Software Fax and Data Intercept/Relay System Specification

MESi
10909 Lamplighter Lane
Potomac, Maryland 20854
E-mail: sales@mesi.net
Web: www.mesi.net

Revision History

Revision	Date	Author	Comments
1.0	03Jan2001	PBM	Original Release
1.1	08Jan2001	MTY	T.30 Interface Specification
1.11	09Jan2001	MTY	Incorporate changes from 09 Jan brain drain
1.12	06Feb2001	MTY	TEP Mods
1.13	04Mar2001	MTY	Theory of Operation and format cleanup
1.2	20Apr2001	CWB	Added T.38 info
1.21	26Apr2001	MTY	Added theory of operation on data underruns
1.3	04May2001	MTY	Added file descriptions, dataflow diagram and explanation of message flow within the relay
2.0	12Sept2001	CWB	Reformatted document to be more in line with a System Specification and updated based on current product capabilities
2.1	16Nov2001	CWB	Updates with additional T.38 info
2.2	03Dec2001	MTY	Moved test section to separate document
2.3	04Jan2002	MTY	Updates on T.38 spoofing implementation

TABLE OF CONTENTS

1 Applicable Documents

The following documents are relevant to this specification.

1. International Telecommunications Union T.30 Amendment 1, Series T: Terminal Equipments and Protocols for Telematic Services, Procedures for document facsimile transmission in the general switched telephone network, July 1997.
2. International Telecommunications Union T.38, Series T: Terminals for Telematic Services, Procedures for real-time Group 3 facsimile communication over IP networks, June 1998.
3. International Telecommunications Union T.4, Series T: Terminal Equipments and Protocols for Telematic Services, Standardization of Group 3 facsimile terminals for document transmission, July 1996.

2 Background

2.1 Overview

The Software Fax and Data Intercept/Relay is composed of three separate products; a fax relay, a fax line monitor and a terminating fax. This integrated product has been demonstrated running on a PC in C source, in real-time, utilizing a standard soundcard to interface to a telephone hybrid or hookswitch. This demonstration is an indication of the portability of this product to any platform that supports a C compiler. Figure 1, below, illustrates the functional capabilities of the fax relay product.

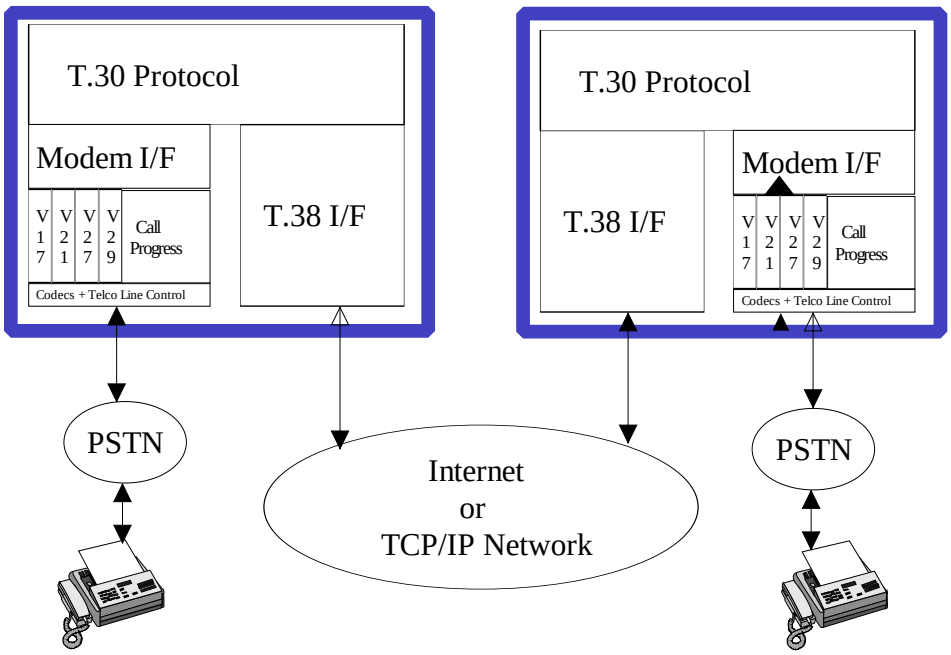


Figure 1 – MESi Fax and Data Relay Functional Diagram

3 Product Plan

The Software Fax and Data Intercept/Relay (SFDIR) will ultimately exist as a “black box” with a Subscriber Line Interface Circuit (SLIC) on one side, and a generic network command/status and data interface on the other side. As with other MESi software products, the SFDIR shall be build-time configurable to achieve a multiplicity of functions from the same software base. When specifically configured, it will be able to emulate an end-to-end dial-up network connection between two subscribers, providing voice pass-through, and fax and data relay. The design for the fax section will allow ultimately for a selection between relay and terminating fax modes, and the data section will allow ultimately for a selection between relay and terminating v.42 modes. In each case, the system operation from end-to-end shall be transparent to the user.

MESi has traditionally been a General Purpose DSP software vendor, concentrating on Texas Instruments and Analog Devices DSP 16-bit integer devices. Over the past year the demand for C Source algorithms and systems specifically tailored for use as a porting source for other processors has risen. Recognizing this trend, the SFDIR will be developed in C in such a way as to facilitate porting to any 16-bit integer device. The code is written such that 32-bit integer devices are also supported with bit-exactness. All of MESi’s existing modem, tone, and signaling products have been developed in this way (i.e. the VSIM simulation), and the result has been a minimized effort in porting to various DSPs (TI CC5000, C6000, ADI 21xx and BlackFin, LSI Logic ZSP). The SFDIR will extend this philosophy by initially simulating the complete system, including the Initiator SFDIR with telephone line impairments simulation, network simulation to include fixed and variable delay and packet loss, and a Responder SFDIR with line impairments. This simulation is built to run under the Windows 98 Operating system on a PC. It will be stimulated initially using digitized fax calls stored as 8 kHz samples. It is important to develop the simulation such that the components are separable and partition-able so that simulated portions can be selectively replaced with real-time portions as the system design progresses. The PSTN interface will then be used to confirm operation over conventional dial-up lines. The network interface will be tested first with a private IP network, and then over the internet to verify proper operation in the presence of network delays, jitter and lost packets.

