SetCtrl | 14 — Requeue Reject Queue

Requeues the queue of reject (unacknowledged) outbound LAN messages. However, it is not the equivalent of the application reading the rejects back one at a time (with the `ioctlc()` command described in the following) and rewriting them. This command will attempt only a single transmission of each rejected message. If the transmission fails, the message is purged without retries or requeuing it as a reject. Furthermore, no `PACKET_MADEIT` or `XMIT_FAILURE` traps are generated and the buffer pointer in the call is ignored.

Although the following scenario is not likely to occur in a normal LAN operation, some aspects of it should be noted. Suppose an application reinitializes its LAN port via `put_lan_config()`. Suppose also that this terminal was involved in a rejected transmission, either as the source (sender) or destination before the reinitialization. In this case, requeuing particular rejects may not work as expected. This is because each terminal keeps a sequence counter for all other terminals on the LAN to ensure protocol integrity. Reinitialization of the LAN port resets the sequence counters maintained by both the reinitialized terminal and all others on the LAN for that terminal, which may put all pending rejects for the involved terminal(s) out of sequence (having had sequence numbers determined before the reinitialization) and prevent their normal receipt and acknowledgment. By contrast, reading back the reject messagess one at a time and attempting to retransmit them will avoid this scenario since new sequence numbers would be furnished by firmware.

SetCtrl | 15 — Empty Reject Queue

Purges the reject queue and discards any messages in it. The buffer pointer in the call is ignored for this variant.

SetCtrl | 29 — Set LAN Exception/Event Trap Mask

Sets the exception/event trap mask (Trap 52), thus enabling the trap to detect one or more events which may occur on a LAN, specifically `XMIT_FAILURE`, `PACKET_MADEIT`,

BAD_LINE, STREAMING_ERR, DUPLICATE_ADDRESS or NODE_CHANGED. The buffer pointer in the ioctl() call is ignored for this variant.

When Trap 52 is set with arm_guard() and Set Ctrl | 29, you may specify one or more of the six above-mentioned events to trap. If Trap 52 is set with only arm_guard(), only XMIT_FAILURE will be detected.

See *LAN Traps* later in this chapter.

GetCtrl | 0 — Get Current LAN Port Status Bytes

Retrieves the current LAN port status. The returned result will contain either 0, indicating there is no output pending, or 1, indicating there is output pending (one or more message(s) have been "written" from the application but have not yet gone onto the LAN). The contents of the returned buffer will be four bytes, formatted as follows:

- Byte 1: Number of slots (maximum of 16) being used for buffers containing pending input messages
- Byte 2: Number of slots (maximum of 16) being used for buffers containing messages in the reject queue
- Byte 3: Number of slots (maximum of 16) not being used for buffers in the output queue at that instant (volatile and not directly under application control)
- Byte 4: Not used

Note that each pending input message, reject message, and pending output message may occupy more than one buffer (depending on the message size), and may use more than one "slot" in the above-noted status bytes.

The "slots" are elements of FIFO queue data structures used to track pending input packets, rejected output packets, and pending output packets. There are a maximum of 16 elements in each of the three queues, drawn from the common buffer pool of between 4 and 32 buffers. Although the same buffer pool is shared by all terminal I/O functions, the LAN is likely to make the most intensive demands upon it. LAN applications should invest

considerable design time in planning a buffer management scheme.

Exercise care when monitoring the status of bytes and managing the underlying input packet and reject queues by respectively reading received input packets and either reading, requeuing, or purging reject packets. If this is not done, accumulating unread input packets and/or reject packets might quickly constrain available buffers and disrupt the proper functioning of the application and firmware.

Example Assume an application is started with `*B=18` (i.e., 18 buffers of 254 bytes each). Two messages have been received on the LAN but not yet read by the application. One was 256 bytes as transmitted across the LAN, the other 512 bytes. Two messages of 500 bytes each were sent earlier to terminals 18 and 19, which were accidentally off the LAN due to a tripped building circuit breaker, and so these messages were not ACKed and are now rejects. The application has just sent a 1000-byte message to terminal 32, and immediately thereupon issues the `ioctl() GetCtrl | 0` command to check status.

```
char *buffer;
result = ioctl (lan_handle, GetCtrl | 0, buffer);
```

If `ioctl()` was immediately issued after `write()`, then `result` will be 1, indicating that output is pending. It would be 0, however, if there was enough delay in issuing the `ioctl()` for the transmission to have gone out over the LAN.

Of the status bytes returned, buffer[0] (byte 1) would be 4, indicating four buffers contain pending input messages. (The first received message was 256 bytes; but after the CRC has been validated and removed by the firmware, the remainder header and data will fit into one 254-byte buffer. The second message occupies three buffers since 512 bytes less the CRC is 510 bytes, which will fill two 254-byte buffers with 2 bytes remaining, requiring a third buffer to accommodate the message.) To free four buffers with pending input messages and return them to the buffer pool, simply issue two successive `read()`s of the LAN.

The second status byte, buffer[1], will also be 4 since that is the number of buffers required to hold the two 500-byte

reject messages to terminals 18 and 19. These buffers may be freed and returned to the buffer pool by either reading, purging, or requeuing the reject queue.

Again, if the `result` returned was 1 then the recent transmission is still pending, and the third status byte, buffer[2], would be 12 since there would be 12 slots remaining of the 16 to support outgoing transmissions. Note that although there are 12 slots left in the transmit queue at that instant, there are only 6 free buffers (the 18 allocated by *B less the 12 buffers being used for pending received messages, reject messages, and pending transmit messages). Once the 1000-byte message goes out on the LAN, this third byte would be 16 again (all transmit queue slots available). Note, however, there would still only be 10 underlying free buffers in the buffer pool until the pending input messages and rejects are handled. The slots represented by this status byte are under the complete control of firmware. The return of the underlying occupied buffers to the buffer pool is automatic and relatively rapid.

