'\nz_multinotice = '%s'\nz_checksum = %lu\n\
z_num_other_fields = %d\nz_other_fields[0] = '%s'\nz_other_fields[1] = '%s'\n\
z_other_fields[2] = '%s'\nz_other_fields[3] = '%s'\n\
z_other_fields[4] = '%s'\nz_other_fields[5] = '%s'\n\
z_other_fields[6] = '%s'\nz_other_fields[7] = '%s'\n\
z_other_fields[8] = '%s'\nz_other_fields[9] = '%s'\n\
z_message_len = %d\nz_message = '%s'\n",
           newnotice.z_version,
           newnotice.z_time.tv_sec, newnotice.z_time.tv_usec,
           newnotice.z_port,
           newnotice.z_auth,
           newnotice.z_authent_len,
           newnotice.z_ascii_authent,
           newnotice.z_class,
           newnotice.z_class_inst,
           newnotice.z_opcode,
           newnotice.z_sender,
           newnotice.z_recipient,
           newnotice.z_default_format,
           newnotice.z_multinotice,
           newnotice.z_checksum,
           newnotice.z_num_other_fields,
           newnotice.z_other_fields[0],
           newnotice.z_other_fields[1],
           newnotice.z_other_fields[2],
           newnotice.z_other_fields[3],
           newnotice.z_other_fields[4],
           newnotice.z_other_fields[5],
           newnotice.z_other_fields[6],
           newnotice.z_other_fields[7],
           newnotice.z_other_fields[8],
           newnotice.z_other_fields[9],
           newnotice.z_message_len,
           newnotice.z_message);
    /* Free storage allocated by ZFormatNotice */
    free(buffer);
    exit(0);
}
```

5.13. Using Raw Notices

It is occasionally useful to be able to transmit packets without having the library fill in missing information for you. For example, the library will usually initialize the **z_uid** and the authentication fields for you. However, if you wanted to send a message with a specific unique ID, as is the case with an acknowledgment, you would not want the library to interfere. The following routines allow an application to handle notices in their "raw", or untouched, form.

5.13.1. ZFormatRawNotice

Function template for ZFormatRawNotice:

Code_t ZFormatRawNotice (notice, buffer, ret_len)

```
        ZNotice_t   *notice;
        char        **buffer;
        int         *ret_len;
```

Prerequisite functions: ZInitialize

Possible errors: ZERR_PKTLEN, ZERR_HEADERLEN

The ZFormatRawNotice function performs identically to the ZFormatNotice function (§5.12.1), except that no part of the notice is changed when it is formatted. The notice in **notice** is formatted into a packet and placed in a newly allocated buffer. The address of this buffer is returned in ***buffer**. The length of the resulting packet is returned in ***ret_len**.

After a successful call to ZFormatRawNotice, the standard C library routine free() should be called with argument ***buffer** when the caller is finished using the packet.

5.13.2. ZFormatSmallRawNotice

Function template for ZFormatSmallRawNotice:

```
Code_t ZFormatSmallRawNotice (notice, buffer, ret_len)
        ZNotice_t   *notice;
        ZPacket_t   buffer;
        int         *ret_len;
```

Prerequisite functions: ZInitialize

Possible errors: ZERR_PKTLEN, ZERR_HEADERLEN

The ZFormatSmallRawNotice function performs identically to the ZFormatRawNotice function above, except that no packet fragmentation is allowed, and if the notice will not fit into a single ZPacket_t buffer, ZERR_PKTLEN is returned. The caller must provide the packet buffer in **buffer**.

5.13.3. ZFormatRawNoticeList

Function template for ZFormatRawNoticeList:

```
Code_t ZFormatRawNoticeList (notice, list, nitems, buffer, ret_len)
        ZNotice_t   *notice;
        char        *list[];
        int         nitems;
        char        **buffer;
        int         *ret_len;
```

Prerequisite functions: ZInitialize

Possible errors: ZERR_PKTLEN, ZERR_HEADERLEN

The ZFormatRawNoticeList function performs identically to the ZFormatNoticeList function (§5.12.2), except that no part of the notice is changed when it is formatted. The notice in **notice** is formatted into a packet and placed in a newly allocated buffer. The address of this buffer is returned in ***buffer**. The body of the notice is take from the **nitems** items in **list**, which are concatenated together with a NULL following each item. The length of the resulting packet is returned in ***ret_len**.

After a successful call to ZFormatRawNoticeList, the standard C library routine free() should be called with argument ***buffer** when the caller is finished using the packet.

5.13.4. ZFormatSmallRawNoticeList

Function template for ZFormatSmallRawNoticeList:

```
Code_t ZFormatSmallRawNoticeList (notice, list, nitems, buffer, ret_len)
        ZNotice_t    *notice;
        char         *list[];
        int          nitems;
        ZPacket_t    buffer;
        int          *ret_len;
```

Prerequisite functions: ZInitialize

Possible errors: ZERR_PKTLEN, ZERR_HEADERLEN

The ZFormatSmallRawNoticeList function performs identically to the ZFormatRawNoticeList function above, except that no packet fragmentation is allowed, and if the notice will not fit into a single ZPacket_t buffer, ZERR_PKTLEN is returned. The caller must provide the packet buffer in **buffer**.

5.13.5. ZSendRawNotice

Function template for ZSendRawNotice:

```
Code_t ZSendRawNotice (notice)
        ZNotice_t    *notice;
```

Prerequisite functions: ZInitialize

Possible errors: UNIX errors, ZERR_PKTLEN, ZERR_HMDEAD, ZERR_BADPKT, ZERR_VERS, ZERR_QLEN

The ZSendRawNotice function performs identically to the ZSendNotice function (§5.7.1), except that no part of the notice is changed when it is formatted, and no authentication is performed.

5.13.6. ZSendRawList

Function template for ZSendRawList:

```
Code_t ZSendRawList (notice, list, nitems)
        ZNotice_t    *notice;
        char         *list[];
        int          nitems;
```

Prerequisite functions: ZInitialize

Possible errors: UNIX errors, ZERR_PKTLEN, ZERR_HMDEAD, ZERR_BADPKT, ZERR_VERS, ZERR_QLEN

The ZSendRawList function performs identically to the ZSendList function (§5.7.2), except that no part of the notice is changed when it is formatted, and no authentication is performed.

5.14. Retrieving User Locations

The following functions allow an application to retrieve information from the user location service (§4.2).

5.14.1. ZLocateUser

Function template for ZLocateUser:

```
Code_t ZLocateUser (user, nlocs)
        char    *user;
        int     *nlocs;
```

Prerequisite functions: ZInitialize

Possible errors: Kerberos errors, UNIX errors, ZERR_PKTLEN, ZERR_ILLVAL, ZERR_HMDEAD, ZERR_BADPKT, ZERR_VERS, ZERR_QLEN

The ZLocateUser function queries the user location service for the location of user **user**. The locations are stored in allocated storage. The number of locations retrieved is returned in ***nlocs**. If another ZLocateUser call is done, the old storage is freed and new storage allocated as necessary.

An error return of ZERR_VERS usually indicates a version mismatch of the following types:

- A new application running on a host with an old HostManager.
- An old server.

The `zstat` program will display the version numbers of both the HostManager and server. Check to be sure the Protocol Version numbers are the same as the protocol version of your application (inspect */usr/include/zephyr/zephyr.h* to find the protocol version numbers of your application).

5.14.2. ZGetLocations

Function template for ZGetLocations:

```
Code_t ZGetLocations (location, numloc)
        ZLocations_t    location[];
        int             *numloc;
```

Prerequisite functions: ZInitialize, ZLocateUser

Possible errors: ZERR_NOLOCATIONS, ZERR_NOMORELOCS

The ZGetLocations function returns the next user location entries that were retrieved with ZLocateUser. If there are no stored locations, ZERR_NOLOCATIONS is returned. ***numloc** should initially contain the maximum number of locations that can fit in the **location** buffer. On return, ***numloc** will contain the number of entries actually returned. Subsequent calls to ZGetLocations will return additional location entries if possible. When there are no more unseen locations in the internal storage, ZERR_NOMORELOCS is returned. As the pointers in the filled-in ZLocations_t structure point into private storage used by ZLocateUser, the contents may change on future calls to ZLocateUser. Thus they should be copied into other storage by the client if necessary.

5.14.3. ZFlushLocations

Function template for ZFlushLocations:

```
Code_t ZFlushLocations ()
```

Prerequisite functions: ZInitialize, ZLocateUser

Possible errors: None

The ZFlushLocations function frees any storage allocated by the most recent ZLocateUser call.

5.14.4. Sample Application

This application demonstrates use of ZLocateUser and ZGetLocations. It verifies that a username was specified on the command line, then initializes the library, queries the server for the locations of the specified user, and prints out any locations found.

