File Transfer

November 1996 **LONWORKS Engineering Bulletin**

Introduction

The purpose of this bulletin is to establish a standard method of exchanging large amounts of data between nodes in a LONWORKS network. The largest practical amount of data that can be transferred in a single LonTalk® packet is 228 bytes, but this file transfer protocol breaks up data files into packets containing 32 bytes of data and transfers the packets sequentially. The size of the packet is fixed at 32 bytes for interoperability and low node cost, but may be increased if there is no interoperability requirement. Larger packets may require off-chip RAM to store application and network buffers on a Neuron® 3150® Chip and thus increase the cost of the node. This file transfer method can be used with nodes based on stand-alone Neuron Chips, or nodes based on host processors attached to LONWORKS network interfaces, such as the Serial LonTalk Adapter (SLTA), the PC LonTalk Adapter (PCLTA), or interfaces based on the Microprocessor Interface Program (MIP).

Data exchange between nodes in the network is modeled as a file transfer, where a file is a stream of bytes, accessed sequentially either for read or for write. The implementation of a file system is not part of this specification. The file system is dependent on the environment of each node, for example whether it is a stand-alone Neuron Chip, or a host-based node. If it is a host-based node, no assumptions are made about the host file system, for example the format of a file name, the existence of any particular directory structure, or file typing. In this way, any node may participate in a file transfer.

Three logical nodes participate in a file transfer operation: the Initiator, the Sender, and one or more Receivers. The Initiator node may also be the Sender or one of the Receivers. In this way, a node may initiate the transfer of a file from another node to itself, or a node may initiate the transfer of a file from itself to one or more other nodes.

The set-up of a file transfer is implemented with network variables of standard types (SNVTs) that allow the Initiator to communicate with the Sender and the Receivers. The actual transfer itself is implemented with explicit messages, using a windowed protocol.

Windowed Transfer Protocol

Most data packets are sent with unacknowledged service, with a request/response packet sent every six packets. These five unacknowledged packets and the sixth request/response packet constitute a data window. This windowing protocol avoids the overhead of acknowledging every packet, but allows recovery from a lost packet no more than six packets later. Each packet contains 32 bytes of data, so that standard buffer sizes may be used on each node, without the need for additional RAM space. The response from the Receivers to the Sender includes an indication of the last packet in the window that was successfully received, incremented by one. The Sender needs to buffer the last data window, in case it receives a request from the Receivers to retransmit one or more packets from that window. For example, if no errors occur, figure 1 shows two successive data windows. Packets marked U are unacknowledged. RQ is a request packet from the Sender, and RSP is a response from the Receiver containing the number of the last packet successfully received incremented by one, which is 5 + 1 = 6.

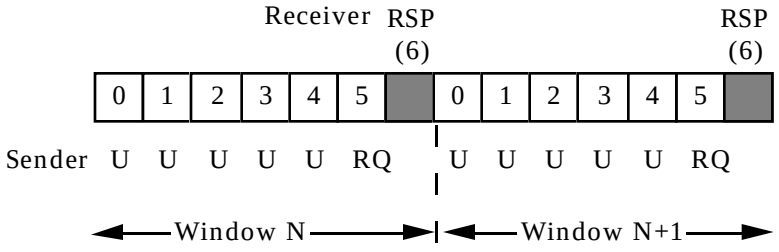


Figure 1 Window successfully received

Figure 2 shows an example of error recovery. If packet 3 in the window is not received correctly, the response packet RSP contains a 3, and the Sender retransmits the window starting with packet 3. Note that packets 3, 4, and 5 are retransmitted to avoid the need for the receiver to buffer packets 4 and 5. In the case of a multicast transfer, the error recovery begins with the lowest numbered packet not received.