3.1 Dumb Demod/Re-mod Fax Relay

The most direct method to build a fax relay is to implement a “dumb demod/remod” relay. The existing MESi Fax modem Bundle (GenDet, V.17, V.21, V.27, V.29) implements sufficient signal processing algorithms to detect and demodulate all signals associated with a complete fax transmission. This form of relay operates by monitoring the GenDet detector state ID, passing the state ID and time-tag information across the network, and regenerating the same detected signals on the other side. Demodulated data are passed through unaltered (HDLC, TCF, and page data), including post-burst garbage bits. The only data interpretation is for HDLC messaging that contains modulation format and rate information so the high-speed modems can be properly initialized.

3.2 Hybrid T.30 Fax Relay

The Hybrid T.30 relay interprets the detected HDLC and TCF information, and relays protocol codes and message content to the other side for regeneration. This method “knows” the T.30 protocol elements, but simply enhances the detector by validating the HDLC data exchange and stripping out unneeded data (such as flags and TCF zeros).

3.3 Full T.30 Fax Relay

The full T.30 fax relay implements a T.30 state machine that interacts locally with the fax machine as if it were a stand-alone fax. Like the Hybrid relay, the Full T.30 relay passes only the required content data across the network along with time-tag information

3.4 Train-Thru Data Relay

This mode of operation is required when an ATM interface is utilized for the network transport mechanism. The SFDIR will be compliant with this method to support fax deliver via ATM.

3.5 Locally-Trained Data Relay

This will be the normal mode of operation for the SFDIR. MESi will comply with the relevant sections of the referenced ITU recommendations.

4 System Design

The Fax Relay has three major elements and some support functions. The main elements are the T.30 protocol layer, the fax modem suite and the network interface. The design allows for the easy replacement of the modem and network layers. ATM, TCP/IP or VSAT networks can be used for the fax relay simply by replacing the network protocol file(s). The modem suite can be replaced with another brand of modems by writing a modem interface module that supports the new modem suite.

The T.30 protocol layer is at the core of the fax relay. All messages from the modems and the network layers are processed by the T.30 protocol layer. The major elements communicate by sending messages to the protocol layer. Four linked lists are used to queue messages. Two linked lists connect the protocol and the modem layers, and two lists connect the network and protocol layers.

4.1 Protocol Layer

The T.30 protocol layer is a state machine driven by messages from the modem and network layers and timers. Based on the state and the message, an action routine is called and the state machine can transition to another state. Timeout timers prevent the relay from hanging in a particular state.

4.2 Modem Layer

The modem layer translates the messages from the protocol and network layers into calls to the modems. Since fax modems are half duplex, the modems must only take messages from the link list when they are capable of acting on them or they must return them to the linked list for future processing.

4.3 Network Layer

The network layer is responsible for delivering the messages to the distant relay error free and in sequence order. For details about fax relay operation over IP networks, see the section on T.38 support.

5 T.30 Relay/Modem/Network Communications Protocol

The protocol layer communicates with the modem software and the network software. All communications between the T.30 protocol interfaces is done using buffers. To facilitate communications, there are two buffer queues associated with each interface. For the Modem software, the ToModem and FmModem queues handle all communications. The network interface is serviced by the ToNetwork and FmNetwork queues. The queue numbers are defined in BufMgr.h.

Seven standard function calls allow buffers to be passed between interfaces.

1. **int *BmrGetBuffer(struct SM_struct *S)** - returns the pointer to a buffer or NULL if no buffer is available. The number of buffers allocated should be sufficient for all normal operations. Receiving a NULL should be taken as an indication of a memory leak or other severe problem.

2. **void BmrReturnBuffer(struct SM_structS *, int *buf)** - returns a buffer to the unused pool of buffers. This must be called when an interface is finished with a buffer.

3. **int BmrLinkBuffer(struct SM_struct *S, int QNum, int *buf, int fTail)** - places a buffer on a linked list for use by another interface. There are currently 4 queues defined: ToModem, FmModem, ToNetwork, and FmNetwork which are indicated by **QNum**. **fTail** is TRUE for normal queuing and places the buffer at the end of the linked list. Making **fTail** FALSE will insert the buffer at the top of the linked list ahead of all other buffers queued. This can be useful if a buffer is taken off a linked list and it is determined for some reason that it can not be processed at this point in time. The interface can the place the buffer at the top of the linked list again. This function returns TRUE if successful, FALSE if not.

4. **int *BmrUnlinkBuffer(struct SM_struct *S,int QNum)** - is called to get a buffer from a queue. It accepts the **QNum** as an indication of the list and returns either the address of the buffer or NULL if there are no buffers on the queue.

5. i**nt BmrWriteOctet(struct SM_struct *S, int index, int data, int *buf)** - writes one octet 'data' to a buffer 'buf' at location 'index'. If **index** is larger than the current length of the buffer, the length is adjusted to the new index. The length is needed for the **BmrReadOctet()** call to determine when the end of buffer is reached. . This function returns TRUE if successful, FALSE if not.

6. **int BmrReadOctet(int index, int *buf)** - read one octet of data from buffer 'buf' at location 'index'. It return either the octet of data or –1 if unsuccessful. Reading past the end of buffer will return –1.

7. **int BmrForceLen(int *buf, int len)** - forces the length of buffer '**buf**' to length '**len**'. Normally the length is adjusted automatically using **BmrWriteOctet()**. If however, a buffer is reused and needs to be shortened, **BmrForceLen()** should be called.

5.1 Typical Sequence of Events

A modem process wishing to send a buffer to the T.30 controller informing it that the call is being aborted would do something similar to the following:

```
int *pBuf;
```

```
pBuf=BmrGetBuffer(S);
if (pBuf == NULL)
   {
      /* error routine – no buffer available  */
   }
/* General Header information */
BmrWriteOctet(OFFSET_BUFFER_TYPE, TYPE_FAX_CONTROL, pBuf);

/* Specific Data for Fax Control Message */
BmrWriteOctet(OFFSET_DATA+0, 1, pBuf);
BmrWriteOctet(OFFSET_DATA+1, REASON_DISTANT_END_HANG_UP, pBuf);

/* now send the buffer */
if( BmrLinkBuffer(S, LL_FM_MODEM, pBuf, TRUE) )
   {
          /* error routine, buffer no sent */
   }
```

Similarily, a process servicing a queue would execute code similar to the example below

```
int *pBuf, BufType;

if(  (pBuf=BmrUnlinkBuffer(S,LL_FM_MODEM) ) != NULL )
{
    /* buffer is available */
   BufType=BmrReadOctet(OFFSET_BUFFER_TYPE, pBuf);
   Switch (BufType)
   {
      case CED:
      break;
      case CNG:
      break;
      …
   }
   /* now return the buffer to the empty pool */
   BmrReturnBuffer(S, pBuf);
}
```

5.2 Buffer contents

Each buffer has a common header and then type specific data. The size of the header is currently 3 octets, but may change in the future. In that light, all software written to access data in the buffers should use a #define to indicate the start of user data instead of a hard coded number. The time stamp will be provided by the Buffer Manager at some time in the future. The user need only fill in the Buffer type and the type-specific data.