```
#include <zephyr/zephyr.h>

main(argc, argv)
    int argc;
    char *argv[];
{
    ZLocations_t location;
    Code_t retval;
    int i, totallocs, numlocs;

    /* verify a username was given on command line */
    if (argc < 2) {
        fprintf(stderr, "No username specified!\n");
        exit (1);
    }

    /* Initialize library */
    if ((retval = ZInitialize()) != ZERR_NONE) {
        com_err("sample", retval, "while initializing");
        exit (1);
    }

    /* Locate the user. */
    if ((retval = ZLocateUser(argv[1], &totallocs)) != ZERR_NONE) {
        com_err("sample", retval, "while locating user");
        exit (1);
    }

    /* complain if not logged in */
    if (totallocs == 0) {
        printf("%s not logged in.\n", argv[1]);
        exit(1);
    }

    /* retrieve each location, one at a time */
    for (i=0; i<totallocs; i++) {
        numlocs = 1;
        if ((retval = ZGetLocations(&location, &numlocs)) != ZERR_NONE) {
            com_err("sample", retval, "while getting location");
            exit (1);
        }
        printf("%s at %s on %s\n", location.host, location.time,
            location.tty);
    }
    exit(0);
}
```

5.15. Retrieving Subscriptions

The following functions allow an application to retrieve information about the user's current subscriptions or the system default subscriptions (§4.1).

5.15.1. ZRetrieveSubscriptions

Function template for ZRetrieveSubscriptions:

```
Code_t ZRetrieveSubscriptions (port, nsubs)
        unsigned short   port;
        int              *nsubs;
```

Prerequisite functions: ZInitialize

Possible errors: Kerberos errors, UNIX errors, ZERR_PKTLEN, ZERR_ILLVAL, ZERR_HMDEAD, ZERR_BADPKT, ZERR_VERS, ZERR_QLEN

The ZRetrieveSubscriptions function queries the server for the current subscriptions for port **port**. The number of subscriptions retrieved is returned in ***nsubs**. If **port** is zero, the port of the current application is substituted. The ZGetWGPort function (§5.9.4.1) can be used to return the port number associated with the user's WindowGram Client.

5.15.2. ZRetrieveDefaultSubscriptions

Function template for ZRetrieveDefaultSubscriptions:

```
Code_t ZRetrieveDefaultSubscriptions (nsubs)
        int    *nsubs;
```

Prerequisite functions: ZInitialize

Possible errors: Kerberos errors, UNIX errors, ZERR_PKTLEN, ZERR_ILLVAL, ZERR_HMDEAD, ZERR_BADPKT, ZERR_VERS, ZERR_QLEN

The ZRetrieveDefaultSubscriptions function queries the server for the default subscriptions. The number of subscriptions retrieved is returned in ***nsubs**.

5.15.3. ZGetSubscriptions

Function template for ZGetSubscriptions:

```
Code_t ZGetSubscriptions (subscription, numsub)
        ZSubscription_t    *subscription;
        int                *numsub;
```

Prerequisite functions: ZInitialize, ZRetrieveSubscriptions or ZRetrieveDefaultSubscriptions

Possible errors: ZERR_NOSUBSCRIPTIONS, ZERR_NOMORESUBSCRIPTIONS

The ZGetSubscriptions function returns the next subscription entries that were retrieved with ZRetrieveSubscriptions or ZRetrieveDefaultSubscriptions. ***numsub** should initially contain the maximum number of subscriptions that can fit in the **subscription** buffer. On return, ***numsub** will contain the number of entries actually returned. Subsequent calls to ZGetSubscriptions will return additional subscription entries if possible. As the pointers in the ZSubscription_t structure may point into ''private'' storage filled in by ZRetrieveSubscriptions or

ZRetrieveDefaultSubscriptions, they may changed on future calls to these functions. Thus they should be copied into other storage by the client if necessary.

5.15.4. ZFlushSubscriptions

Function template for ZFlushSubscriptions:

Code_t ZFlushSubscriptions ()

Prerequisite functions: ZInitialize, ZRetrieveSubscriptions or ZRetrieveDefaultSubscriptions

Possible errors: None

The ZFlushSubscriptions function frees any storage allocated by the ZRetrieveSubscriptions or ZRetrieveDefaultSubscriptions functions. This will be automatically performed one of these functions is called again.

5.15.5. Sample Application

This application demonstrates use of ZGeWGPort, ZRetrieveSubscriptions and ZRetrieveDefaultSubscriptions. It initializes the library, retrieves the WindowGram port (printing an error if it is not able to retrieve the port number), retrieves and prints the subscriptions, and finally retrieves and prints the default subscriptions.

```
#include <zephyr/zephyr.h>

main()
{
    ZSubscription_t subscription;
    Code_t retval;
    int wgport, totalsubs;

    /* Initialize the library */
    if ((retval = ZInitialize()) != ZERR_NONE) {
        com_err("sample", retval, "while initializing");
        exit (1);
    }

    wgport = ZGetWGPort();

    if (wgport == -1)
        printf("Can't retrieve current subscriptions\n");
    else {
        /* retrieve WindowGram's subscriptions */
        if ((retval = ZRetrieveSubscriptions((unsigned short)wgport,
            &totalsubs)) != ZERR_NONE) {
                com_err("sample", retval, "while retrieving subscriptions");
                exit (1);
        }
        printf("Your current subscriptions:\n");
        print_subs(totalsubs);
    }

    /* retrieve default subscriptions */
    if ((retval = ZRetrieveDefaultSubscriptions(&totalsubs)) != ZERR_NONE) {
        com_err("sample", retval,
            "while retrieving default subscriptions");
        exit (1);
```

```
    }
    printf("Default subscriptions:\n");
    print_subs(totalsubs);
    exit(0);
}

print_subs(totalsubs)
    int totalsubs;
{
    ZSubscription_t subscription;
    Code_t retval;
    int i, numsubs;

    for (i=0; i<totalsubs; i++) {
        numsubs = 1;
        if ((retval = ZGetSubscriptions(&subscription,
            &numsubs)) != ZERR_NONE) {
                com_err("sample", retval, "while getting subscription");
                exit (1);
        }
        printf("<%s,%s,%s>\n", subscription.class, subscription.classinst,
            subscription.recipient);
    }
}
```

5.16. Variable Handling

5.16.1. ZGetVariable

Function template for ZGetVariable:

```
char * ZGetVariable (var)
        char    *var;
```

Prerequisite functions: None

Possible errors: NULL = variable not defined

The ZGetVariable function returns the value assigned to the variable **var**. Variable names are case insensitive. Two variable files are searched: the user's private variables file (*~/.zephyr.vars*) and the system default variables file. If the variable is defined in the user's variables file, any system default value is ignored. If the variable is not defined in the user's variables file, but is defined in the system default variables file, the system default value is used. If the variable is not defined in either file, NULL is returned. Errors encountered while opening or reading a variables file are not returned. The pointer returned points to storage internal to the library. The value should be copied before ZGetVariable is called again.

5.16.2. ZSetVariable

Function template for ZSetVariable:

```
Code_t ZSetVariable (var, value)
        char    *var;
        char    *value;
```

Prerequisite functions: None

Possible errors: UNIX errors, ZERR_INTERNAL

The ZSetVariable function sets the value of the variable **var** to **value** in the user's private variables file. If a variable was already present with the same name, it is replaced. Variable names are case insensitive. If the library can't find the user's home directory, an error message is printed and ZERR_INTERNAL is returned. A UNIX error code is returned if an error is encountered while opening, reading, or writing the user's variables file.

5.16.3. ZUnsetVariable

Function template for ZUnsetVariable:

```
Code_t ZUnsetVariable (var)
        char    *var;
```

Prerequisite functions: None

Possible errors: UNIX errors, ZERR_INTERNAL

The ZUnsetVariable function removes the definition of the variable **var** from the user's variables file. If the variable is not defined, no error is returned. Variable names are case insensitive. If the library can't find the user's home directory, an error message is printed and ZERR_INTERNAL is returned. A UNIX error code is returned if an error is encountered while opening, reading, or writing the variables file.

5.16.4. Sample Application

This application demonstrates use of ZSetVariable, ZUnsetVariable, and ZGetVariable. It plays with the "exposure" variable, showing the value, unsetting any private value, showing any default value, and restoring any private value.

```
#include <zephyr/zephyr.h>

main()
{
    char current[100];           /* Assume the exposure is < 100 chars */
    char *value;
    Code_t status;
    int had_exposure = 0;

    /* Retrieve and copy "exposure" variable */
    if ((value = ZGetVariable("exposure")) == NULL)
        printf("No current value for 'exposure'\n");
    else {
        printf("Your value for 'exposure': %s\n", value);
        strcpy(current, value);
        had_exposure = 1;
    }

    /* Unset "exposure" variable */
    if ((status = ZUnsetVariable("exposure")) != ZERR_NONE) {
        com_err("sample", status,
                "while unsetting variable 'exposure'");
        exit(1);
    }
```

```
    /*
     * Retrieve "exposure" variable.  If there is a system default,
     * it should be retrieved here, since exposure was unset above.
     */
    if ((value = ZGetVariable("exposure")) == NULL)
        printf("No default value for 'exposure'\n");
    else
        printf("System default value for 'exposure': %s\n", value);

    /* Re-set exposure if we saved it above */
    if (had_exposure) {
        printf("Setting 'exposure' back to original value\n");
        if ((status = ZSetVariable("exposure", current)) !=
            ZERR_NONE) {
            com_err("sample", status,
                    "while setting variable 'exposure'");
            exit(1);
        }
        /* Verify that the exposure was saved */
        if ((value = ZGetVariable("exposure")) == NULL) {
            printf("Something went really wrong here...\n");
            exit(1);
        } else
            printf("New value for 'exposure': %s\n", value);
    }
    exit(0);
}
```

6. Advanced Programming Topics

The following sections describe functions that will not normally be used by applications, but are available for advanced users.

6.1. Changing Your Location Information

The following functions allow an application to register a user, change a user's exposure level, deregister a user, and flush all locations associated with a certain user from the database.

6.1.1. ZSetLocation

Function template for ZSetLocation:

```
Code_t ZSetLocation (exposure)
        char    *exposure;
```

Prerequisite functions: ZInitialize

Possible errors: Kerberos errors, UNIX errors, ZERR_PKTLEN, ZERR_ILLVAL, ZERR_HMDEAD, ZERR_BADPKT, ZERR_VERS, ZERR_QLEN, ZERR_SERVNAK, ZERR_AUTHFAIL, ZERR_LOGINFAIL

The ZSetLocation function registers the current user with the Zephyr servers. The information that is registered is as follows:

- **user:** The user running the application.
- **host:** The host on which which the application is being run.
- **tty:** The name of the terminal controlling the application (*e.g.*, ttyp5); if the workstation is running the X window system, the display (such as ''unix:0.0'') is used instead.
- **time:** The current time.
- **exposure:** As indicated by the **exposure** argument.

exposure may be one of the following constants (defined in zephyr.h):

EXPOSE_NONE - No exposure

EXPOSE_OPSTAFF - Operational staff exposure

EXPOSE_REALMVIS - Realm visible

EXPOSE_REALMANN - Realm announced

EXPOSE_NETVIS - Net-wide visible

EXPOSE_NETANN - Net-wide announced

More detailed information about the meaning of each exposure level can be found in the ''Exposure Levels'' section (§4.2).

If the user is already registered, this function may be used to change the user's time and exposure information. Note, however, that calling ZSetLocation to change the exposure may generate spurious login notices. If the new exposure is sufficiently broad to allow login notifications (see §4.2.2), a login notice will be sent to any users who have subscribed to such notices.

This function uses an authenticated notice to present the information to the server.

ZERR_NONE is returned if there were no errors. ZERR_INTERNAL is returned if any consistency checks on the acknowledgment failed. ZERR_AUTHFAIL is returned if the server rejected the authentication information in the notice. ZERR_LOGINFAIL is returned if the location information which would have been modified was not already present in the server database. ZERR_SERVNAK is returned if some other server failure occured.

6.1.2. ZUnsetLocation

Function template for ZUnsetLocation:

```
Code_t ZUnsetLocation ()
```

Prerequisite functions: ZInitialize

Possible errors: Kerberos errors, UNIX errors, ZERR_PKTLEN, ZERR_ILLVAL, ZERR_HMDEAD, ZERR_BADPKT, ZERR_VERS, ZERR_QLEN, ZERR_SERVNAK, ZERR_AUTHFAIL, ZERR_LOGINFAIL

The ZUnsetLocation function deletes the user's location information from the location database. Only the information associated with the current host and terminal (or display) are deleted from the database. If the user's exposure was either EXPOSE_REALMANN or EXPOSE_NETANN, a deregistration notice is sent by the server to the recipients subscribed to such notifications. The ''Exposure Levels'' section (§4.2.2) describes exposure levels in detail.

The errors returned and their causes are the same as described above in ZSetLocation.

6.1.3. ZFlushMyLocations

Function template for ZFlushMyLocations:

```
Code_t ZFlushMyLocations ()
```

Prerequisite functions: ZInitialize

Possible errors: Kerberos errors, UNIX errors, ZERR_PKTLEN, ZERR_ILLVAL, ZERR_HMDEAD, ZERR_BADPKT, ZERR_VERS, ZERR_QLEN, ZERR_SERVNAK, ZERR_AUTHFAIL, ZERR_LOGINFAIL

The ZFlushMyLocations function removes all location information about the current user from the location database. This may be used to remove stale data from the database. No deregistration messages are sent.

The errors returned and their causes are the same as described above in ZSetLocation.

6.2. Using Your Own Socket

Applications normally use the socket which is bound by ZOpenPort. However, in some cases it may be desirable to make the Zephyr library to use a socket that the application has already bound. The following functions facilitate this.

6.2.1. ZGetFD

Function template for ZGetFD:

```
int ZGetFD ()
```

Prerequisite functions: ZInitialize or ZSetFD

Possible errors: -1 = No current file descriptor

The ZGetFD function returns file descriptor that the Zephyr library is currently using for the bound socket. If no file descriptor has been assigned yet, -1 is returned.

6.2.2. ZSetFD

Function template for ZSetFD:

```
Code_t ZSetFD (fd)
        int     fd;
```

Prerequisite functions: None

Possible errors: None

The ZSetFD function first closes any port that was opened by ZOpenPort (§5.5.2) and then sets the file descriptor that the Zephyr library will use for all communication to **fd**. This may be useful in an application that needs to open and perform operations on its own port (*e.g.* binding a specific port number) before making it available to the Zephyr library.

6.3. Changing the Destination Address

Applications normally want to send all outgoing notices to the HostManager, which will redistribute them to the proper server. In some cases, though, an application may want to communicate directly with a specific client or server, bypassing the normal routing mechanisms. The following functions facilitate this.

6.3.1. ZGetDestAddr

Function template for ZGetDestAddr:

```
struct sockaddr_in ZGetDestAddr ()
```

Prerequisite functions: ZInitialize

Possible errors: None

The ZGetDestAddr function returns the current destination address used by the Zephyr library. This sockaddr_in (internet address and port number) is the destination for all packets transmitted by the Zephyr library.

6.3.2. ZSetDestAddr

Function template for ZSetDestAddr:

```
Code_t ZSetDestAddr (addr)
```

```
        struct    sockaddr_in *addr;
```

Prerequisite functions: ZInitialize

Possible errors: None

The ZSetDestAddr function sets the destination address for all packets transmitted by the Zephyr library.

6.4. Using Zephyr as a Rendezvous Service

Occasionally there are times when an application needs Zephyr's ability to find and contact users, but needs to send the user a large quantity of information instead of a single message.[3] In this case, Zephyr's notice sending functions (ZSendNotice, *etc.*) prove inefficient because of the overhead involved in packet fragmentation and reassembly, as well as an acknowledgement scheme designed for simple, single-packet transactions. Instead of using Zephyr for the transmission of the information, it can be used as a ''rendezvous service.''

In this case, Zephyr's notice sending capabilities can be used to transmit a host and port number (as part of the message body in a notice). Once these are received, a more efficient TCP/IP connection can be established for the actual data transmission. The IPC tutorial ([4]) gives an introduction describing how to establish a TCP/IP connection under 4.3BSD.

6.5. Server Functions

The following functions are normally only used by the Zephyr server and the HostManager.

6.5.1. ZSetServerState

Function template for ZSetServerState:

```
Code_t ZSetServerState (state)
        int    state;
```

Prerequisite functions: ZInitialize

Possible errors: None

If the Zephyr library is to be used to perform Server or Hostmanager functions, the behavior of various internal routines needs to be modified. If **state** is non-zero, the server behavior is selected. If **state** is zero, the server behavior is inhibited.

By default, the Zephyr library assumes non-server behavior is desired.

6.5.2. ZSrvSendNotice

Function template for ZSrvSendNotice:

```
Code_t ZSrvSendNotice (notice, cert_routine, send_routine)
        ZNotice_t    *notice;
        int          (*cert_routine)();
        int          (*send_routine)();
```

[3] A good example of this is Project Athena's On-Line Consulting System, which needs to send entire conversations between users.

Prerequisite functions: ZInitialize

Possible errors: Kerberos errors, UNIX errors, ZERR_HEADERLEN, ZERR_ILLVAL, ZERR_HMDEAD, ZERR_BADPKT, ZERR_VERS, ZERR_QLEN

The ZSrvSendNotice function performs identically to the ZSendNotice function (§5.7.1), except that **send_routine** is used to transmit each fragment of the resulting notice. **send_routine** will be called as follows:

```
Code_t send_routine (notice, buffer, len, waitforack)
        ZNotice_t    *notice;
        char         *buffer;
        int          len;
        int          waitforack;
```

notice points to the notice fragment corresponding to the formatted fragment. It is provided in case **send_routine** wishes to examine fields, such as uniqe ID's, before transmitting. **buffer** points to a formatted fragment to be transmitted. **len** is the length of the fragment. **waitforack** is non-zero if the notice kind is UNACKED or ACKED and server mode has not been set.

6.5.3. ZSrvSendList

Function template for ZSrvSendList:

```
Code_t ZSrvSendList (notice, list, nitems, cert_routine, send_routine)
        ZNotice_t    *notice;
        char         *list[];
        int          nitems;
        int          (*cert_routine)();
        int          (*send_routine)();
```

Prerequisite functions: ZInitialize

Possible errors: Kerberos errors, UNIX errors, ZERR_HEADERLEN, ZERR_ILLVAL, ZERR_HMDEAD, ZERR_BADPKT, ZERR_VERS, ZERR_QLEN

The ZSrvSendList function performs identically to the ZSendList function (§5.7.2), except that **send_routine** is used to transmit each fragment of the resulting notice. **send_routine** will be called as described above for ZSrvSendNotice.

6.5.4. ZSrvSendRawList

Function template for ZSrvSendRawList:

```
Code_t ZSrvSendRawList (notice, list, nitems, send_routine)
        ZNotice_t    *notice;
        char         *list[];
        int          nitems;
        int          (*send_routine)();
```

Prerequisite functions: ZInitialize

Possible errors: UNIX errors, ZERR_PKTLEN, ZERR_HMDEAD, ZERR_BADPKT, ZERR_VERS, ZERR_QLEN

The ZSrvSendRawList function performs identically to the ZSendRawList function (§5.13.6), except that **send_routine** is used to transmit each fragment of the resulting notice. **send_routine** will be called as described above for ZSrvSendNotice.

6.5.5. ZFormatAuthenticNotice

Function template for ZFormatAuthenticNotice:

```
Code_t ZFormatAuthenticNotice (notice, buffer, buffer_len, ret_len, session)
        ZNotice_t   *notice;
        char        *buffer;
        int         buffer_len;
        int         *ret_len;
        C_Block     session;
```

Prerequisite functions: ZInitialize

Possible errors: Kerberos errors, UNIX errors, ZERR_PKTLEN, ZERR_HEADERLEN, ZERR_ILLVAL

The ZFormatAuthenticNotice function takes **notice** and formats it into a packet. The packet is stored in the user-supplied buffer ***buffer**. **buffer_len** should be the size of the buffer. If the notice will not fit in the buffer, an error is returned. The length of the resulting packet is returned in ***ret_len**. The DES library routine **quad_cksum** is called to checksum the appropriate portions of the notice, using **session** as the session key, thus generating a cryptographic checksum. This checksum is stored as the checksum in the packet. When **session** is shared between the sender and recipient of the notice, the authentication can be verified using ZCheckAuthentication (§5.10.3.1).

6.6. Communicating with the WindowGram Client

The following sections describe control notices that can be sent to a WindowGram client to ask it to perform various functions.

6.6.1. Where to Send Notices

Notices should be sent to the WindowGram's port on the local machine. Thus the destination address should be set with code such as:

```
struct sockaddr_in newsin;

newsin = ZGetDestAddr();
newsin.sin_port = (unsigned short) ZGetWGPort();
ZSetDestAddr(&newsin);
```

(Of course, suitable error checking should be included.)

6.6.2. Available Commands

The following functions are available. They are listed in the form [kind, class, instance, opcode]. The rest of the notice (*e.g.* the message body) has no effect on the requested action.

- [UNSAFE, WG_CTL_CLASS, WG_CTL_USER, USER_REREAD]: Reread the user's *.zephyr.desc* file.
- [UNSAFE, WG_CTL_CLASS, WG_CTL_USER, USER_SHUTDOWN]: Save subscriptions, unsubscribe to all notices, and cease processing all incoming notices except a USER_STARTUP control notice.
- [UNSAFE, WG_CTL_CLASS, WG_CTL_USER, USER_STARTUP]: Re-subscribe to saved subscriptions, and continue processing incoming notices (this is only useful after a USER_SHUTDOWN request.).

6.6.3. Sample Application

This application demonstrates sending a control notice to a WindowGram client to ask it to re-read the user's *.zephyr.desc* file. It initializes the library, changes the destination address to the WindowGram client's port, fills in the control notice, and sends it off. It does not wait for any acknowledgement.