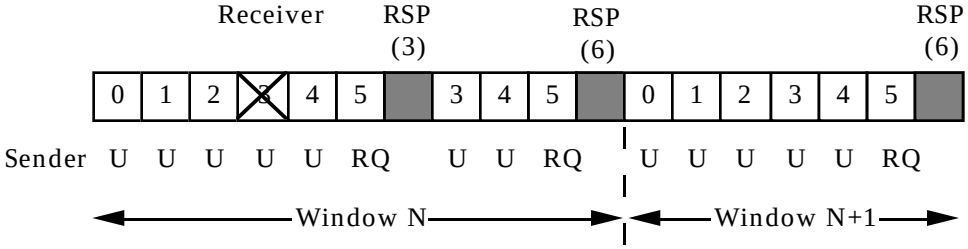


Figure 2 Window with retransmission

Setting Up a File Transfer

There are two Standard Network Variable Types used to set up a file transfer. `SNVT_file_req` is used for communication from the Initiator to the Sender and Receivers. `SNVT_file_status` is used for communication from the Sender and Receivers to the Initiator. These data structures are pre-defined in the Neuron C Compiler. See Appendix A, and also The *SNVT Master List and Programmer's Guide,* part of the *LONWORKS Interoperability Packet.* In the case where the Initiator is the same as one of the other nodes, the network variables are turn-around, meaning that outputs are connected to inputs on the same node. Note that the Initiator can choose to monitor and control the file transfer NVs either by being bound to them or by using explicit updates and polling.

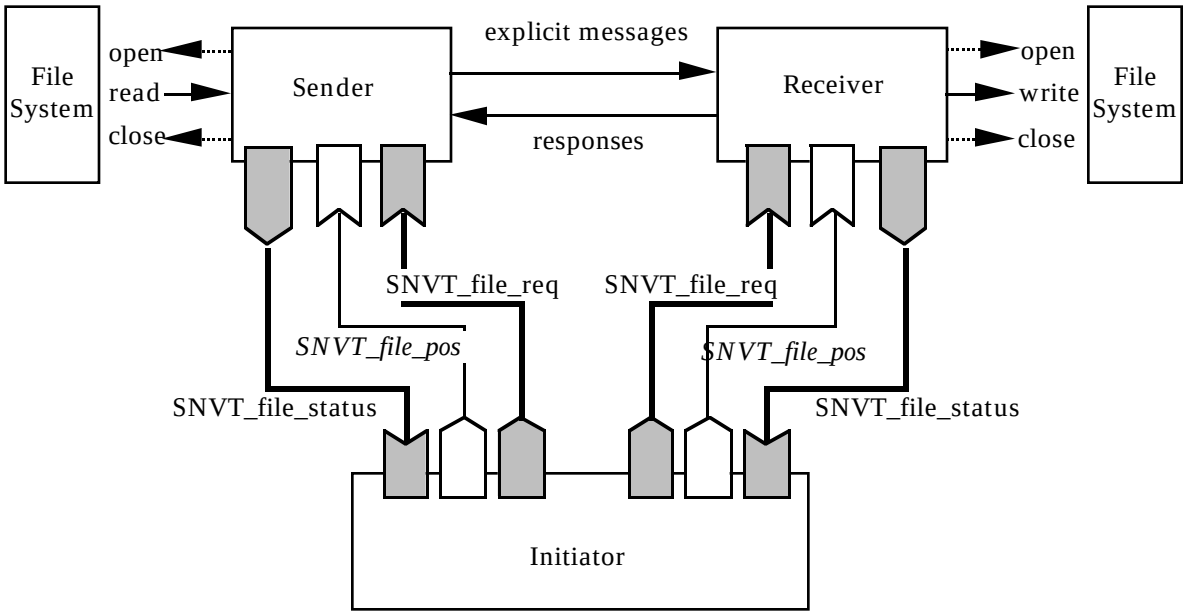


Figure 3 File Transfer Architecture

To start the transfer, the Initiator updates the network variable of type `SNVT_file_req` on the Receivers in order to open the destination files for writing. The result of the file open operation is returned in an update to the network variable of type `SNVT_file_status` on the Receivers. If all the Receiver files were successfully opened, the Initiator then uses the same procedure to open the source file on the Sender for reading. At the same time, the Initiator passes the network address of the Receivers to the Sender. The Sender uses this address to pass the file data in explicit messages to the Receivers. At the conclusion of the transfer, the Sender informs the Initiator that it has transmitted the whole file, and the Initiator then informs the Sender to close the file and Receivers to close and save the files.

During data transfer and setup, several potential errors can occur. The Sender or Receivers can fail to open the specified files. In this case, the initiator backs the other nodes out of the operation appropriately. The Sender can detect a read error (other than end-of-file), and the Receivers can detect a write error. If a Receiver fails to receive a data packet within a configurable time-out, it closes and restores its output file, aborts the transfer and informs the Initiator. The Initiator passes this timeout value to the Receivers as part of the setup request. Similarly, if the Sender fails to receive a response to one of its requests to the Receivers, it closes its input file, aborts the transfer and informs the Initiator. Any error prior to the complete file transfer causes any partially received files to be restored to their original state.