Offset	Contents
0	Buffer type
1	Timestamp LSB
2	Timestamp MSB
3..N	User Data

The table below shows the buffer types and their descriptions.

Buffer type	Direction xx Modem		Function
	To	Fm	
0	X	X	No signal - optional message that is emitted at 1.0 second intervals on start-up
1	X	X	CED – 2100 Hz called station tone
2	X	X	CNG – 1100 Hz calling station tone
3	X	X	V.21 Flag Detect - optional message that can be sent as soon as the V.21 receiver detects the starting flags. It is sent thru the network so that flags can be generated at the distant end without the penalty of network delays.
4	X	X	V.21 Message
5	X	X	Start Training/Fax Data Message
6	X	X	Continue Training/Fax Data
7	X		Fax Control
8		X	Modem Change
9	X		Receiver Configuration

CED Buffer Specific Data

Offset in User Data	Contents
0	1=Start CED, 0=Stop CED

CNG Buffer Specific Data

Offset in User Data	Contents
0	1=Start CNG, 0=Stop CNG

V.21 Flag Detect

Offset in User Data	Contents
(none)	

V.21 Message Specific Data

Offset in User Data	Contents
0	Count of leading flags received or to be modulated
1	Number of octets of data in frame (lsb)
2	Number of octets of data in frame (msb)
3	De-bitstuffed first non-flag octet of first frame
4	De-bitstuffed second non-flag octet of first frame
5..M-2	De-bitstuffed octets of first frame
M-1	De-bitstuffed first CRC octet of first frame
M	De-bitstuffed last CRC octet of first frame

Start Training/Fax Data Specific Data

Offset in User Data	Contents
0	TEP Format 0x00 None 0x17 1700 Hz 0x18 1800 Hz
1	Modulation Type 0x00 V.27 ter fall back 2400 (1200 baud) 0x20 V.29 9600 0x10 V.27 4800 (1600 baud) 0x30 V.29 7200

	0x04 V.17 14400 0x24 V.17 9600 0x14 V.17 12000 0x34 V.17 7200	
2	Mode	Underrun
	0= No ECM Data	Send 0's
	1= ECM Data	Send Flags 0x7E
	2= Training Data	Send last rcvd packet

Continuation Training/Fax Data Specific Data

Offset in User Data	Contents
0	Sequence number (lsb)
1	Sequence number (msb)
2	Number of octets (lsb)
3	Number of octets (msb)
4..N	Training/Fax data

Fax Control Specific Data

Offset in User Data	Contents
0	1=Abort Call and Hang Up
1	Reason Code

Mode Change Specific Data

Offset in User Data	Contents
0	1=Started Transmitting 2=Stopped Transmitting 3=Detected Carrier 4=Detected Carrier Loss

Receiver Configuration Specific Data

Offset in User Data	Contents
0	

	Modulation Type Values above 0x80 are bit mask vars 0x80 None 0x81 V.21 0x82 CNG 0x84 CED 0x20 OR Mask 0xC0 Auto Detect/Run Values below are for specific fax modems 0x00 V.27ter 2400 0x20 V.29 9600 0x10 V.27ter 4800 0x30 V.29 7200 0x04 V.17 14400 0x24 V.17 9600 0x14 V.17 12000 0x34 V.17 7200 To enable a specific fax modem, use the value associated with it; e.g. 0x34 for V.17 7200 bps. Automatic modem startup is automatically selected for fax modems. To enable V.21 and CED, with automatic modem startup, set 0x81 | 0x84 | 0xC0. To enable V.17 and V.21 with CED, CNG and automatic startup, two separate commands must be issued. The first selects the modem, 0x34 for V.17 7200. The second one will be 0x81 | 0x 82 | 0x84 | 0x20 | 0xC0. The V.21, CNG, and CED detectors will be turned on without removing the V.17 detector. Note that the fax modem must be enabled first, then the other detectors must be OR'd in to the mask.	
1	Mode	Underrun
	0= No ECM Data	Send 0's
	1= ECM Data	Send Flags 0x7E
	2= Training Data	Send last rcvd packet

5.3 Example Message Block

This is a V.21 message sent from the protocol to the modem for modulation. It is a Digital Identification Signal (DIS). See paragraph 5.3 thru 5.3.6 of T.30 for specifics.

Offset from start of buffer	Contents	Meaning
0	0x04	Buffer Type = V.21 Message
1	0x10	Time = 0x0510
2	0x05	
3	0x25	37 Leading flags before first octet of data
5	0x13	There are 19 octets in the first message (octet 7 to and including octet 25) = 0x0013=19.
6	0x00	
7	0xFF	Address field of first message (T.30 para 5.3.4)

8	0xC4	Control field of first message. (T.30 para 5.3.5) Format = 1100 X000, X=0 for non-final frames, X=1 for final frames
9	0x01	Fax Control Field (T.30 para 5.3.6) 0000 0001 = DIS
10..21	0x??	96 bits of Fax Information Field specific to DIS message
22	0xAB	CRC =0xAB1F of message
23	0x1F	

5.4 System Resources

5.4.1 Buffers/Buffer Manager

At the heart of the system is a pool of buffers that the layers use to send (link) messages from one layer to another. The messages can contain V21 or fax data or command and control information. The size of the message is limited to the aggregate size of the buffers. Internally, buffers are appended to each other to give the illusion to the user that there is no limit to the buffer size.
There are nine standard calls to interface to the buffer manager.

1. **int *BmrGetBuffer(struct SeqStruct *q)** - takes the address of the Sequence structure and returns the address of a buffer if available or NULL if the buffer pool is empty.

2. **void BmrReturnBuffer(struct SeqStruct *q,int *buf)** - takes the address of the Sequence structure and the address of the buffer and places the buffer back into the pool of empty buffers.