```
#include <zephyr/zephyr.h>

main()
{
    ZNotice_t notice;
    struct sockaddr_in newsin;
    Code_t retval;
    int newport;

    /* Initialize the library */
    if ((retval = ZInitialize()) != ZERR_NONE) {
        com_err("sample", retval, "while initializing");
        exit (1);
    }

    /* Change the destination to be the user's WindowGram client */
    newsin = ZGetDestAddr();
    if ((newport = ZGetWGPort()) == -1) {
        fprintf(stderr, "Can't find WindowGram port\n");
        exit (1);
    }
    newsin.sin_port = (unsigned short) newport;
    ZSetDestAddr(&newsin);

    bzero((char *)&notice, sizeof(notice));
    /* initialize the notice after zeroing the entire structure */
    /* Fill in notice for a REREAD command */
    notice.z_kind = UNSAFE;
    notice.z_port = 0;
    notice.z_class = WG_CTL_CLASS;
    notice.z_class_inst = WG_CTL_USER;
    notice.z_opcode = USER_REREAD;
    notice.z_sender = 0;
    notice.z_recipient = "";
    notice.z_default_format = "";
    notice.z_message = "";
    notice.z_message_len = 0;

    /* Send the command */
    if ((retval = ZSendNotice(&notice, ZNOAUTH)) != ZERR_NONE) {
        com_err("sample", retval, "while sending notice");
        exit (1);
    }
    exit(0);
}
```

6.7. Communicating with the HostManager

The following sections describe control notices that can be sent to a HostManager to ask it to perform various functions.

6.7.1. Where to Send Notices

Notices should be sent to the HostManager port on the desired host. Since the default destination is the HostManager on the local host, the destination need not be changed if the control message is destined for the local host's HostManager. Otherwise, the port may be found by looking up port name ‘‘zephyr-hm’’, protocol ‘‘udp’’ using the C library routine getservent().

6.7.2. Available Commands

The following functions are available. They are listed in the form [kind, class, instance, opcode]. The rest of the notice (*e.g.* the message body) has no effect on the requested action. Functions which are only for internal use are not described here.

- [HMCTL, HM_CTL_CLASS, HM_CTL_CLIENT, CLIENT_FLUSH]: Send a state-flush command to the current server.
- [HMCTL, HM_CTL_CLASS, HM_CTL_CLIENT, CLIENT_NEW_SERVER]: Find a new server.
- [STAT, HM_STAT_CLASS, HM_STAT_CLIENT, HM_GIMMESTATS]: Reply with a notice containing statistics information.

6.7.3. Sample Application

This application demonstrates sending a control notice to a HostManager, asking it to find a new server. The application initializes the library, fills in the control notice, and sends it off. It does not wait for any acknowledgement.

```
#include <zephyr/zephyr.h>

main()
{
    ZNotice_t notice;
    Code_t retval;

    /* Initialize library */
    if ((retval = ZInitialize()) != ZERR_NONE) {
        com_err("sample", retval, "while initializing");
        exit (1);
    }

    bzero((char *)&notice, sizeof(notice));
    /* initialize the notice after zeroing the entire structure */
    /* Fill in notice for a FIND-NEW-SERVER command */
    notice.z_kind = HMCTL;
    notice.z_port = 0;
    notice.z_class = HM_CTL_CLASS;
    notice.z_class_inst = HM_CTL_CLIENT;
    notice.z_opcode = CLIENT_NEW_SERVER;
    notice.z_sender = 0;
    notice.z_recipient = "";
    notice.z_default_format = "";
    notice.z_message = "";
    notice.z_message_len = 0;

    /* send it off to the local HostManager */
    if ((retval = ZSendNotice(&notice, ZNOAUTH)) != ZERR_NONE) {
```

```
        com_err("sample", retval, "while sending notice");
        exit (1);
    }
    exit(0);
}
```

A. Additional Examples

The following three sections contain the source code for three typical Zephyr applications: zwrite, zlocate, and zstat.

A.1. zwrite

```
/* This file is part of the Project Athena Zephyr Notification System.
 * It contains code for the "zwrite" command.
 *
 *      Created by:     Robert French
 *
 *      Source: /mit/zephyr/src/clients/zwrite/RCS/zwrite.c,v
 *      Author: jtkohl
 *
 *      Copyright (c) 1987,1988 by the Massachusetts Institute of Technology.
 *      For copying and distribution information, see the file
 *      "mit-copyright.h".
 */

#include <zephyr/mit-copyright.h>

#include <zephyr/zephyr.h>
#include <string.h>
#include <netdb.h>

#ifndef lint
static char rcsid_zwrite_c[] =
 "Header: zwrite.c,v 1.24 88/08/01 14:13:55 jtkohl Exp ";
#endif lint

#define DEFAULT_CLASS "MESSAGE"
#define DEFAULT_INSTANCE "PERSONAL"
#define URGENT_INSTANCE "URGENT"
#define FILSRV_CLASS "FILSRV"

#define MAXRECIPS 100

int nrecips, msgarg, verbose, quiet;
char *whoami, *inst, *class, *recips[MAXRECIPS];
int (*auth)();
void un_tabify();

extern char *malloc(), *realloc();
char *fix_filsrv_inst();

main(argc, argv)
    int argc;
    char *argv[];
{
    ZNotice_t notice;
    int retval, arg, nocheck, nchars, msgsize, filsys, tabexpand;
    char bfr[BUFSIZ], *message, *signature;
    char classbfr[BUFSIZ], instbfr[BUFSIZ], sigbfr[BUFSIZ];

    whoami = argv[0];
```

```
    if ((retval = ZInitialize()) != ZERR_NONE) {
        com_err(whoami, retval, "while initializing");
        exit(1);
    }

    if (argc < 2)
        usage(whoami);

    bzero((char *) &notice, sizeof(notice));

    auth = ZAUTH;
    verbose = quiet = msgarg = nrecips = nocheck = filsys = 0;
    tabexpand = 1;

    if (class = ZGetVariable("zwrite-class")) {
        (void) strcpy(classbfr, class);
        class = classbfr;
    }
    else
        class = DEFAULT_CLASS;
    if (inst = ZGetVariable("zwrite-inst")) {
        (void) strcpy(instbfr, inst);
        inst = instbfr;
    }
    else
        inst = DEFAULT_INSTANCE;
    signature = ZGetVariable("zwrite-signature");
    if (signature) {
        (void) strcpy(sigbfr, signature);
        signature = sigbfr;
    }

    arg = 1;

    for (;arg<argc&&!msgarg;arg++) {
        if (*argv[arg] != '-') {
            recips[nrecips++] = argv[arg];
            continue;
        }
        if (strlen(argv[arg]) > 2)
            usage(whoami);
        switch (argv[arg][1]) {
        case 'a':               /* Backwards compatibility */
            break;
        case 'o':
            class = DEFAULT_CLASS;
            inst = DEFAULT_INSTANCE;
            break;
        case 'd':
            auth = ZNOAUTH;
            break;
        case 'v':
            verbose = 1;
            break;
        case 'q':
            quiet = 1;
            break;
        case 'n':
            nocheck = 1;
```

```
            break;
        case 't':
            tabexpand = 0;
            break;
        case 'u':
            inst = URGENT_INSTANCE;
            break;
        case 'i':
            if (arg == argc-1 || filsys == 1)
                usage(whoami);
            arg++;
            inst = argv[arg];
            filsys = -1;
            break;
        case 'c':
            if (arg == argc-1 || filsys == 1)
                usage(whoami);
            arg++;
            class = argv[arg];
            filsys = -1;
            break;
        case 'f':
            if (arg == argc-1 || filsys == -1)
                usage(whoami);
            arg++;
            class = FILSRV_CLASS;
            inst = fix_filsrv_inst(argv[arg]);
            filsys = 1;
            break;
        case 'm':
            if (arg == argc-1)
                usage(whoami);
            msgarg = arg+1;
            break;
        default:
            usage(whoami);
        }
    }

    if (!nrecips && !(strcmp(class, DEFAULT_CLASS) ||
                      strcmp(inst, DEFAULT_INSTANCE))) {
        fprintf(stderr, "No recipients specified.\n");
        exit (1);
    }

    notice.z_kind = ACKED;
    notice.z_port = 0;
    notice.z_class = class;
    notice.z_class_inst = inst;
    notice.z_opcode = "PING";
    notice.z_sender = 0;
    notice.z_message_len = 0;
    notice.z_recipient = "";
    if (filsys == 1)
            notice.z_default_format = "\
@bold(Filesystem Operation Message for $instance:)\n\
From: @bold($sender)\n$message";
    else if (auth == ZAUTH)
        notice.z_default_format = "Class $class, Instance $instance:\n\
```

```
@center(To: @bold($recipient))\n$message";
    else
        notice.z_default_format =
            "@bold(UNAUTHENTIC) Class $class, Instance $instance:\n$message";

    if (!nocheck && !msgarg && filsys != 1)
        send_off(&notice, 0);

    if (!msgarg && isatty(0))
        printf("Type your message now.  \
End with control-D or a dot on a line by itself.\n");

    message = NULL;
    msgsize = 0;
    if (signature) {
        message = malloc((unsigned)(strlen(signature)+sizeof("From: ")+2));
        (void) strcpy(message, "From: ");
        (void) strcat(message, signature);
        msgsize = strlen(message)+1;
    }

    if (msgarg) {
        int size = msgsize;
        for (arg=msgarg;arg<argc;arg++)
                size += (strlen(argv[arg]) + 1);
        size++;                         /* for the newline */
        if (message)
                message = realloc(message, (unsigned) size);
        else
                message = malloc((unsigned) size);
        for (arg=msgarg;arg<argc;arg++) {
            (void) strcpy(message+msgsize, argv[arg]);
            msgsize += strlen(argv[arg]);
            if (arg != argc-1) {
                message[msgsize] = ' ';
                msgsize++;
            }
        }
        message[msgsize] = '\n';
        message[msgsize+1] = '\0';
        msgsize += 2;
    } else {
        if (isatty(0)) {
            for (;;) {
                if (!fgets(bfr, sizeof bfr, stdin))
                    break;
                if (bfr[0] == '.' &&
                    (bfr[1] == '\n' || bfr[1] == '\0'))
                    break;
                if (message)
                        message = realloc(message,
                                      (unsigned)(msgsize+strlen(bfr)));
                else
                        message = malloc((unsigned)(msgsize+strlen(bfr)));
                (void) strcpy(message+msgsize, bfr);
                msgsize += strlen(bfr);
            }
            message = realloc(message, (unsigned)(msgsize+1));
            message[msgsize] = '\0';
```