Random Access

The random access protocol allows a file to be opened and then to be transferred in a piecemeal fashion. The Initiator initiates transfers of a subset of the file by performing a seek operation first on the Receiver and then on the Sender. The seek includes an offset and a length. Multiple seeks may be executed within the scope of a single open/close by the initiator. The length must be the same in both the Sender and Receivers while the offsets may differ.

Each seek causes an exchange which is virtually identical to that of a full file transfer. The term data exchange is used to refer to the process which occurs following each seek as well as that during a full file transfer. A full file data exchange differs from that of a seek in that a full file data exchange always starts with window 0 whereas each seek data exchange starts with a window number one greater than the previous (starting with 0).

Delayed Responses

An Initiator normally expects that following the update of the file transfer request NV, the status NV will immediately be updated to indicate a success or failure condition. However, it is possible that a node accessing files on a disk or network file system may require some extra time to prepare the status NV response. To this end, the Initiator must tolerate a "working on it" response to either an open, close, or look-up operation. Read, write and seek operations must be responded to immediately. This may require buffering of data in the node to satisfy this requirement.

The "working on it" response to an open or look-up operation is FS_XFER_OK. Note that for a look-up operation, the status NV union must contain the requester address rather than directory information data. The "working on it" response to a close request is either FS_XFER_UNDERWAY or FS_SEEK_WAIT depending on whether a random access transfer is under way.

In a polling situation, upon getting a "working on it" response, the Initiator should wait for a period of time, such as one second, before trying again to get the status.

Completing a Data Exchange

Senders define normal completion of a data exchange as the transmittal of all data within the file or, for random access, transmittal of the number of bytes specified in the seek operation. Receivers define normal completion of a data exchange as the receipt of a file transfer packet with less than the maximum amount of data. It is implementation dependent whether a Receiver treats receipt of more or less data than the file length (or seek length for random access) as an error. However, it is recommended that an FS_IO_ERR be returned if the data exchange is too long. Note that it is incumbent upon the Sender to always transmit the last packet for a data exchange with a length less than the maximum, even if it means transmitting a packet with zero data.

Completing a File Transfer

Whether there are errors or not, it is always the Initiator that closes the files on the Sender and the Receivers. This guards against race conditions which could otherwise occur in multiple initiator scenarios. To guard against the Initiator never closing the file, the Sender and the Receivers must have the ability to do a local close in the event that the Initiator does not close the file in a timely manner, for example, within one minute after an error condition or normal completion. Furthermore, the Initiator must close the file on the Sender first then the

Receivers. This protects against race conditions where the Sender is still sending while the Receiver is closed and reopened by another Initiator.

Multicast File Transfers

A multicast file transfer requires that the Sender be given a group number as the destination. The group size specified by the Initiator should include the Sender even if the sender is not a member of the group. The group may also include nodes which are neither Senders or Receivers as long as there is no chance of ambiguity when they receive the file transfer messages. That is, these nodes can not also be potential Receivers. Such nodes are not included in the group size and thus do not respond. The typical case of this is where the Initiator is bound to the Receivers via a group and then tells the Sender to use that same group for the transfer.

Concurrency

It is possible for a single Initiator or multiple Initiators to concurrently conduct file transfers involving the same Sender and/or Receivers. This requires that the Sender and/or Receivers have multiple sets of file transfer NVs. It is the Initiators job to find an available set of file transfer NVs. If multiple file transfer NVs are defined, they must be defined in NV arrays. A set of file transfer NVs is thus grouped based on their common array index. Note that an exception to this is that LONMARK® nodes may delineate a set of NVs by virtue of their belonging to a common LONMARK object.

Note that regardless of the number of file transfer NVs, a Receiver can not have multiple incoming files in progress because it has no way to differentiate incoming file transfer messages. An exception to this rule is if the Receiver/Initiator are the same node then there may be multiple simultaneous incoming transfers.

Multiple concurrent file transfers using a single set of NVs is possible but problematic. First, it does not work for random access transfer because SNVT_file_pos does not include a file index. Second, it requires that the Initiators be bound to the Sender/Receivers but there is no means for an Initiator to determine that such bindings are required. Third, it complicates multicast transfers in term of determining which group to use as the Sender's destination. For these reasons, this form of operation is not considered to be interoperable.

`SNVT_file_req` Data Structure

In order to avoid operating system dependencies, a file on a Sender or Receiver is identified with a unique 16-bit number called the file index. This allows up to

65,535 files to be identified on any node. The file index is used as an argument to file open and directory lookup operations.