3. **int BmrLinkBuffer(struct SeqStruct *q,int ListNum,int *buf,int fTail)** - takes the address of the Sequence structure, a list number, the buffer address and a Boolean flag and returns TRUE on success, FALSE on error. This function queues the buffer to the end of the linked list if the Boolean fTail is TRUE or the head if it is FALSE. Normally, buffers are added to the tail end of the linked list, but can be added to the head if the buffer was taken off the list and cannot be processed at the current time. The second parameter is the linked list number. A number is used so that an event can be generated when the buffer is queued. Adding a buffer to linked list 2 causes a BufferLinkEvent+2 to be generated and added to the event buffer. All state machines are checked to see if this event is relevant to them. When a state machine responds to this event, it can remove the buffer from the linked list. This prevents polling of the queues and allows notification to all state machines that an event has occurred.

4. **int BmrLinkLocalBuffer(struct SeqStruct *q,int **List,int *buf,int fTail)** - similar to BmrLinkBuffer except that it acts on a local linked list and not a numbered one. This allows modules to save buffers for further processing. An example is when fax data buffers arrive at the regeneration end. Most likely, the packets arrive well before they are needed since a preceeding V21 message has caused delay in the start of the data generation. In this case, the buffers are taken off the FromNetwork linked list and given to the modems ToModem linked list. Since the modem can not modulate the buffers at this point in time, it temporarily stores them on an internal linked list.

5. **int *BmrUnlinkBuffer(struct SeqStruct *q,int ListNum)** - used to get a message from a numbered linked list. It take the Sequence structure and the linked list number and returns the address of the buffer or NULL if no buffer exists.

6. **int *BmrUnlinkLocalBuffer(struct SeqStruct *q,int **List)** - same as BmrUnlinkBuffer except that it uses a local linked list.

7. **struct BufStruct *BmrInit(struct BufStruct *S)** - initializes the buffer structure. It takes the address of the Buffer structure as a parameter and must be called before any buffer calls are made.

8. **int BmrWriteOctet(struct SeqStruct *q,int index,int data,int *buf)** - used to write octet aligned data to a buffer that was returned from a BmrGetBuffer() call. It accepts the Sequence structure, the offset of the data in the buffer, the data and the address of the buffer as parameters and returns TRUE on success, FALSE on failure. The user does not need to be concerned about the size of the buffer; it is internally regulated.

9. **int BmrReadOctet(int index,int *buf)** - used to get data out of a buffer. It accepts the offset of the data and the address of the buffer. It returns the octet of data on success or –1 on failure.

5.4.2 Timers

There are a finite number of timers available that have a 1 mS granularity. They are driven off the receive sample counter; thus 8 samples equal 1 mS. The number of timers that can be concurrently active is limited by the size of the timer array. The total number of timers in the system is limited by the max value of an int. When a timer expires, the timer number is placed in the event buffer for processing. The timer numbers should be declared in event.h, since they are actually events. Timers can be started, stopped and their time remaining checked by any process. The timer function calls are listed below.

1. **void StopTimer(struct SeqStruct *q, int TimeNum)** - stops the timer TimeNum. If the timer is still in the stack, it is removed from the array of timers. If the timer has expired and is now an event in the event buffer, it is removed from the event buffer. This eliminates any race condition.

2. **void StartTimer(struct SeqStruct *q, int TimeNum, unsigned int Nomst)** - starts a 1 mS timer for Nomst milliseconds. The timer is placed on the array of active timers.

3. **unsigned int ReadTimer(struct SeqStruct *q, int TimeNum)** - returns the number of milliseconds left before TimeNum timer expires. If the timer is not on the array of active timers, ReadTimer returns 0.

4. **void TimerTick(struct SeqStruct *q)** - internal function that must be called every timer period (1 mS). This is the mechanism by which the timers are decremented. TimerTick manages the timer stack and places the timer number in the event buffer when the timers expire.

5.4.3 Events

Events are simply numbered items that correspond to occurrences in the protocol. When a message is received, an event of that type is added to the event buffer. Timers expiring also place their timer numbers in the event buffer. The sequencer reads events out of the event buffer and checks each state machine that is running to see if there is an entry for an event. If so, the action routine associated with that event and state machine is called. Events are declared in event.h.

5.4.4 Sequencer

The sequencer is the heart of the system. Associated with each sequencer are event buffers, state machines and timer stacks.

State Machines are added and removed from the sequencer structure with the following calls.

1. **void AddEvent(struct SeqStruct *q, unsigned int Event)**
2. **void DelEvent(struct SeqStruct *q, unsigned int Event)**

3. **void Sequencer(struct START_PTRS *s)**
4. **int AddStateMachine(struct SeqStruct *q, int StateID, struct ETS *StateTable)** - starts a new state machine for a sequencer. A sequencer can support multiple state machines. If a state machine is added a second time, the StateTable is updated for the first instance of the state machine. Only one instance of each StateID can exist in each sequencer.
5. **int DelStateMachine(struct SeqStruct *q, int StateID)** - removes StateID state machine from the sequencer.
6. **void InitSequencer(struct START_PTRS *)** - initializes the sequencer. It must be call before anything is done with the protocol.

void SeqBufDepEvent(struct SeqStruct *q,int *buf, int Event) - normally events are generated from timers and the arrival of message buffers. However there are some messages that are dependent on the information in the buffer. SeqBufDepEvent is an alternate entry point for event processing. It saves the address of buffer in the sequencer structure and then processes the new event "Event" as if it came from the event buffer. The state machines can then act on the event and use the data in the buffer. If an action routine wishes to save the buffer, it should set the buffer pointer to NULL. The sequencer will then not return the buffer.

5.4.5 State Machines

Each state machine is composed state arrays. Each state array is a snapshot in time of a particular state of the software. The state array consists of arrays of structures. Each structure contains three elements: 1) an event number, 2) an action routine, and 3) the next state. The event number of the last entry of each state must be or'd with LAST_ENTRY, which tells the sequencer that there are no more entries for this state.