`GetCtrl | 10` Read Exception/Event Trap Mask

Reads the exception/event trap mask to identify the events the trap has been set to detect when armed. `buffer` must point to an area of 4 bytes, the length of the exception/event trap mask. This mask defaults to detect `XMIT_FAILURE` when the trap is armed, if it is not explicitly set otherwise by the application.

`GetCtrl | 11` Read Exception/Event Trap Status

Reads the exception/event trap status to identify the events that have occurred since the last time the command was called. This status is four bytes long, and the `buffer` must point to an area of that length. Alternatively, `buffer` may simply be a pointer to a long integer. Each call resets the status.

`GetCtrl | 12` Read LAN Node List

Reads the resource definitions for all terminals on the LAN (node list) pointed to by `buffer`. The node list data is 32

bytes long. The first byte of the buffer is the resource definition of terminal address 1, the second byte is the resource definition of terminal address 2, etc. The resource definition is set by the application on each terminal when the LAN port is configured and initialized using the `put_lan_config()` function.

The LAN node list includes the resource definitions for all terminals on the LAN, including that of the terminal on which the present application is running. The list is updated as protocol messages are issued by the firmware (transparent to the application) of all active terminals on the LAN. Thus a terminal can change its resource definition and other terminals will know about it and can act on the information. A change in the resource definition of any terminal on the LAN will be detected by setting the appropriate bit of the exception event trap mask.

`GetCtrl | 13` — Read LAN ACK List

Reads the LAN ACK list into the 31-byte area of memory pointed to by `buffer`, and in `result` returns the number of ACKs received since the last `GetCtrl | 13` call. The LAN ACK list is a FIFO queue of 0 to 31 bytes where each successive byte gives the address of a terminal which ACKed a message, in the order those ACKs were received. The list is purged with each call of this command.

If there were no ACKs since the last call, the ACK list is 0 bytes long, and 0 bytes are returned to the `buffer` along with a result of 0. The application must use the returned `result` to indicate how many bytes of significant information was returned, and/or clear the buffer between calls to avoid using old data. If more than 31 ACKs were received since the command was last called, the older ones are removed from the queue. Since `result` always returns the total number of ACKs received, its value can be greater than the length of the data returned in `buffer`.

`ioctlc()`

The `ioctlc()` call reads the next message on the reject queue. The reject queue contains messages that the LAN

driver attempted to transmit, but which were undeliverable. The prototype for this function is:

```
#include <io.h>

result = ioctlc (hLAN, GetCtrl | 2, buffer, count);
int result, hLAN, count;
void *buffer;
```

The reject message is removed from the queue and transferred to the area pointed to by `buffer`. `count` specifies the maximum number of bytes to retrieve, while `result` indicates the number of bytes actually retrieved. If `count` is specified adequately large, the entire original message will be read, including all header information and data of the failed transmission (but excluding the original CRC). If `count` is less than the actual length of the reject message, only `count` bytes of the reject message will be retrieved, and the entire reject message will be deleted from the queue.

On error, `result` will contain -1 and `errno` will be set to either `EINVAL`, for an invalid command, or `EBADF` if the LAN is not open or not initialized.

close()

Closes the LAN port and releases any resources associated with it.

```
#include <lan.h>

result = close (hLAN);
int result, hLAN;
```

`close()` shuts down the LAN port, purges all pending messages and frees any memory allocated for the port. It always succeeds, returning 0 when called with a valid LAN handle.

LAN Com 3 Port Functions

An OMNI 3X5 LAN terminal's Com 3 port provides RS-232 serial support for external devices, such as synchronous (SDLC) or asynchronous modems. It operates at up to 19,200 baud in either packet or character mode and accepts Hayes-compatible operations described in *Chapter 9* under *Modem Interface*. The port does not support DTR signals, but has two clock inputs to support synchronous operation in a "slave" mode.

See *Chapter* 9. Sync/Async Serial Port (COM3) for more information.

LAN Traps

TXO's operating system uses a system of traps to signal events and exceptions. Two traps are used to signal LAN-related events, Trap 52, for LAN exceptions or events, and Trap 53, which signals that a data packet has been received.

Exception/Event, LAN Trap 52

Trap 52 signals that one or more LAN exceptions or events has occured. To determine which event(s) has occured, call `ioctl()` using `GetCtrl | 11`, see *Querying Events Status* later in this section.

Six LAN events are capable of causing Trap 52 to occur. The mnemonics for the corresponding exception bit masks are as follows:

`XMIT_FAILURE`	Attempted transmission has failed
`PACKET_MADEIT`	Packet received at destination and ACKed
`BAD_LINE`	No terminals detected on LAN

STREAMING_ERR	Oversize packet (>1520 bytes) detected
DUPLICATE_ADDRESS	Duplicate address detected
NODE_CHANGED	Resource definition in node list changed

One or more of the above symbolically-defined bits may be set in order to detect any combination of the above events. In addition, the current mask setting, and the events which have occurred may be queried, as detailed in the following sections.

☞ *If the application does not set the LAN event mask, only the transmission failure (* XMIT_FAILURE *) is detected.*

Setting the Exception/Event Mask

The following code is an example of how to set the exception/event mask to detect the XMIT_FAILURE and PACKET_MADEIT events:

```
unsigned long set_events;

/*  Set Trap 52 on transmit failure          */
/*  or packet being acknowledged by the host */
set_events = XMIT_FAILURE | PACKET_MADEIT;
result = ioctl (handle, SetCtrl | 29, &set_events);
```

Reading the Exception/Event Mask

The following code is an example of how to query the current mask settings to identify the events the trap is set to detect:

```
unsigned long get_events;

result = ioctl (handle, GetCtrl | 10, &get_events);
```

Querying Occurred Events

If the event mask is set to detect more than one event, the application will need to know which particular event(s) has occurred. Use the following `ioctl()`call to identify the events that have occured:

```
unsigned long events;

result = ioctl (handle, GetCtrl | 11, &events);
```

This call returns an integer with a 1 in the bit position(s) represented by the event(s) that have occurred since the last time this command was called.