The network variables of type SNVT_file_req contain an operation code and a file index. When a node receives a network variable update of this type, it performs the indicated operation, and returns the status of that operation in a network variable of type SNVT_file_status. The request operation codes are defined in the Neuron C include file SNVT_FR.H as follows:

```
0 FR_OPEN_TO_SEND
1 FR_OPEN_TO_RECEIVE
2 FR_CLOSE_FILE
3 FR_CLOSE_DELETE_FILE
4 FR_DIRECTORY_LOOKUP
5 FR_OPEN_TO_SEND_RA
6 FR_OPEN_TO_RECEIVE_RA
```

The request functions are:

FR_OPEN_TO_RECEIVE

Opens the indicated file for writing. The Receiver node executes a file open operation. The request also contains a timeout value in milliseconds to be used to recover from Sender node failures. Status returned may be FS_OPEN_FAIL or FS_XFER_UNDERWAY. FS_XFER_OK can also be returned to indicate a delayed response (see "Delayed Responses" above). Once a file is open, the node will reject all further attempts to open a file until the file is closed.

FR_OPEN_TO_SEND

Opens the indicated file for sequential reading. Additional parameters to this request are a destination explicit address, and two booleans to indicate whether authenticated and/or priority messaging should be used. If there is only one Receiver, the destination explicit address is a subnet/node address. If there is more than one Receiver, it is a group address. The explicit address also contains a retry count and a transaction timer to be used for the request/response message at the end of every window. The Sender node executes a file open operation, and begins a file transfer by sending packets to the indicated nodes on the domain in which the FR_OPEN_TO_SEND was received. Status returned may be FS_OPEN_FAIL or FS_XFER_UNDERWAY. FS_XFER_OK can also be returned to indicate a delayed response (see "Delayed Responses" above). Once a file is open, the node will reject all further attempts to open a file until the file is closed.

FR_CLOSE_FILE

Closes and saves the specified file. Status returned is FS_XFER_OK. The status can also be left at its current value to indicate a delayed response (see Delayed Responses above).

FR_CLOSE_DELETE_FILE

Closes and backs out any changes to the specified file. Status returned is FS_XFER_OK. This is used for backing out of an aborted transfer. The file is restored to the state it was in prior to the start of transfer. Note that the "DELETE_FILE" name is somewhat of a misnomer in that the file remains in the directory.

FR_DIRECTORY_LOOKUP

Retrieves directory information for the specified file. Status returned is FS_LOOKUP_OK or FS_LOOKUP_ERR. FS_XFER_OK can be returned to indicate a delayed response (see "Delayed Responses" above).

FR_OPEN_TO_SEND_RA

Same as FR_OPEN_TO_SEND except it opens the indicated file for reading using random access. The normal status is FS_SEEK_WAIT rather than FS_XFER_UNDERWAY.

FR_OPEN_TO_RECEIVE_RA

Same as FR_OPEN_TO_RECEIVE except it opens the indicated file for writing using random access. The normal status is FS_SEEK_WAIT rather than FS_XFER_UNDERWAY.

SNVT_file_status Data Structure

The status field in the file status structure contains the status of the last honored request to that node. As long as the node is the process of a data exchange, the status is FS_XFER_UNDERWAY. If the node is awaiting a seek operation, the status is FS_SEEK_WAIT. At the end of the transfer, the status becomes FS_XFER_OK. If a file read or write operation fails, the status is FS_IO_ERR. If the transfer was aborted due to a time-out or transaction failure, the status is FS_TIMEOUT_ERR. If a window is received out of sequence, the Receiver node's status is FS_WINDOW_ERR. The returned status codes are defined in the Neuron C include file SNVT_FS.H as follows:

```
0 FS_XFER_OK
1 FS_LOOKUP_OK
2 FS_OPEN_FAIL
3 FS_LOOKUP_ERR
```

```
4 FS_XFER_UNDERWAY
5 FS_IO_ERR
6 FS_TIMEOUT_ERR
7 FS_WINDOW_ERR
8 FS_AUTH_ERR
9 FS_ACCESS_UNAVAIL
10 FS_SEEK_INVALID
11 FS_SEEK_WAIT
```

The status structure always contains the number of files on the node, and the index of the file that was the subject of the last operation.

If the last operation was an FR_OPEN_TO_SEND, FR_OPEN_TO_RECEIVE, FR_OPEN_TO_SEND_RA or FR_OPEN_TO_RECEIVE_RA operation, the data structure returned from a Sender or Receiver to the Initiator always contains the full (domain, subnet, node) address of the Initiator. This is for the case of multiple Initiators when there may be several operations attempted concurrently on the same set of file transfer NVs. Each Initiator is responsible for checking its own address against the value returned in the file status structure to ensure that it was granted the requested access. An Initiator must not close a file (i.e., as part of its error handling) unless it was granted access.