5.4.6 Structures

5.4.7 START_PTRS

START_PTRS is the structure that contains pointers to all components of the system. The TRANSMITTER and RECEIVER_START_PTRS are used by the modems. The ModemIf pointer points to the modem interface layer structures and the Seq pointer points to the sequencer structure that runs the protocol layer.

```
struct START_PTRS {
        TRANSMITTER_START_PTRS;
        RECEIVER_START_PTRS;
        struct ModemIfStruct *ModemIf;
        struct SeqStruct *Seq;
        };
```

5.4.8 SequenceStruct

The SequenceStruct contains the variables used by the sequencer. These allow the sequencer to track the events, state machines and timers. The sequencer functions more as part of an operating system than a protocol.

```
struct SeqStruct{
        unsigned int MasterTimer;
        unsigned int nTimers:10;
        struct tmr Timer[MAX_TIMERS];
        struct sm StateMachine[MAX_STATE_MACHINE];
        unsigned int Events[MAX_EVENTS];  /* size of 8 bits yields 128 entry max */
        unsigned int EventHead:8;
```

```
unsigned int EventTail:8; /* size of 10 yields 1024 timers max */
                          /* size of 6 yields 64 state machines max */
int nStateMachines:6;
int *LastRxSampleHead;
int ResidualSamples; /* pRxSampleHead contains the address of the RxSampleHead pointer */
                     /* This contained &START_PTRS->Rx_sample_block->sample_head */
int **pRxSampleHead;
void *pBufMgr;
struct ProtocolStruct *Protocol;
unsigned int RealTime;/* the count in mS since the call began */
int *DepBuffer;
};
```

5.4.9 BufStruct

The Bufstruct is used by the buffer manager to track the pool of empty buffers and linked lists. Both EmptySections and LinkedLists are linked lists. The difference is that a message is composed one or more buffer sections. Each buffer section is linked together to form a message. The empty sections are stored in the EmptySections linked list. Once the message is built, they are stored in the LinkedLists array. This is very similar to the EmptySections linked list, but not identical. Different pointers are used to link complete message buffers as opposed to buffer sections.

```
struct BufStruct
    {
    int *(LinkedLists[NLLISTS]);
    int *EmptySections;
    int nSections;
    int buf[NSECTIONS][SIZE_BUFFER];
    };
```

5.4.10 ModemIfStruct

The ModemIfStruct is unique to the suite of software modems. Variables necessary for the modem operation should be saved here.

```
struct ModemIfStruct{
    int OldRxState;
    int OldTxState;
    int *RxBuf;             /* point to BufMgr buffer for rcvr */
    int RxRaw;
    int RxByte;             /* unpacked data byte */
    int RxCount;    /* count of bytes in current frame */
    int RxBitCount;
    int RxOnesCount;        /* one's count for bit stuffing */
    int RxOffsetMsg; /* offset in buffer of beginning of messaage */
    int RxFlagCount;
    int *TxBuf;             /* pointer to buffer for the transmitter */
    int TxFlagCount;
    int TxState;
    int *TxMsgBuf;   /* pointer to buffer containing active msg to tx*/
    int TxFlagReq;  /* number of flags that need to be modulated */
    int TxOffsetMsg; /* offset of beginning of message in buffer */
    int TxCount;    /* offset from TxOffsetMsg of next octet to tx */
```

```
int TxFrameSize; /* number of octets in current frame */
int TxOnesCount; /* bit stuffing count of ones */
int SeqNr;
int TxDataType;  /* ecm, non-ecm, training */
};
```

6 T.38 Support

The T.38 recommendation provides for real-time group 3 facsimile delivery over IP. This recommendation permits the use of either TCP or UDP over IP for delivery of the facsimile, which would be any IP network, including the Internet. This product is compliant with the T.38 recommendation. A block diagram is shown in Figure 1.

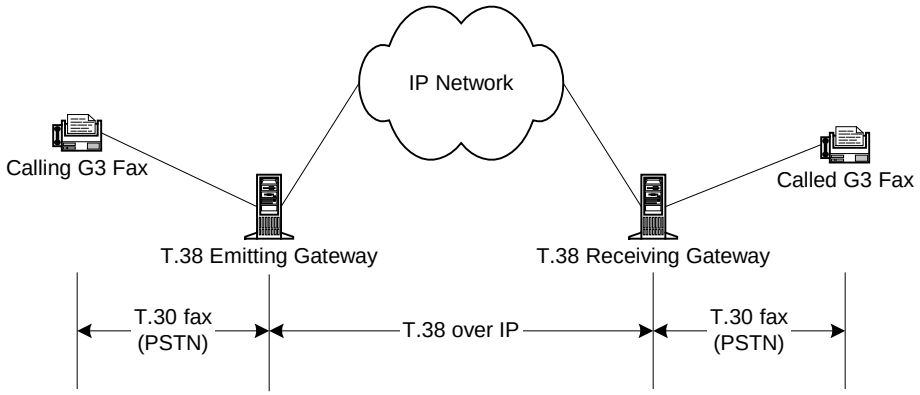


Figure 1 – Facsimile Transmission over IP Networks

The calling fax dials up the emitting gateway via the PSTN. The emitting gateway locally handles the CNG, CED and TCF tones. It should be noted that TCF is handled locally only when TCP is utilized over IP. When UDP is utilized over IP, TCF is generated by the called fax. The emitting gateway will indicate to the receiving gateway the detection of these tone signals in order that it may regenerate them to the called fax. Normal T.30 procedures previously defined are used for establishment and transfer of the facsimile information. The receiving gateway dials the called fax and completes the virtual link between the two fax machines. All communications between the modem layer and the network layer is handled through buffers as previously described.

Both TCP and UDP over IP are supported for communication between the gateways. Mapping of the T.30 is such that bit order between the PSTN and IP networks is preserved. The following describes the packet structures for both TCP/IP and UDP/IP.

In the case of TCP/IP, the IP packet consists of the IP header and the IP payload. The IP payload consists of the TCP header and the TCP payload, which in this case is the IFP packet. The IFP packet is the method of communication between gateways. The IFP packets contain either TYPE or DATA elements or a T.38 packet. The T.38 packet provides an alert for the receiving gateway indicating a start of message and is used to assure message alignment. The TYPE element is used to convey an indicator of fax signals, preamble flags or modulation types, or a T.30 data field. Figure 2 shows the high-level packet structure for IFP packets over TCP/IP.

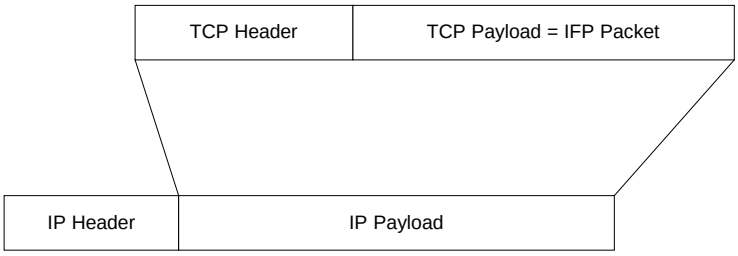


Figure 2 – High-level IFP packet structure over TCP/IP Networks

In the case of UDP/IP, the IP packet is the same as in the TCP case. The IP payload consists of the UDP header and the UDP payload. The UDP payload contains the UDPTL header and UDPTL payload, which in this case is the IFP packet. Figure 3 shows the high-level packet structure for IFP packets over UDP/IP.