```
        }
        else {  /* Use read so you can send binary messages... */
            while (nchars = read(fileno(stdin), bfr, sizeof bfr)) {
                if (nchars == -1) {
                    fprintf(stderr, "Read error from stdin!  Can't continue!\n");
                    exit(1);
                }
                message = realloc(message, (unsigned)(msgsize+nchars));
                bcopy(bfr, message+msgsize, nchars);
                msgsize += nchars;
            }
        }
    }

    notice.z_opcode = "";
    if (tabexpand)
        un_tabify(&message, &msgsize);
    notice.z_message = message;
    notice.z_message_len = msgsize;

    send_off(&notice, 1);
    exit(0);
}

send_off(notice, real)
    ZNotice_t *notice;
    int real;
{
    int i, success, retval;
    char bfr[BUFSIZ];
    ZNotice_t retnotice;

    success = 0;

    for (i=0;i<nrecips || !nrecips;i++) {
        notice->z_recipient = nrecips?recips[i]:"";
        if (verbose && real)
            printf("Sending %smessage, class %s, instance %s, to %s\n",
                   auth?"authenticated ":"",
                   class, inst,
                   nrecips?notice->z_recipient:"everyone");
        if ((retval = ZSendNotice(notice, auth)) != ZERR_NONE) {
            (void) sprintf(bfr, "while sending notice to %s",
                    nrecips?notice->z_recipient:inst);
            com_err(whoami, retval, bfr);
            break;
        }
        if ((retval = ZIfNotice(&retnotice, (struct sockaddr_in *) 0,
                                ZCompareUIDPred,
                                (char *)&notice->z_uid)) !=
            ZERR_NONE) {
            ZFreeNotice(&retnotice);
            (void) sprintf(bfr, "while waiting for acknowledgement for %s",
                    nrecips?notice->z_recipient:inst);
            com_err(whoami, retval, bfr);
            continue;
        }
        if (retnotice.z_kind == SERVNAK) {
            printf("Received authorization failure while sending to %s\n",
```

```
                  nrecips?notice->z_recipient:inst);
            ZFreeNotice(&retnotice);
            break;                      /* if auth fails, punt */
        }
        if (retnotice.z_kind != SERVACK || !retnotice.z_message_len) {
            printf("Detected server failure while receiving \
acknowledgement for %s\n",
                  nrecips?notice->z_recipient:inst);
            ZFreeNotice(&retnotice);
            continue;
        }
        if (!real || (!quiet && real))
            if (!strcmp(retnotice.z_message, ZSRVACK_SENT)) {
                if (real) {
                    if (verbose)
                        printf("Successful\n");
                    else
                        printf("%s: Message sent\n",
                               nrecips?notice->z_recipient:inst);
                }
                else
                    success = 1;
            }
            else
                if (!strcmp(retnotice.z_message,
                            ZSRVACK_NOTSENT)) {
                    if (verbose && real) {
                        if (strcmp(class, DEFAULT_CLASS))
                            printf("Not logged in or not subscribing to\
class %s, instance %s\n",
                                   class, inst);
                        else
                            printf("Not logged in or not subscribing to \
messages\n");
                    }
                    else
                        if (!nrecips)
                            printf("No one subscribing to class %s, instance %s\n",
                                   class, inst);
                        else {
                            if (strcmp(class, DEFAULT_CLASS))
                                printf("%s: Not logged in or not subscribing\
to class %s, instance %s\n",
                                       notice->z_recipient, class, inst);
                            else
                                printf("%s: Not logged in or not subscribing \
to messages\n",
                                       notice->z_recipient);
                        }
                }
                else
                    printf("Internal failure - illegal message field \
in server response\n");
        ZFreeNotice(&retnotice);
        if (!nrecips)
            break;
    }
    if (!real && !success)
        exit(1);
```

```
}

usage(s)
    char *s;
{
    printf("Usage: %s [-a] [-d] [-v] [-q] [-u] [-o] \
[-c class] [-i inst] [-f fsname]\n\t[user ...] [-m message]\n", s);
    printf("\t-f and -c are mutually exclusive\n\
\t-f and -i are mutually exclusive\n");
    exit(1);
}

/*
  if the -f option is specified, this routine is called to canonicalize
  an instance of the form hostname[:pack].  It turns the hostname into the
  name returned by gethostbyname(hostname)
 */

char *fix_filsrv_inst(str)
char *str;
{
        static char fsinst[BUFSIZ];
        char *ptr;
        struct hostent *hp;

        ptr = index(str,':');
        if (ptr)
                *ptr = '\0';

        hp = gethostbyname(str);
        if (!hp) {
                if (ptr)
                        *ptr = ':';
                return(str);
        }
        (void) strcpy(fsinst, hp->h_name);
        if (ptr) {
                (void) strcat(fsinst, ":");
                ptr++;
                (void) strcat(fsinst, ptr);
        }
        return(fsinst);
}

/* convert tabs in the buffer into appropriate # of spaces.
   slightly tricky since the buffer can have NUL's in it. */

#ifndef TABSTOP
#define TABSTOP 8                       /* #chars between tabstops */
#endif /* ! TABSTOP */

void
un_tabify(bufp, sizep)
char **bufp;
register int *sizep;
{
    register char *cp, *cp2;
    char *cp3;
    register int i;
```

```
    register int column;                /* column of next character */
    register int size = *sizep;

    for (cp = *bufp, i = 0; size; size--, cp++)
        if (*cp == '\t')
            i++;                        /* count tabs in buffer */

    if (!i)
        return;                         /* no tabs == no work */

    /* To avoid allocation churning, allocate enough extra space to convert
       every tab into TABSTOP spaces */
    /* only add (TABSTOP-1)x because we re-use the cell holding the
       tab itself */
    cp = malloc((unsigned)(*sizep + (i * (TABSTOP-1))));
    if (!cp)                            /* XXX */
        return;                         /* punt expanding if memory fails */
    cp3 = cp;
    /* Copy buffer, converting tabs to spaces as we go */
    for (cp2 = *bufp, column = 1, size = *sizep; size; cp2++, size--) {
        switch (*cp2) {
        case '\n':
        case '\0':
            /* newline or null: reset column */
            column = 1;
            *cp++ = *cp2;               /* copy the newline */
            break;
        default:
            /* copy the character */
            *cp = *cp2;
            cp++;
            column++;
            break;
        case '\t':
            /* it's a tab, compute how many spaces to expand into. */
            i = TABSTOP - ((column - 1) % TABSTOP);
            for (; i > 0; i--) {
                *cp++ = ' ';            /* fill in the spaces */
                column++;
                (*sizep)++;             /* increment the size */
            }
            (*sizep)--;                 /* remove one (we replaced the tab) */
            break;
        }
    }
    free(*bufp);                        /* free the old buf */
    *bufp = cp3;
    return;
}
```

A.2. zlocate

```
/* This file is part of the Project Athena Zephyr Notification System.
 * It contains code for the "zlocate" command.
 *
 *      Created by:     Robert French
 *
 *      Source: /mit/zephyr/src/clients/zlocate/RCS/zlocate.c,v
 *      Author: jtkohl
 *
 *      Copyright (c) 1987,1988 by the Massachusetts Institute of Technology.
 *      For copying and distribution information, see the file
 *      "mit-copyright.h".
 */

#include <zephyr/mit-copyright.h>

#include <zephyr/zephyr.h>
#include <string.h>

#ifndef lint
static char rcsid_zlocate_c[] =
 "Header: zlocate.c,v 1.8 88/08/01 14:12:14 jtkohl Exp ";
#endif lint

main(argc,argv)
        int argc;
        char *argv[];
{
        int retval,numlocs,i,one,ourargc,found;
        char *whoami,bfr[BUFSIZ],user[BUFSIZ];
        ZLocations_t locations;

        whoami = argv[0];

        if (argc < 2) {
                printf("Usage: %s user ... \n",whoami);
                exit(1);
        }

        if ((retval = ZInitialize()) != ZERR_NONE) {
                com_err(whoami,retval,"while initializing");
                exit(1);
        }

        argv++;
        argc--;

        one = 1;
        found = 0;
        ourargc = argc;

        for (;argc--;argv++) {
                (void) strcpy(user,*argv);
                if (!index(user,'@')) {
                        (void) strcat(user,"@");
                        (void) strcat(user,ZGetRealm());
                }
```

```
		if ((retval = ZLocateUser(user,&numlocs)) != ZERR_NONE) {
			(void) sprintf(bfr,"while locating user %s",user);
			com_err(whoami,retval,bfr);
			continue;
		}
		if (ourargc > 1)
			printf("\t%s:\n",user);
		if (!numlocs) {
			printf("Hidden or not logged-in\n");
			if (argc)
				printf("\n");
			continue;
		}
		for (i=0;i<numlocs;i++) {
			if ((retval = ZGetLocations(&locations,&one))
			    != ZERR_NONE) {
				com_err(whoami,retval,
					"while getting location");
				continue;
			}
			if (one != 1) {
				printf("\
%s: internal failure while getting location\n",whoami);
				exit(1);
			}
			/* just use printf; make the field widths one
			 * smaller to deal with the extra separation space.
			 */
			printf("%-*s %-*s %s\n",
			       42, locations.host,
			       7, locations.tty,
			       locations.time);
			found++;
		}
		if (argc)
			printf("\n");
		(void) ZFlushLocations();
	}
	if (!found)
	    exit(1);
	exit(0);
}
```

A.3. zstat