The application may use the exception bit masks to determine what event(s) occurred. After the `ioctl()` is made, all event status bits are reset to 0.

Read ACK List

If the bit indicating an ACK was received (`PACKET_MADEIT`) was set, the following `ioctl()` may be called:

```
char *buffer;

result = ioctl (handle, GetCtrl | 13, buffer);
```

This call copies the ACK list into the area of memory pointed to by `buffer`, which must be at least 31 bytes in length. The number of terminals which have responded with an ACK since the last read of the LAN ACK list, is returned in `result`. The first byte of the list contains the terminal address of the first ACK received since last called, the second byte contains the address of the second ACK, and so on. The list is cleared after the `ioctl()` has completed.

❖ *If more than 31 terminals have ACKed a message since the last GetCtrl / 13 call, the return value will be greater than 31.*

Streaming Error

The `STREAMING_ERR` bit is set when the operating system detects a serious situation on the LAN.

The most likely cause for such an error is a terminal on the LAN continuously transmitting data, preventing other

terminals from sending their data. When this situation occurs, whether or not it is trapped, from the application's point of view, the firmware disables the LAN port hardware without actually closing the LAN. Thus `ioctl()` calls will still run successfully, and the application may still `read()` any pending (already received) input packets. However, `write()` commands will only appear to succeed. That is, they will pass the data to the firmware and return reporting success, but the data will not go out on the LAN. To clear the impasse, the application should `close()` and re-`open()` the port.

If the streaming condition still exists, the same firmware-initiated blocking occurs, and another streaming error is reported to the application. Normally, this situation must be resolved by a technician troubleshooting the LAN. The application should notify the terminal operator of this condition, via a display message such as "`CALL HELPDESK`".

Transmit Failures

The `XMIT_FAILURE` trap is activated when a data packet cannot be transmitted to its intended destination. This will happen if an ACK is not received from the destination terminal after the packet is transmitted and re-transmitted as specified by the terminal parameters `transmit_retries` and `receive_timeout` (see `get_lan_config()` and `put_lan_config()`). The rejected packet is then transferred to the reject queue.

A typical `XMIT_FAILURE` guard function will re-process rejected messages by transmitting them to another destination, or processing them locally.

☞ *Pay special attention to managing the reject queue, since it may otherwise fill with rejects, which will likely disable other functions from successfully using the buffer pool.*

Only one packet can be pending for transmission to any one address at any time. If an application attempts to transmit more that one packet to a terminal address before an ACK is received, the `write()` command fails with an `EBUSY errno` return. This situation is not detected by the transmit failure trap, since no attempt has actually been made by the firmware to transmit the second message.

Data Packet Received, LAN Trap 53

Trap 53 is activated when an incoming application data packet is processed by the protocol driver. This trap tells the application that a data packet is available and a `read()` should be performed to retrieve the packet and free up the occupied buffer(s). This read may be placed in, or activated by, a guard function.

LAN Downloading

OMNI Peer-to-Peer LAN downloading is initiated using one of three methods:

- Terminal-initiated download to self
- Server-initiated download
- Terminal-initiated download to all terminals

Each of these three methods can be performed under three different terminal conditions:

- From an empty (no application) terminal following power up
- From the System Mode by pressing the [0] key
- From the application using the `SVC_ZONTALK()` library call

Terminal-initiated Download to Self

As illustrated in Figure 10-6 on the ajoining page, the terminal initiates a download by sending a Download Request (DLR) message to the download server (see *Download Protocol and Packets* later in this section). This message can be a BROADCAST type packet, or directed to a single address. The download server responds by starting up the standard ZONTALK download protocol (i.e., by sending an ENQ). The terminal responds by sending its sign-on packet, then carries on the standard

download with the server that issued the first received ENQ, ignoring any ENQs from other servers.

The following paragraphs describe the various terminal-initiated downloads (to self) according to differing terminal conditions.