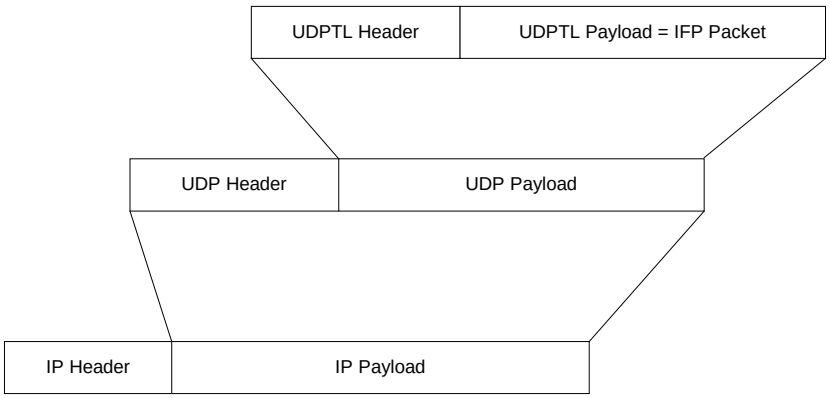


Figure 3 – High-level IFP packet structure over UDP/IP Networks

The UDPTL header contains a sequence number for detection of out-of-sequence packets. The UDP/IP packet also supports message redundancy. If this is selected, the current packet is sent along with the previous two IFP packets, called secondary packets. These secondary packets are appended to the current IFP packet in decreasing consecutive order.

The T30_INDICATOR part of the TYPE element contains signal detection information. This is CNG, CED, V.21 preamble, modulation training for the various modulation schemes, and no signal. The T30_DATA TYPE indicates what modulation was used to convey the information in the DATA element. The DATA element contains HDLC data, and the following signal indications; signal end, FCS bad, FCS good, and Non ECM T.4. Multiple fields within a single IFP DATA element are supported.

The T.38 packet for both TCP/IP and UDP/IP adheres to the ASN.1 protocol as described in Annex A of ITU Recommendation T.38. This packet is encoded using the BASIC-ALIGNED version of Packed Encoding Rules (PER) as per ITU Recommendation X.691.

7 Normal Operations

The current fax relay assumes that the phone connections are made externally. The incoming call is answered and the outgoing call is placed outside of the scope of the relay. A PSTN interface was developed to provide the ability to test the complete SFDIR product with the PSTN, but it is not intended to be part of the final product. This may change at sometime in the future, but the PSTN interface is intended to be provided by the customer.

There is only one flavor of the current relay; it supports two PSTN connections and two virtual network connections to permit operation on an IP network. Normal operations would have one PSTN connection and one network connection. A distant relay would contain an identical network and PSTN connection. Data would be sent over some network from one relay to a distant relay.

The state machine for the each end of the relay waits for CNG tone, CED tone or a DIS message from either the local PSTN connection or from the network. Once one of these events transpires, the fate of both ends of the relay is set. One is setup as the initiator and the other as the responder. This is important since the echoes of the transmitted V21 messages are demodulated by the modems. Since the initiator and the responder do not send the same messages, the echoes can be distinguished by the message types.

The modem interface layer and the network layer communicate with the protocol layer thru a set of linked lists. All communications are done with by passing messages buffers to linked lists. There are three linked lists for each component; 1) linked list from modem, 2) linked list to modem, 3) linked list from network, 4) linked list to network 5) linked list to network protocol translator and 6) linked list from network translator. The last two linked lists provide a streamlined method to map the MESi native protocol to an alternate network protocol; e.g. T.38, I366.2, or proprietary VSAT. There is a common pool of buffers that all of the modules use for communications. A typical sequence of events is described in the table below.

Modem Interface	T.30 Protocol	Network	Network	T.30 Protocol	Modem Interface
Modem I/F detects CNG tone, gets a buffer from the pool of empty buffers, write CNG_ON message in the buffer and places the buffer on the FROM_MODEM linked list.					
	The T.30 Protocol takes the buffer off of the FROM_MODEM list, reads the contents, transitions to the Wait_Net_Dis State and places the buffer to the TO_NETWORK linked list.				
		Network layer takes the buffer off the linked list, copies the contents of the buffer into a network buffer and sends the network buffer on to its peer. The network layer retains the original buffer until the buffer is acknowledged by the distant network.			
			The network layer receives the CNG message from its initiating peer network layer. It gets a buffer from the pool of local buffers, copies the message into the buffer and place the buffer on the		

			FROM_NETWORK linked list. It acknowledges the receipt of the message thru its network protocol so the distant end can return its buffer.		
				The protocol layer takes the buffer, examines it, sees that it is a CNG message, transitions into the Wait_Lcl_Dis state, then places the buffer on the TO_MODEM linked list.	
					The modem removes the buffer from the TO_MODEM list, commands the modem to generate CNG and returns the buffer to the pool of empty buffers.
					The modem detects CED, gets a buffer from the pool of empty buffers, writes a CED message in to buffer and places the buffer on the FROM_MODEM linked list.
				(and so on)	

8 Sequencer Operation

The sequencer is an event driven state machine with a minor processing exception. Since most of the events are related to buffers, there is an additional sequencer call that allows the sequencer to be reentrant and pass the address of a buffer with it. Normally, an event would trigger a state transition and an action routine would be called. In the case of message coming from the modem, the state machine is called multiple times.

For example, when a message is placed on the FM_MODEM linked list, a FromModemEvent is placed in the event buffer of the sequencer. This causes the state machine to take the buffer off the linked list. The message is examined and found to be a V.21 message containing a "DIS" message. The processing routine generates a new buffer dependent event "T30LclDisEvent" and stores the address of the buffer in the sequencer structure. The state machine is invoked again looking for a match for T30LclDisEvent in the state table. When it finds it, it calls the action routine that can then access the buffer. The action routine can then use the buffer manager calls to examine the data in the buffer. If an action routine wishes to take the buffer and save it, it must set the pointer variable "DepBuffer" in the sequencer structure to NULL. The action routine is then responsible for returning the buffer to the empty pool of buffers when it is finished with it. After the sequencer finishes with all of the state machines, it checks to see if the variable DepBuffer is not NULL. If it is not, it returns the buffer to the empty pool. This added complexity allows the sequencer to easily parse and pass messages that are state dependent.