```
/* This file is part of the Project Athena Zephyr Notification System.
 * It contains the zstat program.
 *
 *      Created by:     David C. Jedlinsky
 *
 *      Source: /mit/zephyr/src/clients/zstat/RCS/zstat.c,v
 *      Author: jtkohl
 *
 *      Copyright (c) 1987,1988 by the Massachusetts Institute of Technology.
 *      For copying and distribution information, see the file
 *      "mit-copyright.h".
 */

#include <zephyr/zephyr.h>
#include "../server/zserver.h"
#include <sys/param.h>
#include <sys/socket.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>

#ifndef lint
#ifndef SABER
static char rcsid_zstat_c[] =
 "Header: zstat.c,v 1.6 88/06/28 10:42:54 jtkohl Exp ";
#endif SABER
#endif lint

extern long atol();

char *hm_head[] = { "Current server =",
                    "Items in queue:",
                    "Client packets received:",
                    "Server packets received:",
                    "Server changes:",
                    "Version:",
                    "Looking for a new server:",
                    "Time running:",
                    "Size:",
                    "Machine type:"
};
#define HM_SIZE (sizeof(hm_head) / sizeof (char *))
char *srv_head[] = {
        "Current server version =",
        "Packets handled:",
        "Uptime:",
        "Server states:",
};
#define SRV_SIZE        (sizeof(srv_head) / sizeof (char *))

int serveronly = 0,hmonly = 0;
int outoftime = 0;
u_short hm_port,srv_port;

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

```
	char *argv[];
{
	Code_t ret;
	char hostname[MAXHOSTNAMELEN];
	int optchar;
	struct servent *sp;
	extern char *optarg;
	extern int optind;

	if ((ret = ZInitialize()) != ZERR_NONE) {
		com_err("zstat", ret, "initializing");
		exit(-1);
	}

	if ((ret = ZOpenPort((u_short *)0)) != ZERR_NONE) {
		com_err("zstat", ret, "opening port");
		exit(-1);
	}

	while ((optchar = getopt(argc, argv, "sh")) != EOF) {
		switch(optchar) {
		case 's':
			serveronly++;
			break;
		case 'h':
			hmonly++;
			break;
		case '?':
		default:
			usage(argv[0]);
			exit(1);
		}
	}

	if (serveronly && hmonly) {
		fprintf(stderr,"Only one of -s and -h may be specified\n");
		exit(1);
	}

	if (!(sp = getservbyname("zephyr-hm","udp"))) {
		fprintf(stderr,"zephyr-hm/udp: unknown service\n");
		exit(-1);
	}

	hm_port = sp->s_port;

	if (!(sp = getservbyname("zephyr-clt","udp"))) {
		fprintf(stderr,"zephyr-clt/udp: unknown service\n");
		exit(-1);
	}

	srv_port = sp->s_port;

	if (optind == argc) {
		if (gethostname(hostname, MAXHOSTNAMELEN) < 0) {
			com_err("zstat",errno,"while finding hostname");
			exit(-1);
		}
		do_stat(hostname);
```

```
                exit(0);
        }

        for (;optind<argc;optind++)
                do_stat(argv[optind]);

        exit(0);
}

do_stat(host)
        char *host;
{
        char srv_host[MAXHOSTNAMELEN];

        if (serveronly) {
                (void) srv_stat(host);
                return;
        }

        if (hm_stat(host,srv_host))
                return;

        if (!hmonly)
                (void) srv_stat(srv_host);
}

hm_stat(host,server)
        char *host,*server;
{
        char *line[20],*mp;
        int sock,i,nf,ret;
        struct hostent *hp;
        struct sockaddr_in sin;
        long runtime;
        struct tm *tim;
        ZNotice_t notice;
        extern int timeout();

        bzero((char *)&sin,sizeof(struct sockaddr_in));

        sin.sin_port = hm_port;

        if ((sock = socket(PF_INET, SOCK_DGRAM, 0)) < 0) {
                perror("socket:");
                exit(-1);
        }

        sin.sin_family = AF_INET;

        if ((hp = gethostbyname(host)) == NULL) {
                fprintf(stderr,"Unknown host: %s\n",host);
                exit(-1);
        }
        bcopy(hp->h_addr, (char *) &sin.sin_addr, hp->h_length);

        printf("Hostmanager stats: %s\n",hp->h_name);

        (void) bzero((char *)&notice, sizeof(notice));
        notice.z_kind = STAT;
```

```
notice.z_port = 0;
notice.z_class = HM_STAT_CLASS;
notice.z_class_inst = HM_STAT_CLIENT;
notice.z_opcode = HM_GIMMESTATS;
notice.z_sender = "";
notice.z_recipient = "";
notice.z_default_format = "";
notice.z_message_len = 0;

if ((ret = ZSetDestAddr(&sin)) != ZERR_NONE) {
        com_err("zstat", ret, "setting destination");
        exit(-1);
}
if ((ret = ZSendNotice(&notice, ZNOAUTH)) != ZERR_NONE) {
        com_err("zstat", ret, "sending notice");
        exit(-1);
}

(void) signal(SIGALRM,timeout);
outoftime = 0;
(void) alarm(10);
if (((ret = ZReceiveNotice(&notice, (struct sockaddr_in *) 0))
     != ZERR_NONE) &&
    ret != EINTR) {
        com_err("zstat", ret, "receiving notice");
        return (1);
}
(void) alarm(0);
if (outoftime) {
        fprintf(stderr,"No response after 10 seconds.\n");
        return (1);
}

mp = notice.z_message;
for (nf=0;mp<notice.z_message+notice.z_message_len;nf++) {
        line[nf] = mp;
        mp += strlen(mp)+1;
}

(void) strcpy(server,line[0]);

printf("HostManager protocol version = %s\n",notice.z_version);

for (i=0; (i < nf) && (i < HM_SIZE); i++) {
        if (!strncmp("Time",hm_head[i],4)) {
                runtime = atol(line[i]);
                tim = gmtime(&runtime);
                printf("%s %d days, %02d:%02d:%02d\n", hm_head[i],
                        tim->tm_yday,
                        tim->tm_hour,
                        tim->tm_min,
                        tim->tm_sec);
        }
        else
                printf("%s %s\n",hm_head[i],line[i]);
}

printf("\n");
```

```
        (void) close(sock);
        ZFreeNotice(&notice);
        return(0);
}

srv_stat(host)
        char *host;
{
        char *line[20],*mp;
        int sock,i,nf,ret;
        struct hostent *hp;
        struct sockaddr_in sin;
        ZNotice_t notice;
        long runtime;
        struct tm *tim;
        extern int timeout();

        bzero((char *) &sin,sizeof(struct sockaddr_in));

        sin.sin_port = srv_port;

        if ((sock = socket(PF_INET, SOCK_DGRAM, 0)) < 0) {
                perror("socket:");
                exit(-1);
        }

        sin.sin_family = AF_INET;

        if ((hp = gethostbyname(host)) == NULL) {
                fprintf(stderr,"Unknown host: %s\n",host);
                exit(-1);
        }
        bcopy(hp->h_addr, (char *) &sin.sin_addr, hp->h_length);

        printf("Server stats: %s\n",hp->h_name);

        (void) bzero((char *)&notice, sizeof(notice));
        notice.z_kind = UNSAFE;
        notice.z_port = 0;
        notice.z_class = ZEPHYR_ADMIN_CLASS;
        notice.z_class_inst = "";
        notice.z_opcode = ADMIN_STATUS;
        notice.z_sender = "";
        notice.z_recipient = "";
        notice.z_default_format = "";
        notice.z_message_len = 0;

        if ((ret = ZSetDestAddr(&sin)) != ZERR_NONE) {
                com_err("zstat", ret, "setting destination");
                exit(-1);
        }
        if ((ret = ZSendNotice(&notice, ZNOAUTH)) != ZERR_NONE) {
                com_err("zstat", ret, "sending notice");
                exit(-1);
        }

        (void) signal(SIGALRM,timeout);
        outoftime = 0;
        (void) alarm(10);
```

```
        if (((ret = ZReceiveNotice(&notice, (struct sockaddr_in *) 0))
            != ZERR_NONE) &&
            ret != EINTR) {
                com_err("zstat", ret, "receiving notice");
                return (1);
        }
        (void) alarm(0);
        if (outoftime) {
                fprintf(stderr,"No response after 10 seconds.\n");
                return (1);
        }

        mp = notice.z_message;
        for (nf=0;mp<notice.z_message+notice.z_message_len;nf++) {
                line[nf] = mp;
                mp += strlen(mp)+1;
        }

        printf("Server protocol version = %s\n",notice.z_version);

        for (i=0; i < nf; i++) {
                if (i < 2)
                        printf("%s %s\n",srv_head[i],line[i]);
                else if (i == 2) { /* uptime field */
                        runtime = atol(line[i]);
                        tim = gmtime(&runtime);
                        printf("%s %d days, %02d:%02d:%02d\n",
                               srv_head[i],
                               tim->tm_yday,
                               tim->tm_hour,
                               tim->tm_min,
                               tim->tm_sec);
                } else if (i == 3) {
                        printf("%s\n",srv_head[i]);
                        printf("%s\n",line[i]);
                } else printf("%s\n",line[i]);
        }
        printf("\n");

        (void) close(sock);
        return(0);
}

usage(s)
        char *s;
{
        fprintf(stderr,"usage: %s [-s] [-h] [host ...]\n",s);
        exit(1);
}

timeout()
{
        outoftime = 1;
}
```

B. Error Codes

The following error codes are defined in zephyr_err.h, which is included by zephyr.h:

ZERR_NONE	No error was detected.
ZERR_PKTLEN	A notice was too big to fit into the supplied buffer, or ran over the maximum allowed packet size.
ZERR_HEADERLEN	A formatted notice header exceeded the maximum header length.
ZERR_ILLVAL	A member of the notice structure contained an illegal value.
ZERR_HMPORT	The port number for the HostManager could not be found in */etc/services*.
ZERR_PORTINUSE	The port explicitly requested in a call to ZOpenPort couldn't be bound.
ZERR_BADPKT	The packet is badly formatted and can't be parsed.
ZERR_VERS	The packet was formatted using a version of the protocol which is incompatible with the protocol supported by this version of the library function.
ZERR_NOPORT	A function which requires an open port was called before a port was opened.
ZERR_NONOTICE	No notice which was accepted by the predicate was found by ZCheckIfNotice.
ZERR_QLEN	Too many notices have been read in to the input queue but not retrieved.
ZERR_HMDEAD	The HostManager did not send an acknowledgment to a packet.
ZERR_INTERNAL	Something really strange is going on, probably with the system instead of with the application itself (for example, */etc/passwd* can't be read).
ZERR_NOLOCATIONS	No locations were available to return using ZGetLocations or flush using ZFlushLocations. This usually means that ZLocateUser hasn't been called yet.
ZERR_NOMORELOCS	No more locations were available to return. This means that ZGetLocations wants to retrieve more locations than are available.
ZERR_FIELDLEN	The data passed to ZMakeAscii exceeds the size of the supplied buffer.
ZERR_BADFIELD	The data passed to ZReadAscii is in an improper format.
ZERR_SERVNAK	A server negative-acknowledgment was received while performing a privileged operation (such as subscribing). This usually means that user authentication failed.
ZERR_AUTHFAIL	A login notice was not authenticated properly.
ZERR_LOGINFAIL	A login notice was not accepted by the server.