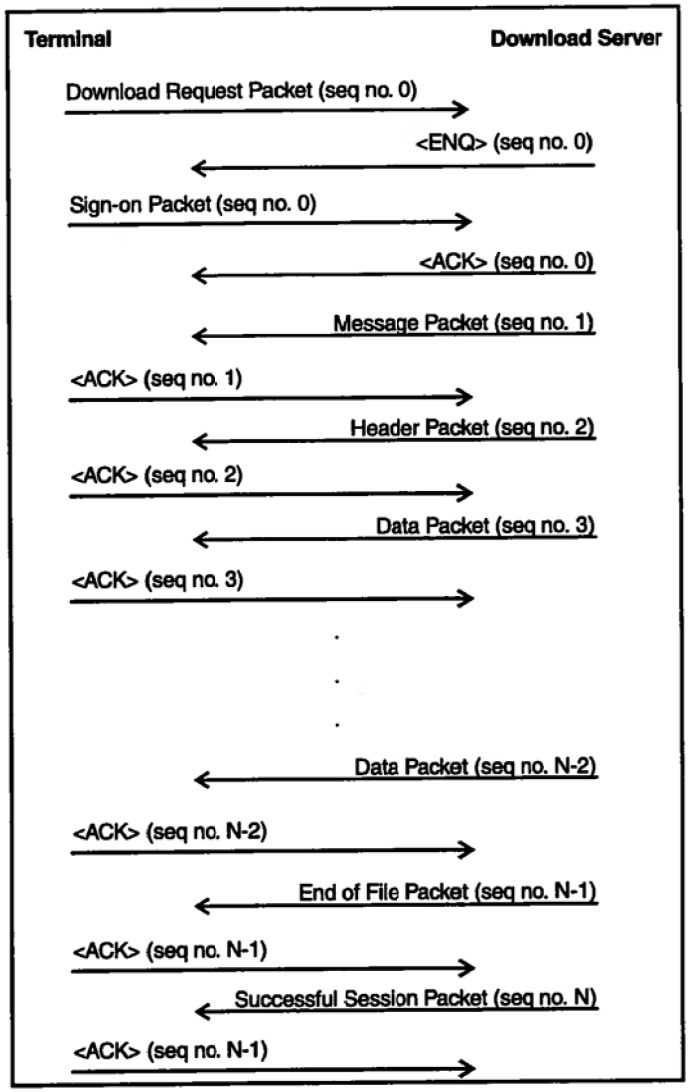

Figure 10-5: Terminal-initated Download

Empty Terminal at Power Up

On power up, if no application is present in the terminal, and the required CONFIG.SYS variables are set, the terminal displays LAN DOWNLOAD and sends out the Download Request message, waiting a specified timeout period (60 seconds) for the return ENQ. If an ENQ is received within the timeout period, the download is performed, otherwise another Download Request message is sent and the process repeats. If any packet is NAKed by the server, the terminal restarts the download request by resending the Download Request message.

If any of the required CONFIG.SYS variables are not set, or are set to invalid values, the terminal enters System Mode and prompts the user for these values. *See* CONFIG.SYS *Settings*.

The user can enter System Mode from this power up mode in the standard way: simultaneously pressing [FUNC/ENTER] and [7], and then entering the System Mode password.

If the requested application file or application serial number is not found, the download is aborted and the terminal displays DOWNLOAD FAILED. Any key press enters System Mode.

Upon completion of the download, the terminal is restarted and the downloaded application begins execution.

Terminal in System Mode

Press the [0] key in System Mode to initiate a LAN download. The operating system validates several CONFIG.SYS variables in order to successfully receive a LAN download. If the required CONFIG.SYS variables are not set, or are set to invalid values, the terminal prompts the user to enter values for them, as described under CONFIG.SYS Settings. Once all required values have been validated, the firmware displays LAN DOWNLOAD and sends out the Download Request packet, waiting for an ENQ. If it receives the ENQ within

the timeout period, the download is performed. If it times out, the request packet is resent and the terminal waits for the ENQ. This is repeated until the ENQ is received or the user interrupts by pressing the [CLEAR] key.

If any packet is NAKed by the server, the terminal restarts the download request by sending the Download Request message again.

If the requested application file or application serial number is not found, the download is aborted and the terminal displays `DOWNLOAD FAILED`. Any key press will enter System Mode.

Upon completion of the download, `DOWNLOAD DONE` is displayed. The user presses any key to return to System Mode, then presses the [CLEAR] key to start the application.

Application-initiated Download

The application may initiate a LAN download by using the following call:

```
#include <lan.h>

int  result;
char type;
result = SVC_ZONTALK(type);
```

where `type` specifies the type of download to perform:

`FULL_DL`	Full download and restart application immediately after download completes.
`PARTIAL_DL`	Partial download and restart application immediately after download completes.
`PART_DL_CONT`	Partial download and continue—after download completes, control returns to the application at the point the call was made.

REJECT_DL	Do not perform download at this time—EOT packet is sent to download server signaling that the download request was refused.

The following flags may be logically ORed to the above types:

NO_DIAL	Do not open or configure the port, do not send Download Request packet.
LAN_DOWNLOAD	Use LAN port instead of MODEM port.

See *Chapter 11, Library Functions*, SVC_ZONTALK(), *LAN.*

❖ *3X5 LAN terminals do not support an internal modem. Calling* SVC_ZONTALK() *through a port other than LAN port causes* errno *to be set to* EINVAL.

To specify a LAN download, as opposed to a modem download, the constant value LAN_DOWNLOAD is added (or logical ORed) to type.

The SVC_ZONTALK() function opens the LAN device before performing the download, unless the NO_DIAL flag is set. If the flag is not set, and the LAN has already been opened by the application, it is closed before it is re-opened. SVC_ZONTALK() uses certain variables stored in CONFIG.SYS to configure the LAN. If the required CONFIG.SYS variables are not set, or are set to invalid values, the terminal prompts the user to enter values for them, as described under CONFIG.SYS Settings.

NO_DIAL *Flag*

The NO_DIAL flag is used when the application receives a Download Request from the Server as opposed to the application requesting the download. This value signals the firmware to not send a Download Request message, but to go directly into waiting for the ENQ from the Download Server.

If the value NO_DIAL is added (or logical ORed) to type, the terminal will not open, nor will it configure the LAN. This flag implies that the application has already opened and configured the LAN properly.

If the NO_DIAL flag is used, and the port is not already open and configured, an error is returned (EBADF).

Server-Initiated Downloads

A server-initiated download provides a method for the server to download to a terminal, specifying what type of download to perform. The terminal has the option of ignoring the server request, or agreeing to proceed with the download.

The following paragraphs describe the various server-initiated downloads according to differing terminal conditions.

Empty Terminal at Power Up

As illustrated in Figure 10-7, on power up, an empty terminal sends a Download Request message and waits for an ENQ. If, instead of an ENQ, the terminal receives a Download Request message from the server, the terminal acknowledges the request (sends an ACK) and waits for the ENQ. This provides a way to insure that the terminal gets a download from the requesting server.

If the server request message designates a different terminal as the source of the download, the terminal adjusts its sign-on message accordingly before sending it in response to the ENQ. In this case, the terminal sends a new Download Request packet to the newly designated download server. See *Download Request Packet*, for more details on optional information in the request message.

Terminal in System Mode

In System Mode, the user presses [0], which causes the terminal to send a Download Request message to the server. If the terminal receives a Download Request message from the server instead of the ENQ, the terminal acknowledges the request and waits for the ENQ. This insures the download comes from the requesting server, and also allows the server to send information to the terminal about the download it will provide.