There is one sequencer for each side of the relay. The sequence can support multiple state machines. The number of state machines that the sequencer supports is defined at build time to 5 state machines. Currently two state machines are used by the relays. The first state machine is used for events that occur never change state. T30BufManState primarily processes buffers in the FromModem and FromNet linked lists, but it can be used for any event processing that does not require a state change. This provides a synchronous/non-polled method of processing the buffers. The second state machine is the T.30 protocol for the relay. It starts out in T30IdleState and is there until a CED, CNG, or DIS event starts the process. The T.30 states are depicted in the figure below.

State machines can be added with the AddStateMachine call and deleted with the RemoveStateMachine call. Normally, a state machine is added during processor initialization and never removed.

Timers are also valid events for the sequencer. Timers are in 1 mS increments and can be set to 65.535 seconds for 16 bit implementations. The number of timer concurrently running is a build time define that is currently 16. Timers are implemented on a stack. Each timer has two numbers associated with it; a timer (event) number and the expiration time in mS. There is a master timer for each sequencer. The expiration time of any particular timer is the value of master timer plus the timer value in the stack. Every millisecond the master timer is decremented by 1 until it reaches 0. When the master timer is 0, the timer (event) number is added to the event buffer in the sequencer and the timer stack is reevaluated. The value of the timer with the earliest expiration time is copied into the master timer variable, and all of the timers on the stack are decremented by that amount. This maintains the correct expiration timer for each timer. Timers can be started, stopped and check by any of the action routines. When a timer is stopped, the timer is removed from the timer stack (if it is there) and/or the event is removed from the event buffer (if it is there). This prevents a timer that just expired from triggering a state machine event in error.

9 Buffers

All communications between layers is done through buffers and linked lists. From the viewpoint of the user, a buffer is an infinitely long area where data can be written. The user makes a call to BmrGetBuffer and is returned the address of the buffer. The user can then make calls to BmrWriteOctet and BmrReadOctet to write and read data in the buffers. The buffer protocol does not allow users to access the buffers directly; i.e. the pointer to the buffer cannot be treated as a pointer to an array of chars. When the user finished writing data to the buffer, a call to BmrLinkBuffer is made to send the buffer to another layer. After a layer is finished with a buffer, a call is made to BmrReturnBuffer that returns the buffer to the pool of empty buffers. The number of buffers available is a build time define (NSECTIONS) that is currently set to 370. Each buffer can contain OCTETS_PER_BUFFER (currently 64) octets of data. The number of buffers required depends mainly on the maximum network delay. Since unacknowledged data must be stored in the event a retransmission is necessary, the buffer requirements can be computed directly from the max delay.

Internally, the pool of available buffers is stored in a linked list. When a user desires a buffer, it is taken from the empty pool. As stated above, each buffer can contain 64 octets of data. When a user attempt to write the 65^{th} octet of data to the buffer, a new buffer section is removed from the pool of empty buffers and is linked to the first section. The 65^{th} octet of data of the original buffer is in reality the 1^{st} octet of data of the second section. All of this is hidden from the user. The user needs to make on call to get a buffer (BmrGetBuffer), a series of call to write data (BmrWriteOctet) and one call to return the buffer to the pool (BmrReturnBuffer). The linking, writing of data and returning the sections is done internally.

The user views a buffer as an unlimited resource. In reality, the amount of buffer space available to the user is one buffer of size NSECTIONS*OCTETS_PER_BUFFER (370*64) octets or NSECTIONS (370) buffers of size OCTETS_PER_BUFFER (64) octets or any combination in between these extremes.

10 Handling of Transmit Data Under-runs

Once the transmitter starts sending fax data or training data, it is assumed that the modem will be provided data by the distant end to keep the pipe full. When this does not occur because of delays in the networks, the modems must gracefully accommodate this situation.

The underrun handling is broken down into 3 possible scenarios: 1) underruns during TCF training data, 2) underruns during non-ECM fax data, and 3) underruns during ECM fax data. There is some processing done at both the receiver side and at the transmitter side of the fax relay depending on the mode of operation.

For TCF training data, no special processing is done at the relay receiver. The symbols are collected until the threshold is reached and then the packet is delivered to the network. At the transmitter relay, the symbols are taken out of the packet and sent to the transmit data buffer. If there are no packets available, the transmitter underruns by transmitting '0' symbols until a packet arrives.

For non-ECM data, fill data (0 bits) can be inserted before the end of line (EOL) sequence. The EOL sequence is eleven '0' bits followed by one '1' bit. The processing is divided between the receiver and the transmitter. At the receiving side, the data stream is examined for EOL sequences. If an EOL sequence is found and there are enough symbols in the packet, the packet is delivered to the network. There is a threshold set to prevent sending very short packets that contain a few runs and then an EOL. When an EOL is detected, the symbol that contains the last '1' bit is not put in the data packet. That means that the packet ends with some portion of the '0' bits that are part of the EOL. At the transmitter side, if there is not a packet available, the transmitter simply transmits '0' symbols until the data packet is available. The first symbol of the next packet contains the '1' bit of the EOL and transmission continues normally.

For ECM data, an entire frame of data is contained in a buffer. Since ECM frames are either 64 or 256 octets, with the exception of a short final frame, there will most likely be 1 frame per buffer. Similar to the non-ECM mode, the receiving relay examines the data stream looking for a flag. When a flag is found and the number of octets is greater than the threshold, the packet is delivered. The symbol that contains the final bits of the flag is not transmitted. That symbol will be the first symbol in the next frame. To make processing easier at the transmitter, the number of bits of the flag that are included in the data packet is included in the buffer. At the transmitter side, when an underrun occurs, a number flags are transmitted. The number of flags is equal to the number of bits per symbol. This is so that at the end of this sequence, the position of the in relation to the symbol alignment is the same.

The following example shows the steps. For this example, we are relaying a V.17 12000 bps, or 5 bits/baud. The final two symbols of a frame contain xxxx0 and 11111 and the first symbol of the next frame contains 10yyy, where x and y are don't care bits. The flag character is split between the frames. The receiving relay indicates that 6 bits of the flag character are contained in the buffer. If, at the end of sending the entire buffer, a new buffer is available, it is transmitted. If it is not, the transmitter then composes 5 flags worth of symbols starting with bit number 7 of the flag bit (6 bits were already transmitted).