ZERR_NOSUBSCRIPTIONS	No subscriptions were available to return using ZGetSubscriptions or flush using ZFlushSubscriptions. This usually means that ZRetrieveSubscriptions or ZRetrieveDefaultSubscriptions hasn't been called yet.
ZERR_NOMORESUBSCRIPTIONS	No more subscriptions were available to return. This means that ZGetSubscriptions wants to retrieve more subscriptions than are available.
ZERR_EOF	The file descriptor that the library is using to read its packets was inadvertently closed.

C. Function Templates

Function template for ZInitialize:

```
Code_t ZInitialize ()
```

Prerequisite functions: None

Possible errors: ZERR_HMPORT

Function template for ZOpenPort:

```
Code_t ZOpenPort (port)
        unsigned short    *port;
```

Prerequisite functions: ZInitialize

Possible errors: UNIX errors, ZERR_PORTINUSE

Function template for ZClosePort:

```
Code_t ZClosePort ()
```

Prerequisite functions: None

Possible errors: None

Function template for ZSendNotice:

```
Code_t ZSendNotice (notice, cert_routine)
        ZNotice_t    *notice;
        int          (*cert_routine)();
```

Prerequisite functions: ZInitialize

Possible errors: Kerberos errors, UNIX errors, ZERR_HEADERLEN, ZERR_ILLVAL, ZERR_HMDEAD, ZERR_BADPKT, ZERR_VERS, ZERR_QLEN

Function template for ZSendList:

```
Code_t ZSendList (notice, list, nitems, cert_routine)
        ZNotice_t    *notice;
        char         *list[];
        int          nitems;
        int          (*cert_routine)();
```

Prerequisite functions: ZInitialize

Possible errors: Kerberos errors, UNIX errors, ZERR_HEADERLEN, ZERR_ILLVAL, ZERR_HMDEAD, ZERR_BADPKT, ZERR_VERS, ZERR_QLEN

Function template for ZGetSender:

```
char * ZGetSender ()
```

Prerequisite functions: ZInitialize

Possible errors: None

Function template for ZGetRealm:

```
char * ZGetRealm ()
```

Prerequisite functions: ZInitialize

Possible errors: None

Function template for ZMakeAscii:

```
Code_t ZMakeAscii (buffer, buffer_len, field, field_len)
        char            *buffer;
        int             buffer_len;
        unsigned char   *field;
        int             field_len;
```

Prerequisite functions: None

Possible errors: ZERR_FIELDLEN

Function template for ZCheckIfNotice:

```
Code_t ZCheckIfNotice (notice, from, predicate, args)
        ZNotice_t            *notice;
        struct sockaddr_in   *from;
        int                  (*predicate)();
        char                 *args;
```

Prerequisite functions: ZInitialize, ZOpenPort

Possible errors: UNIX errors, ZERR_BADPKT, ZERR_VERS, ZERR_PKTLEN, ZERR_NOPORT, ZERR_QLEN, ZERR_NONOTICE

Function template for ZFreeNotice:

```
void ZFreeNotice (notice)
        ZNotice_t    *notice;
```

Prerequisite functions: None

Possible errors: None

Function template for ZIfNotice:

```
Code_t ZIfNotice (notice, from, predicate, args)
        ZNotice_t             *notice;
        struct sockaddr_in    *from;
        int                   (*predicate)();
        char                  *args;
```

Prerequisite functions: ZInitialize, ZOpenPort

Possible errors: UNIX errors, ZERR_BADPKT, ZERR_VERS, ZERR_PKTLEN, ZERR_NOPORT, ZERR_QLEN

Function template for ZCompareUID:

```
int ZCompareUID (uid1, uid2)
        ZUnique_Id_t    *uid1;
        ZUnique_Id_t    *uid2;
```

Prerequisite functions: None

Possible errors: None

Function template for ZSubscribeTo:

```
Code_t ZSubscribeTo (sublist, nitems, port)
        ZSubscription_t    sublist[];
        int                nitems;
        unsigned short     port;
```

Prerequisite functions: ZInitialize

Possible errors: Kerberos errors, UNIX errors, ZERR_PKTLEN, ZERR_ILLVAL, ZERR_HMDEAD, ZERR_BADPKT, ZERR_VERS, ZERR_QLEN

Function template for ZUnsubscribeTo:

```
Code_t ZUnsubscribeTo (sublist, nitems, port)
        ZSubscription_t    sublist[];
        int                nitems;
        unsigned short     port;
```

Prerequisite functions: ZInitialize

Possible errors: Kerberos errors, UNIX errors, ZERR_PKTLEN, ZERR_ILLVAL, ZERR_HMDEAD, ZERR_BADPKT, ZERR_VERS, ZERR_QLEN

Function template for ZCancelSubscriptions:

```
Code_t ZCancelSubscriptions (port)
        unsigned short    port;
```

Prerequisite functions: ZInitialize

Possible errors: Kerberos errors, UNIX errors, ZERR_PKTLEN, ZERR_ILLVAL, ZERR_HMDEAD, ZERR_BADPKT, ZERR_VERS, ZERR_QLEN

Function template for ZGetWGPort:

```
int ZGetWGPort ()
```

Prerequisite functions: None

Possible errors: -1 = No port number available

Function template for ZReceiveNotice:

```
Code_t ZReceiveNotice (notice, from)
        ZNotice_t             *notice;
        struct sockaddr_in    *from;
```

Prerequisite functions: ZInitialize, ZOpenPort

Possible errors: UNIX errors, ZERR_NOPORT, ZERR_PKTLEN, ZERR_QLEN, ZERR_BADPKT, ZERR_VERS

Function template for ZReadAscii:

```
Code_t ZReadAscii (buffer, buffer_len, field, field_len)
        char    *buffer;
        int     buffer_len;
        char    *field;
        int     field_len;
```

Prerequisite functions: None

Possible errors: ZERR_BADFIELD

Function template for ZCheckAuthentication:

```
Code_t ZCheckAuthentication (notice, from)
        ZNotice_t             *notice;
        struct sockaddr_in    *from;
```

Prerequisite functions: ZInitialize, ZReceiveNotice

Possible errors: None

Function template for ZPending:

```
int ZPending ()
```

Prerequisite functions: ZInitialize, ZOpenPort

Possible errors: UNIX errors, ZERR_MAXQLEN, ZERR_EOF

Function template for ZQLength:

```
int ZQLength ()
```

Prerequisite functions: ZInitialize, ZOpenPort

Possible errors: None

Function template for ZPeekNotice:

```
Code_t ZPeekNotice (notice, from)
        ZNotice_t             *notice;
        struct sockaddr_in    *from;
```

Prerequisite functions: ZInitialize, ZOpenPort

Possible errors: UNIX errors, ZERR_NOPORT, ZERR_PKTLEN, ZERR_QLEN, ZERR_BADPKT, ZERR_VERS

Function template for ZPeekIfNotice:

```
Code_t ZPeekIfNotice (notice, from, predicate, args)
        ZNotice_t             *notice;
        struct sockaddr_in    *from;
        int                   (*predicate)();
        char                  *args;
```

Prerequisite functions: ZInitialize, ZOpenPort

Possible errors: UNIX errors, ZERR_BADPKT, ZERR_VERS, ZERR_PKTLEN, ZERR_NOPORT, ZERR_QLEN

Function template for ZFormatNotice:

```
Code_t ZFormatNotice (notice, buffer, ret_len, cert_routine)
        ZNotice_t    *notice;
        char         **buffer;
        int          *ret_len;
        int          (*cert_routine)();
```

Prerequisite functions: ZInitialize

Possible errors: Kerberos errors, UNIX errors, ZERR_PKTLEN, ZERR_ILLVAL

Function template for ZFormatNoticeList:

```
Code_t ZFormatNoticeList (notice, list, nitems, buffer, ret_len, cert_routine)
        ZNotice_t    *notice;
        char         *list[];
        int          nitems;
        char         **buffer;
        int          *ret_len;
        int          (*cert_routine)();
```

Prerequisite functions: ZInitialize

Possible errors: Kerberos errors, UNIX errors, ZERR_PKTLEN, ZERR_ILLVAL

Function template for ZSendPacket:

```
Code_t ZSendPacket (packet, len, waitforack)
        char    *packet;
        int     len;
        int     waitforack;
```

Prerequisite functions: ZInitialize

Possible errors: UNIX errors, ZERR_ILLVAL, ZERR_HMDEAD, ZERR_BADPKT, ZERR_VERS, ZERR_PKTLEN, ZERR_QLEN

Function template for ZReceivePacket:

```
Code_t ZReceivePacket (buffer, ret_len, from)
        ZPacket_t              buffer;
        int                    *ret_len;
        struct sockaddr_in     *from;
```

Prerequisite functions: ZInitialize, ZOpenPort

Possible errors: UNIX errors, ZERR_NOPORT, ZERR_PKTLEN, ZERR_QLEN

Function template for ZPeekPacket:

```
Code_t ZPeekPacket (buffer, ret_len, from)
        char                  **buffer;
        int                   *ret_len;
        struct sockaddr_in    *from;
```