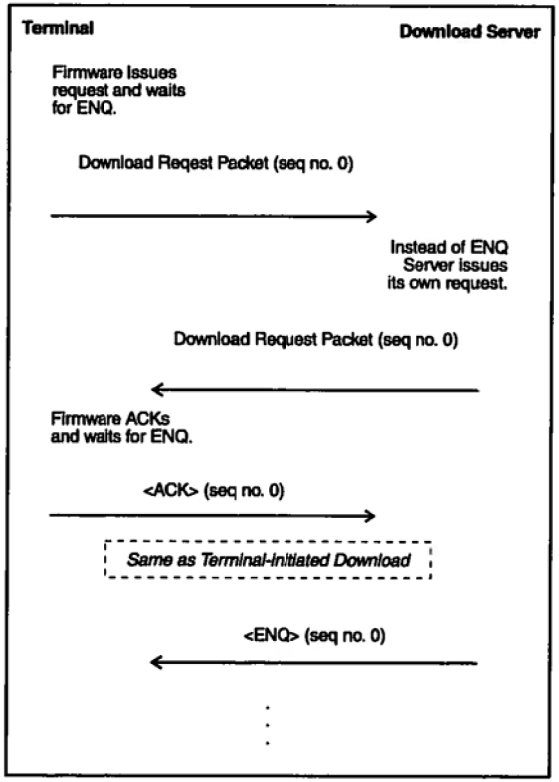

Figure 10-6: Server-intiated Download to Empty Terminal

Terminal Running Application

If, while running an application, a terminal is signaled by the download server to receive a download, the terminal has the option of accepting or rejecting the request by calling `SVC_ZONTALK()` with different parameter values.

As illustrated in Figure 10-8 on the next page, the download server sends a Download Request packet to the terminal, which triggers the input packet pending trap (Trap 53). The terminal issues a `read()` to get the contents of the Download Request packet—in particular, getting the terminal address of the download server. The application should then respond with an ACK if the packet was received okay or NAK if not.

At this point, the application can decide if it wants to proceed with the download now or wait until later. It

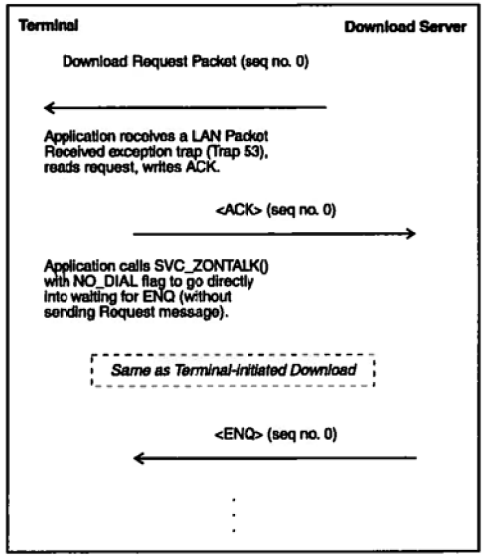

Figure 10-7: Server-initiated Download to Application

issues a SVC_ZONTALK() call with either the NO_DIAL flag set to either begin downloading, or with REJECT_DL and NO_DIAL set to reject the download request (as shown in Figure 10-9 below.

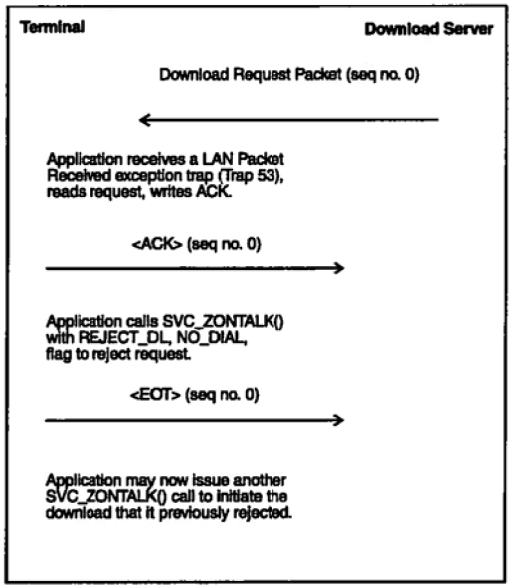

Figure 10-8: Application Rejects Download Request

❖ *Before calling* SVC_ZONTALK(), *certain LAN-related CONFIG.SYS parameters must be set. The values of these parameters may be provided as part of the Download Request packet.*

The application must disarm the LAN event trap (Trap 52) before calling SVC_ZONTALK() to prevent calling the guard function as a result of packets received during the download. The pseudo-code below shows how to build a LAN exception event guard function:

```
void lan_excep_handler()

{
   read the packet;

   if (packet is a DOWNLOAD REQUEST packet)
   {
   get source terminal addressss, app id, term id;
   set *LDS, *ZA, *ZT in CONFIG.SYS;
   if app. is ready to download
     SVC_ZONTALK(FULL_DL | LAN_DOWNLOAD | NO_DIAL);
   else
     SVC_ZONTALK(REJECT_DL | LAN_DOWNLOAD | NO_DIAL);
   }
}

main()
{
  int lan,slot;
  struct trap_mask trap;
  struct Opn_Blk lan_opn_block;

  /* open and configure the LAN */
  set lan_opn_block fields;
  lan = open("/dev/lan",0);
  result = ioctl(lan,SetCtrl|0,lan_opn_block);

  trap = mask_of(52);
  slot = arm_guard(lan_excep_handler, trap, trap);
  .
  .
  .
 }
```

Terminal Request to Download All Terminals

❖ *Note: Before a LAN terminal can request a download to all terminals on the LAN, its *LZF variable in the CONFIG.SYS file must be set to A. The *LZF variable determines what type of download request message the terminal will send whenever it initiates a download: A=Download All terminals; any other value=Download Request for single terminal.*

When a terminal, having its *LZF variable set to A, initiates a download, a Download-all message (described in the *Download Protocol and Packets* section) is sent to the download server. The download server then sends a Download Request message to every terminal on the LAN, at which point the protocol is identical to that described in the preceding *Server Initiated Downloads* section.