If the last operation was a successful FR_DIRECTORY_LOOKUP operation, the status structure contains the directory entry for the specified file. The directory entry is composed of a 16-bit file type, a 32-bit file size, and a 16-character file information array. The latter can be used for any purpose though a NULL terminated ASCII string is recommended. Neither the type nor information string fields have any significance to the file transfer software, they are provided as a convenience to the application. For example, the information string may be used as a file name for a host operating system.

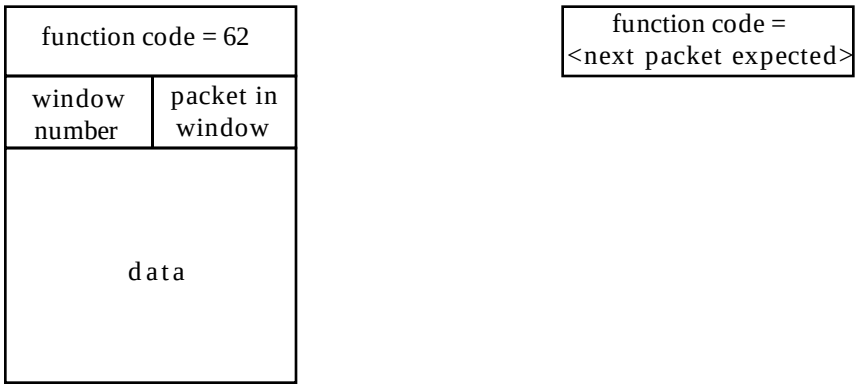
It is recommended that file types be at least 256 to avoid conflicting with file types that may be assigned by the LONMARK Association.

SNVT_file_pos Data Structure

This structure is used when the file is opened for random access. It contains a 32-bit file position value, representing a byte offset from the beginning of the file. If the specified offset is beyond the end of file, a status of FS_SEEK_INVALID is returned. This structure also contains a 16-bit byte count which specifies the length of the next file transfer. The file must be opened using one of the random access modes (FR_OPEN_TO_SEND_RA or FR_OPEN_TO_RECEIVE_RA). If the node does not support random access then it will not have a SNVT_file_pos NV and it returns a status of FS_ACCESS_UNAVAIL to a random access open request.

Application Protocol Data Unit Structure

Each packet of data is transmitted in an explicit message with a message code of 62, a one-byte header, and 32 bytes of data. The message code of 62 is reserved by the *LONMARK™ Application Layer Interoperability Guidelines* for interoperable file transfer. The most significant nibble of the header contains the window number (modulo 16), and the least significant nibble contains the packet number within the window (0-5). The Receivers check that packets are received in sequential order, and respond with a request for retransmission starting from the first packet not received correctly within the current window. Receivers also check that windows are received in sequential order. This is to handle the case where one Receiver has received all of a window successfully, and another Receiver has not received the first packet of that window. The window number allows the Receivers to distinguish a retransmission of the current window from the start of a new window.



Application Protocol Data Unit for packets from Sender to Receiver (Unacknowledged or Request)

Application Protocol Data Unit for packets from Receiver to Sender (Response)

Figure 4 Application Protocol Data Unit structures

Operating System Dependencies

The file transfer software does not implement a file system. This must be provided by the implementor. For example, on a Neuron Chip-hosted node, a file on a Sender node may be a RAM array, or a data stream acquired from some sensor in real time as the transfer is proceeding. On a node implemented with a host computer, a file may be a real disk file, or some other sequential device. The timeouts specified in the file open requests should take into consideration the time needed to obtain the data with a read operation on the Sender node, and to dispose of the data with a write operation on the Receiver node.

The file system must provide the following operations:

```
file_handle open(file_index, operation);
```

This opens the specified file for reading or writing, and returns a file descriptor to be used for later read or write operations.

```
int read(file_handle, void *, int);
```

This reads the specified number of bytes from the file into the specified buffer, and returns the number of bytes read. If this is less than the number of bytes specified, an end-of-file has been reached. If an error has occurred, -1 is returned.

```
int write(file_handle, void *, int);
```

This writes the specified number of bytes from the specified buffer into the file, and returns the number of bytes written. If this is less than the number specified, an error has occurred.

```
void close(file_handle, int backout);
```

This closes the file. If `backout` is `TRUE`, the changes are backed out.