Prerequisite functions: ZInitialize, ZOpenPort

Possible errors: UNIX errors, ZERR_NOPORT, ZERR_PKTLEN, ZERR_QLEN, ZERR_BADPKT, ZERR_VERS

Function template for ZParseNotice:

```
Code_t ZParseNotice (buffer, buffer_len, notice)
        char        *buffer;
        int         buffer_len;
        ZNotice_t   *notice;
```

Prerequisite functions: ZInitialize

Possible errors: ZERR_BADPKT, ZERR_VERS

Function template for ZFormatRawNotice:

```
Code_t ZFormatRawNotice (notice, buffer, ret_len)
        ZNotice_t   *notice;
        char        **buffer;
        int         *ret_len;
```

Prerequisite functions: ZInitialize

Possible errors: ZERR_PKTLEN, ZERR_HEADERLEN

Function template for ZFormatSmallRawNotice:

```
Code_t ZFormatSmallRawNotice (notice, buffer, ret_len)
        ZNotice_t   *notice;
        ZPacket_t   buffer;
        int         *ret_len;
```

Prerequisite functions: ZInitialize

Possible errors: ZERR_PKTLEN, ZERR_HEADERLEN

Function template for ZFormatRawNoticeList:

```
Code_t ZFormatRawNoticeList (notice, list, nitems, buffer, ret_len)
        ZNotice_t   *notice;
        char        *list[];
        int         nitems;
        char        **buffer;
        int         *ret_len;
```

Prerequisite functions: ZInitialize

Possible errors: ZERR_PKTLEN, ZERR_HEADERLEN

Function template for ZFormatSmallRawNoticeList:

```
Code_t ZFormatSmallRawNoticeList (notice, list, nitems, buffer, ret_len)
        ZNotice_t   *notice;
        char        *list[];
        int         nitems;
        ZPacket_t   buffer;
        int         *ret_len;
```

Prerequisite functions: ZInitialize

Possible errors: ZERR_PKTLEN, ZERR_HEADERLEN

Function template for ZSendRawNotice:

```
Code_t ZSendRawNotice (notice)
        ZNotice_t   *notice;
```

Prerequisite functions: ZInitialize

Possible errors: UNIX errors, ZERR_PKTLEN, ZERR_HMDEAD, ZERR_BADPKT, ZERR_VERS, ZERR_QLEN

Function template for ZSendRawList:

```
Code_t ZSendRawList (notice, list, nitems)
        ZNotice_t   *notice;
        char        *list[];
        int         nitems;
```

Prerequisite functions: ZInitialize

Possible errors: UNIX errors, ZERR_PKTLEN, ZERR_HMDEAD, ZERR_BADPKT, ZERR_VERS, ZERR_QLEN

Function template for ZLocateUser:

```
Code_t ZLocateUser (user, nlocs)
        char    *user;
        int     *nlocs;
```

Prerequisite functions: ZInitialize

Possible errors: Kerberos errors, UNIX errors, ZERR_PKTLEN, ZERR_ILLVAL, ZERR_HMDEAD, ZERR_BADPKT, ZERR_VERS, ZERR_QLEN

Function template for ZGetLocations:

```
Code_t ZGetLocations (location, numloc)
        ZLocations_t    location[];
        int             *numloc;
```

Prerequisite functions: ZInitialize, ZLocateUser

Possible errors: ZERR_NOLOCATIONS, ZERR_NOMORELOCS

Function template for ZFlushLocations:

```
Code_t ZFlushLocations ()
```

Prerequisite functions: ZInitialize, ZLocateUser

Possible errors: None

Function template for ZRetrieveSubscriptions:

```
Code_t ZRetrieveSubscriptions (port, nsubs)
        unsigned short  port;
        int             *nsubs;
```

Prerequisite functions: ZInitialize

Possible errors: Kerberos errors, UNIX errors, ZERR_PKTLEN, ZERR_ILLVAL, ZERR_HMDEAD, ZERR_BADPKT, ZERR_VERS, ZERR_QLEN

Function template for ZRetrieveDefaultSubscriptions:

```
Code_t ZRetrieveDefaultSubscriptions (nsubs)
        int     *nsubs;
```

Prerequisite functions: ZInitialize

Possible errors: Kerberos errors, UNIX errors, ZERR_PKTLEN, ZERR_ILLVAL, ZERR_HMDEAD, ZERR_BADPKT, ZERR_VERS, ZERR_QLEN

Function template for ZGetSubscriptions:

```
Code_t ZGetSubscriptions (subscription, numsub)
        ZSubscription_t     *subscription;
        int                 *numsub;
```

Prerequisite functions: ZInitialize, ZRetrieveSubscriptions or ZRetrieveDefaultSubscriptions

Possible errors: ZERR_NOSUBSCRIPTIONS, ZERR_NOMORESUBSCRIPTIONS

Function template for ZFlushSubscriptions:

```
Code_t ZFlushSubscriptions ()
```

Prerequisite functions: ZInitialize, ZRetrieveSubscriptions or ZRetrieveDefaultSubscriptions

Possible errors: None

Function template for ZGetVariable:

```
char * ZGetVariable (var)
        char    *var;
```

Prerequisite functions: None

Possible errors: NULL = variable not defined

Function template for ZSetVariable:

```
Code_t ZSetVariable (var, value)
        char    *var;
        char    *value;
```

Prerequisite functions: None

Possible errors: UNIX errors, ZERR_INTERNAL

Function template for ZUnsetVariable:

```
Code_t ZUnsetVariable (var)
        char    *var;
```

Prerequisite functions: None

Possible errors: UNIX errors, ZERR_INTERNAL

Function template for ZSetLocation:

```
Code_t ZSetLocation (exposure)
        char    *exposure;
```

Prerequisite functions: ZInitialize

Possible errors: Kerberos errors, UNIX errors, ZERR_PKTLEN, ZERR_ILLVAL, ZERR_HMDEAD, ZERR_BADPKT, ZERR_VERS, ZERR_QLEN, ZERR_SERVNAK, ZERR_AUTHFAIL, ZERR_LOGINFAIL

Function template for ZUnsetLocation:

```
Code_t ZUnsetLocation ()
```

Prerequisite functions: ZInitialize

Possible errors: Kerberos errors, UNIX errors, ZERR_PKTLEN, ZERR_ILLVAL, ZERR_HMDEAD, ZERR_BADPKT, ZERR_VERS, ZERR_QLEN, ZERR_SERVNAK, ZERR_AUTHFAIL, ZERR_LOGINFAIL

Function template for ZFlushMyLocations:

```
Code_t ZFlushMyLocations ()
```

Prerequisite functions: ZInitialize

Possible errors: Kerberos errors, UNIX errors, ZERR_PKTLEN, ZERR_ILLVAL, ZERR_HMDEAD, ZERR_BADPKT, ZERR_VERS, ZERR_QLEN, ZERR_SERVNAK, ZERR_AUTHFAIL, ZERR_LOGINFAIL

Function template for ZGetFD:

```
int ZGetFD ()
```

Prerequisite functions: ZInitialize or ZSetFD

Possible errors: -1 = No current file descriptor

Function template for ZSetFD:

```
Code_t ZSetFD (fd)
        int     fd;
```

Prerequisite functions: None

Possible errors: None

Function template for ZGetDestAddr:

```
struct sockaddr_in ZGetDestAddr ()
```

Prerequisite functions: ZInitialize

Possible errors: None

Function template for ZSetDestAddr:

```
Code_t ZSetDestAddr (addr)
        struct   sockaddr_in *addr;
```

Prerequisite functions: ZInitialize

Possible errors: None

Function template for ZSetServerState:

```
Code_t ZSetServerState (state)
        int    state;
```

Prerequisite functions: ZInitialize

Possible errors: None

Function template for ZSrvSendNotice:

```
Code_t ZSrvSendNotice (notice, cert_routine, send_routine)
        ZNotice_t   *notice;
        int         (*cert_routine)();
        int         (*send_routine)();
```

Prerequisite functions: ZInitialize

Possible errors: Kerberos errors, UNIX errors, ZERR_HEADERLEN, ZERR_ILLVAL, ZERR_HMDEAD, ZERR_BADPKT, ZERR_VERS, ZERR_QLEN

Function template for ZSrvSendList:

```
Code_t ZSrvSendList (notice, list, nitems, cert_routine, send_routine)
```

```
ZNotice_t   *notice;
char        *list[];
int         nitems;
int         (*cert_routine)();
int         (*send_routine)();
```

Prerequisite functions: ZInitialize

Possible errors: Kerberos errors, UNIX errors, ZERR_HEADERLEN, ZERR_ILLVAL, ZERR_HMDEAD, ZERR_BADPKT, ZERR_VERS, ZERR_QLEN

Function template for ZSrvSendRawList:

```
Code_t ZSrvSendRawList (notice, list, nitems, send_routine)
    ZNotice_t   *notice;
    char        *list[];
    int         nitems;
    int         (*send_routine)();
```

Prerequisite functions: ZInitialize

Possible errors: UNIX errors, ZERR_PKTLEN, ZERR_HMDEAD, ZERR_BADPKT, ZERR_VERS, ZERR_QLEN

Function template for ZFormatAuthenticNotice:

```
Code_t ZFormatAuthenticNotice (notice, buffer, buffer_len, ret_len, session)
    ZNotice_t   *notice;
    char        *buffer;
    int         buffer_len;
    int         *ret_len;
    C_Block     session;
```

Prerequisite functions: ZInitialize

Possible errors: Kerberos errors, UNIX errors, ZERR_PKTLEN, ZERR_HEADERLEN, ZERR_ILLVAL

References

[1] DellaFera, C. Anthony, *et al. Zephyr Notification Service.* MIT Project Athena Technical Plan, section E.4.1.

[2] Miller, Steven P. and Clifford Neuman. *Kerberos.* MIT Project Athena Technical Plan, section E.2.1.

[3] Raeburn, Kenneth. *A Common Error Description Library for UNIX.*

[4] Sechres, Stuart. *An Introductory 4.3BSD Interprocess Communication Tutorial.* UNIX Programmer's Supplementary Documents, Volume 1. 4.3 Berkeley Software Distribution.