A Download-all request causes the requesting terminal to restart after the download completes even if the download type is `PART_DL_CONT` (Partial and Continue).

Download Protocol and Packets

Download Request (DLR) packets sent between a download server and a LAN terminal are standard OMNI LAN packets, of type DOWNLOAD (in the LAN packet header `type=0x82`). Download packets follow the VeriFone ZONTALK protocol used for RS-232 and modem downloads. These packets, *including framing and CRC characters*, are fully contained within the data section of the LAN packet. (The ZONTALK protocol framing and CRC characters are simply passed as data, and are not the same framing and CRC characters used by the LAN packet.)

Download Request Packet

The Download Request (DLR) Packet is used by both the terminal and the server to initiate a download. The

packet is of type DOWNLOAD. The format of the packet is:

<STX>DLR[,A,<type>,<app id>,<app serial #>,
<server address>,<password>,<time>,
<user-defined field>,<user-defined field>,...]
<ETX><CRC-16>

Where each item within the square brackets is optional.

Download Request Packet Optional Fields

Field	Description
A	If present, request is for a download of ALL terminals *A packet using this field is called a Download-all packet or message.* CONFIG.SYS equivalent: *LZF = A
<type>	Download type: F=Full P=partial p=partial & continue *<type> follows* SVC_ZONTALK() conventions CONFIG.SYS equivalents: *LFP = F (Full) / P (Partial)
<app id>	File name of application to download without .DLD extension (*ZA)
<app ser #>	Application serial number (*ZT)
<server address>	Address to get download from
<password>	To prevent unauthorized downloading
<time>	Time and date—when to request download
<user-defined field>, . . .	User-defined values separated by commas

These optional fields may be used in a server-initiated download scenario to specify to the terminal what kind of download to request, and from which terminal to request it. Typically, a download request from a server needs no optional fields. When optional fields are used, only those fields required for the specified download

need to be present. Initial commas separate each field; when any field is used, all preceeding empty fields must be designated by their comma separtors, as shown in the following:

Example Download Request packets:

<STX>DLR<ETX><CRC-16>
Simplest DLR)

<STX>DLR,,F<ETX><CRC-16>
DLR requesting Full download–
<type> field set to F

<STX>DLR,A,p<ETX><CRC-16>
DLR requesting Partial and Continue download
A field present,
<type> field set to p

❖ *Much of the information in the DLR packet is the same information defined for each terminal in the CONFIG.SYS file. In the event that the server sends a DLR packet with optional information that conflicts with the terminal's CONFIG.SYS settings, the packet's settings take precedence. The CONFIG.SYS settings are not changed, and DLR packet information applies only to the current download.*

In cases where the application receives the DLR packet (versus the terminal firmware—on power up or System Mode key [0]), the application can choose whether or not to use the information in the packet. If it decides to use the information within the packet, the application must set the corresponding CONFIG.SYS parameters accordingly before calling `SVC_ZONTALK()`.

When the firmware receives a DLR packet with optional fields, only the type, application id, application serial number, and server address are used; all other fields (password, time, optional fields) are ignored.

Standard ZONTALK Packets

ZONTALK packets are used in conjunction with LAN download request packets, and are included with the data portion of a LAN data packet.

❖ *ZONTALK framing and CRC characters are internal to the LAN message packet, which includes its own framing and CRC characters.*

Sign-on Packet

<STX>VFI,<term. type>,XDL,<d/l type>,<filename>,,
<eprom id>,<data size>,<ETX><CRC-16>

Example:

<STX>VFI,OMNI-385,XDL,F,PROGRAM,,VC01DV22,
128,<ETX><CRC-16>

Message Packet

<STX>M<message><ETX><CRC-16>

Example:

<STX>M****------<ETX><CRC-16>

Header Packet

<STX>O<file type><file size><file checksum, not used>
<YYYYMMDDHHMMSS><reserved, 8 bytes><file name>
<ETX><CRC-16>

Example:

<STX>OC00001B9F000000001993010514325 2
XXXXXXXXprogram.out<ETX><CRC-16>

Data Packet

<STX>W<2 byte packet length><4 byte offset>
<binary data><CRC-16>

Example:

<STX>W<0x0080><0x00000000><binary data><CRC-16>

End of File Packet

<STX>C<ETX><CRC-16>

Successful Session Packet

<STX>S<ETX><CRC-16>

<EOT> Packet

(reject download, one byte):
<EOT> = 0x04

Flow Control Packets

(one byte each):
<ENQ> = 0x05 <ACK> = 0x06 <NAK> = 0x15

Download Sequence Numbers

The firmware checks sequence numbers of all packets in the download protocol. Any packet with an invalid sequence number is ignored by the firmware, and is not passed up to the application. In all protocols, the sequence numbers of packets prior to and including the ENQ packet is zero. After the ENQ packet, the sequence numbers are incremented after every ACK is sent or received. The sequence numbers of all ACKs must match the sequence numbers of the packets they are ACKing.

During a download session, any packets with an invalid sequence number (i.e., not matching the sequence number expected by the termimal is discarded. This leads to two different scenarios based on the assumption that the server may have gotten out of sync but is still sending the packets in the proper order.

First of all, the terminal could receive a sequence number less than what it was expecting; for example, it was expecting sequence #3 and received sequence #2. This could happen if the ACK packet to the server was corrupted. The server would send the sequence #2, but the receiving terminal is expecting #3. In this case, the receiving terminal will ACK the packet and throw it away. The server would then send the proper sequence number and the download will continue normally from this point.