If random access is supported, it must also provide the following:

```
int seek(file_handle, offsetType, lengthType);
```

Implementation

Two model implementations of the file transfer protocol are available. One is in Neuron C for Neuron Chip-hosted nodes, and the second is in C for PC-hosted nodes. These examples may be downloaded from Echelon's website at http://www.lonworks.echelon.com.

The Neuron C model implementation contains five files:

The file `FILE.H` contains the definitions of the types `file_index`, `file_descriptor` and `file_packet`, as well as the defined symbols `FILE_PACKET_SIZE` (32) and `FILE_WINDOW_SIZE` (6).

The file INITIATOR.NC contains the code necessary to initiate a transfer between a Sender and one or more Receivers. Random access file transfer support is not included. The user defines the parameters of the transfer, which are:

Parameter	*Type*	*Meaning*
sender_index	file_index	The index of the source file on the Sender node
receiver_index	file_index	The index of the destination file on the Receiver nodes
auth_on	boolean	Whether to use authentication for the request messages
prio_on	boolean	Whether to use priority for all messaging
receiver_timeout	unsigned long	Timeout for the Receiver nodes in msec
sender_timeout	unsigned long	Timeout for the Sender node in msec
retry_count	unsigned int	Number of retries for the request messages

This implementation makes the following assumptions. These are for simplification of the implementation and are not requirements for interoperability.

- The parameters for all Receivers (in the multicast case) must be the same, specifically, the file index and the receiver timeout.
- The Initiator is bound to the Sender and Receivers. In the multicast case, the group connecting the Initiator to the Receivers contains only the Initiator and Receivers (i.e., no other members, no group overloading).

The file FILEXFER.NC contains the code for the Sender and Receiver nodes. It implements random access file transfer as well. Any of Sender, Receiver or Random Access capability can be removed via conditional compilation. An additional conditional compilation option to simulate network errors is provided and is off by default. FSYS.H contains a skeleton for a file system on the Neuron Chip. The code is marked to show the places in which the real file system operations should be implemented.

On a 1.25 Mbps twisted pair channel, the maximum file transfer throughput for this interoperable model implementation is 2.0 kbytes/second. If the packet size is increased to 225 bytes, and the window size to 15 packets, then the throughput becomes 5.0 kbytes/sec, but this implementation is not interoperable.

Appendix A. Definitions of File Transfer SNVTs

These structures are implicitly defined by the Neuron C compiler.

```
typedef struct {
    file_request    request;
    unsigned long   index;
    unsigned long   receive_timeout;
    union {
        struct {
        unsigned    type;    // 1 for subnet/node
        unsigned    domain  : 1;
        unsigned    node    : 7;
        unsigned            : 4;
        unsigned    retry   : 4;
        unsigned            : 4;
        unsigned    tx_timer: 4;
        unsigned    subnet;
        } sn;
        struct {
        unsigned    type    : 1;    // 1 for group
        unsigned    size    : 7;
        unsigned    domain  : 1;
        unsigned            : 7;
        unsigned            : 4;
        unsigned    retry   : 4;
        unsigned            : 4;
        unsigned    tx_timer: 4;
        unsigned    group;
        } gp;
    } dest_address;
    int    auth_on;
    int    prio_on;
} SNVT_file_req;

typedef struct {
    file_status   status;
    unsigned long number_of_files;
    unsigned long selected_file;
    union {
        struct {
            char     file_info[ 16 ];
            unsigned size[ 4 ];
            unsigned long type;
        } descriptor;
```

```
        struct {
            unsigned domain_id[ 6 ];
            unsigned domain_length;
            unsigned subnet;
            unsigned node;
        } address;
    } adr;
} SNVT_file_status;

typedef struct {
    unsigned      rw_ptr[ 4 ];    // 32-bit value compatible with s32_type
    unsigned long rw_length;
} SNVT_file_pos;
```


005-0025-01 Rev D

Echelon Corporation
4015 Miranda Avenue
Palo Alto, CA 94304
Telephone (415) 855-7400
Fax (415) 856-6153

Echelon Europe Ltd
Elsinore House
77 Fulham Palace Road
London W6 8JA
England
Telephone +44-181-324-1800
Fax +44-181-563-7055

Echelon Japan K.K.
Kamino Shoji Bldg. 8F
25-13 Higashi-Gotanda 1-chome
Shinagawa-ku, Tokyo 141
Telephone (03) 3440-7781
Fax (03) 3440-7782