The other possibility is that the terminal receives a sequence number greater than expected. It ACKs it and throws it away. The server is not aware of any problem and continues sending with increasing sequence numbers. When the server is through with the download, the receiving terminal will realize that it did not get a successful download and will go into its normal "download failed" procedure.

LAN Programming Example

```
/* O3X5LAN.C                                                    */
/*   Common lan functions executed by pressing keys 1 2 3 or 4. */
/*     (1) Send message                                         */
/*     (2) Show last message received                           */
/*     (3) Check address                                        */
/*     (4) Show node list                                       */
/*                                                              */
/*   These are just simple sample executions of these function. */
/*   For real world use, it is recommened more error checking is*/
/*   done.  All returns should be checked.                      */

#include <errno.h>   /* WB ERROR header  */
#include <io.h>      /* WB IO header     */
#include <lan.h>     /* WB LAN header    */
#include <memory.h>  /* WB MEMORY header */
#include <stdio.h>   /* WB STDIO header  */
#include <string.h>  /* WB STRING header */
#include <trap.h>    /* WB TRAP header   */
#include <txostd.h>  /* WB SVC header    */
#include <txosvc.h>  /* WB SVC header    */


void lan_input_proc ();
void lan_error_proc ();
void process_selection ();
void display_lan_error();
void display_lan_msg();

int lan;                  /* LAN handle                       */
int result;               /* Results                          */
int got_lan_msg = 0;      /* Flag to signal we got a lan msg  */
int lan_bytes_read = 0;   /* Bytes read over lan              */
```

```
unsigned long lan_error_events = 0L;      /* Lan error occured */

union {
  struct {                          /* LAN messages              */
    struct packet_header header;
    unsigned char data[80];
  }lan_pkt;
  char lan_buf[100];               /* Char array for writes      */
} tx_packet, rx_packet, reject_packet;

#define ebeep() ioctl(STDERR, SetCtrl|6, 0)


/* Get CONFIG.SYS parameter */
int get_parm( parm )
char *parm;
{
  char str[8];
  int len;
  len = get_env(parm, &str[1], sizeof(str)-1);
  str[0] = len + 1;
  return (SVC_2INT(str));
}


/* Initialize lan                                              */
/*   LAN is opend and initialized.                             */
/*   Traps are set up for input packets, transmit failures,    */
/*     and LAN exception events.                               */
int init_lan ()
{
  static char iobuf[16];
```

```
  int rtn= -1, term_addr;
  int slot1;                    /* Slot number */
  unsigned long events;
  struct trap_mask mask;
  LAN_TERM_PARMS term_parms;

  /* Prompt for terminal address if needed. */
  /* Have to have valid *LAD to open lan.   */
  term_addr = (unsigned char) get_parm("*LAD");
  if ((term_addr < 1) || (term_addr > 32))
  {
    write ( STDOUT, "\fADDRESS (1-32)?", 16 );
    while ( term_addr == 0 )
    {
      (void)SVC_KEY_TXT ( iobuf, 0, 2, 1 );
      term_addr = (unsigned char) SVC_2INT ( iobuf );
      term_addr = ( (term_addr < 1) || (term_addr > 32) ) ? 0 : term_addr;
    }
    /* Set terminal lan address. */
    put_env ("*LAD", &iobuf[1], iobuf[0] - 1);
  }


  /* Open LAN.  If ok, set up port. */
  if ( (lan = open("/DEV/LAN", 0)) >= 0)
  {
    term_parms.receive_timeout     = 3;  /* 3 second timeout on ACKs */
    term_parms.resource_definition = 1;  /* not applicable */
    term_parms.transmit_retries    = 3;  /* 3 retries on transmit errs
*/
    term_parms.throttle            = THROTTLE_ON;

#ifdef 04XX
```

```
    /* 4xx Only - also need to set buffer size and number */

    rtn = SVC_INFO_TYPE(0);
    if(rtn == 2) {
      term_parms.buff_size   = 150;
      term_parms.buffer_number = 62;  /* 2 buffs per addr */
    }
    rtn = -1;
#endif

    /* If ok, set up software interrupts guard for input packets */
    /* (RX_TRAP) and exception events.                           */

    if ( (rtn = put_lan_config (lan, &term_parms)) >= 0)
    {
      mask  = mask_of(RX_TRAP);
      slot1 = arm_guard (lan_input_proc, mask, mask);
      mask = mask_of(TX_TRAP);
      slot1 = arm_guard (lan_error_proc, mask, mask);

      /* Define exception condition. */
      events = NODE_CHANGED | XMIT_FAILURE | BAD_LINE |
            STREAMING_ERR | DUPLICATE_ADDRESS;
      rtn = ioctl (lan, SetCtrl | 29, &events);
    }
  }
  return (rtn);
}


/* Wait # of seconds.                          */
/* Allow display of lan messages during wait. */
void wait(sec)
```

```
  int sec;
{
  int i,j;
  int bytes_read=0 , size=100;
  char buffer[100];

  for (i=0; i<sec; i++)
  {
    for (j=0; j<10; j++)
    {
      /* Process selection when key is pressed. */
      if (bytes_read = read (STDIN, buffer, size))
      {
    process_selection (buffer[0]);
    bytes_read = 0;
      }

      if (lan_error_events)
    display_lan_error();
      if (got_lan_msg)
    display_lan_msg();
      SVC_WAIT(100);
    }
  }
}


/* Display lan error message. */
void display_lan_error()
{
  int i;
  int bytes_read=0;
  char dint[3];
```

```
while ( lan_error_events > 0L)
{
  if ( lan_error_events & NODE_CHANGED )
     write ( STDOUT,"\fCHECK NODELIST", 15);

  else if ( lan_error_events & BAD_LINE )
     write ( STDOUT,"\fBAD LINE", 9);

  else if ( lan_error_events & STREAMING_ERR )
     write ( STDOUT,"\fSTREAMING ERROR", 16);

  else if ( lan_error_events & DUPLICATE_ADDRESS )
     write (STDOUT,"\fDUPLICATE ADDR",15 );

  else if ( lan_error_events & XMIT_FAILURE )
  {
    /* Read reject queue */
    bytes_read = ioctlc (lan, GetCtrl | 0x0002, &reject_packet,
   sizeof(reject_packet));

    ebeep();
    write ( STDOUT,"\fXMIT FAILED ", 13);
    SVC_INT2 (bytes_read, dint);
    write (STDOUT, &dint[1], dint[0]-1);
    SVC_WAIT(800);
    if (bytes_read > 0)
    {
   write (STDOUT, "\fDATA:", 6);
   write (STDOUT, reject_packet.lan_buf+18, lan_bytes_read-20);
   SVC_WAIT(2000);
   write (STDOUT, "\fHEADER:", 9);
   write (STDOUT, reject_packet.lan_buf, 18);
```

```
      SVC_WAIT(4000);
        }
      }
      ioctl (lan, GetCtrl | GET_EXCEPTION_STATUS, &lan_error_events) ;
      for (i=0; i<2; ++i)
      {
         ebeep();
         wait(1);
      }
    }
}


/* Error handler */
void lan_error_proc ()
{
  /* Set flag to signal we got a trapable event. */
  ioctl (lan, GetCtrl | GET_EXCEPTION_STATUS, &lan_error_events);
}


/* Input handler */
void lan_input_proc ()
{
  lan_bytes_read = read (lan, rx_packet.lan_buf,
sizeof(rx_packet.lan_buf));

  /* Set flag to signal we got a lan msg. */
  got_lan_msg = 1;
}


/* (1) Send message: tx_packet */
```

```
int send_msg ()
{
  int addr, bytes_written;
  static char iobuf[16];

  /* Get address to send to */
  addr = 0;
  while ( (addr<=0) || (addr>32))
  {
     write(STDOUT, "\fADDR?", 6);
     addr = getchar();
     addr = addr - 48;
     sprintf(iobuf, " %d", addr);
  }

  /* Set up message */
  tx_packet.lan_pkt.header.dst_addr = addr;
  tx_packet.lan_pkt.header.type = NORM;
  (void)strcpy ((char *)tx_packet.lan_pkt.data,"TEST MESSAGE");

  /* Send messge */
  if ( (bytes_written =
       write (lan, tx_packet.lan_buf, 32)) > 0)
  {
    write (STDOUT,"\fSENT TO", 8);
    write(STDOUT, iobuf, 2);
  }
  else
    write (STDOUT,"\fSEND FAILED", 12);  /* see errno for reason */
  wait(2);
  return (bytes_written);
}
```

```
/* (2) Show last message received. */
void display_lan_msg()
{
  got_lan_msg = 0;
  write (STDOUT, "\fHEADER:", 8);
  write (STDOUT, rx_packet.lan_buf, 18);
  wait(5);
  write (STDOUT, "\fDATA:", 6);
  write (STDOUT, rx_packet.lan_buf+18, lan_bytes_read-20);
  wait(1);
}


/* (3) Check if an address is on the LAN. */
void check_addr()
{
  unsigned char lan_list[32];
  int addr;

  /* Get address to check */
  addr = 0;
  while ( (addr<=0) || (addr>32))
  {
     write(STDOUT, "\fADDR?", 6);
     addr = getchar();
     addr = addr - 48;
  }

  ioctl (lan, GetCtrl | 12, lan_list);
  if (lan_list[addr-1] > 0 )
    write (STDOUT, "\fADDR ON LAN", 12);
  else
```

```
    write (STDOUT, "\fADDR NOT ON LAN", 16);
  wait(1);
}


/* (4) Show node list. */
void show_node_list ()
{
  int i;
  unsigned char nodelist[32];
  static char iobuf[16];

  clrscr();

  write( STDOUT, "NODES:", 6);

  /* Get node list. */
  ioctl (lan, GetCtrl | 12, nodelist);

  /* Show nodes. */
  for (i=0; (nodelist[i] > 0 )&&(i<32) ; i++)
  {
    sprintf(iobuf, "%d=%d ", i+1, nodelist[i]);
    write( STDOUT, iobuf, strlen(iobuf));
  }
  wait(1);
}


/* Process user selection. */
void process_selection (selection)
  int selection;
{
```

```
  static char iobuf[16];

  selection = selection - 48;
  sprintf(iobuf, "%d", selection);
  write(STDOUT, "\f SEL=", 6);
  write(STDOUT, iobuf, 1);
  SVC_WAIT(100);
  switch (selection)
  {
    case 1: result = send_msg(); break;
    case 2: display_lan_msg(); break;
    case 3: check_addr(); break;
    case 4: show_node_list(); break;
  }
}


main()
{
  /* Initialize lan port. */

  if (init_lan() < 0)
  {
    write(STDOUT, "\fINIT FAILED", 12);
    SVC_WAIT(500);
  }
  else
  {
    write (STDOUT,"\fINIT OK", 8);

    /*  Wait for selection. */
    while (1)
    {
```

```
        wait(1);
        write (STDOUT,"\fSELECT FUNCTION:", 17);
        wait(1);
        write (STDOUT,"****SEL 1 2 3 4", 15);
        /*                    (1) Send message                  */
        /*                    (2) Show last message received    */
        /*                    (3) Check address                 */
        /*                    (4) Show node list                */
        wait(1);

      }
    }
}
```

C H A P T E R

11

Library Functions

This section is a complete, alphabetical listing of all OMNI 300 Series terminal functions. Each function is documented according to the following format:

Prototype	Defines the function and parameter types and lists header files that should be included when using this function.
Parameters	Explanation of the input and output parameters.
Returns	Defines how to interpret values returned by the function.
Notes	Special comments pertaining to the function.
Related	Related functions or device descriptions documented elsewhere in this manual.
Example	C programming example which shows how the function may be used.