Here are the final symbols, followed by the 5 flags.
Final symbols….inserted flags
xxxx0 11111 10 01111110 01111110 01111110 01111110 011111

The transmitter then breaks the flags into symbols.
Final symbols….inserted flags broken into symbols
xxxx0 11111 10011 11110 01111 11001 11111 00111 11100 11111

As you can see, after inserting the 5 flags, the position of the flag relative to the symbols is the same. At this point, one of two things happens. If the new buffer has not arrived, the transmitter sends 5 additional flags. If the new buffer is ready, it is sent to the modulator. The end result is a number of flags slipped into the data stream between the frames.

Underrun lasts only 5 flags periods.
Final symbols….inserted flags broken into symbols…….... new frame begins
xxxx0 11111 10011 11110 01111 11001 11111 00111 11100 11111 10yyy

In T.38, ECM data is transferred in complete or partial frames. In this case, complete frames are reassembled at the receiver between the network translator linked list and the network linked list so that the interface to the protocol remains the same.

11 Handling of Non-Standard Facilities

T.30 protocol allows for proprietary, non-standard messaging and protocol using the T.30 NSF, NSS, and NCS message types. The Fax Relay handles these by trapping on received NSF, NSS, or NCS frames and over-writing the country code and manufacturers code with a replacement code that is known to be not recognized – that is, like an unknown manufacturer. The existing frame body content is replaced with all zeros and the FCS is re-calculated and overwritten. Therefore, the NSF, NSS, or NCS frame is relayed with the same duration but with different country and manufacturer's codes and a valid FCS at the end. This action prevents two fax machines from negotiating a non-standard proprietary session, which the relay can not support under T.30, and thus proceeds as a normal fax session.

12 Spoofing

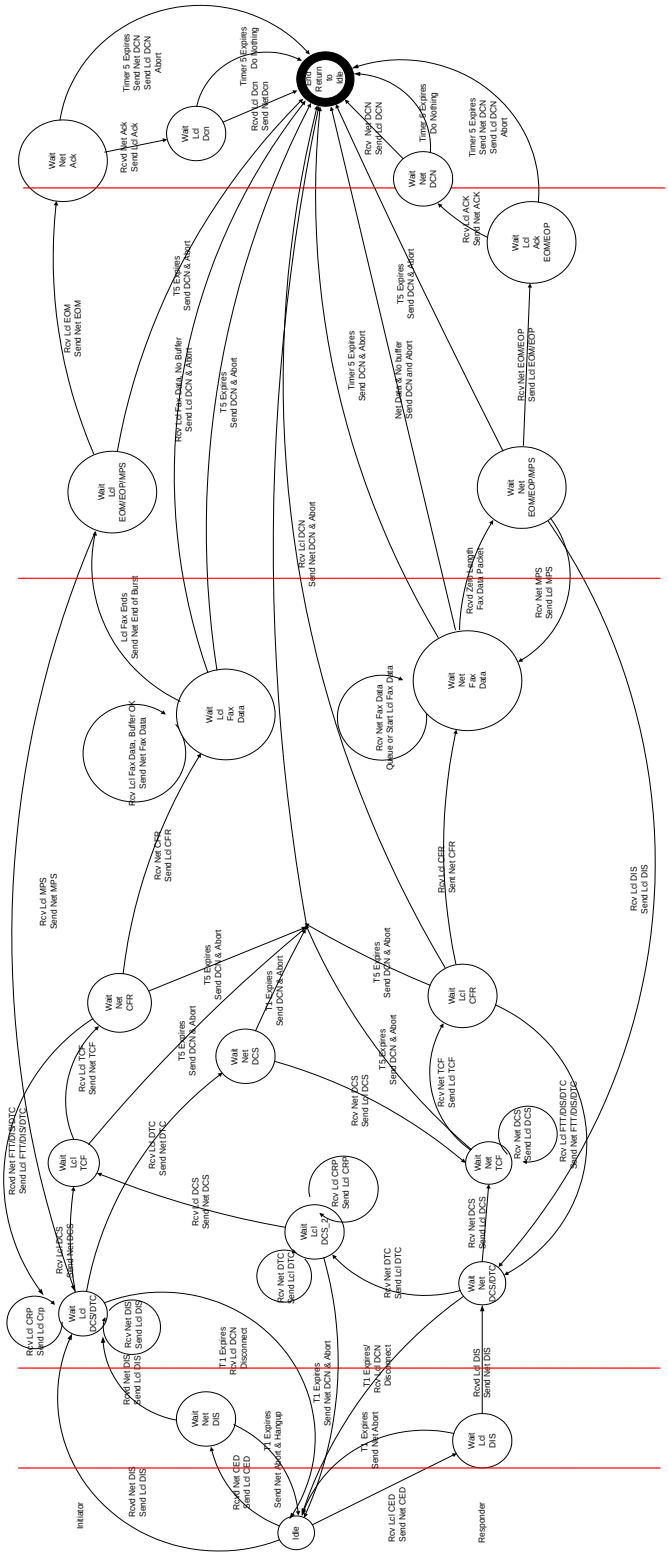
Spoofing is required to compensate for delays, jitter and lost packets that are inherent in a public IP network (i.e., internet). Normal fax machine transmission timeouts are too short for these problems that are encountered on a public IP network. Spoofing will compensate for these problems transparently to the connected fax machines to overcome this limitation. Spoofing will be accomplished by transmitting extra flags for the maximum time allowed when a network response is expected. At the end of the allowed time, a repeat command (CRP) is issued or the transmission is stopped. Then the relay waits for a new command to be issued. If a network response is received, it is saved in a collision buffer, pending the retransmission of the command from the command initiator. When the retransmitted command is received by the relay, it is discarded and the saved collision response is modulated to the local fax machine. In ECM mode, RNRs are sent in response to commands until the responder end finishes transmission. Spoofing is accomplished differently between DIS and DCS. The previously described technique is effective when there is a defined command and response. Since DIS and DCS are both commands, this technique will not work. To accomplish spoofing, a dummy TSI message is sent at the end of the timeout period while awaiting the DCS message. This allows for an additional 3 seconds of delay in the network.

13 Source Files

The following is a list of the source files and a brief description of the contents.

BitRev.c - performs bit reversal.
g711.c - Mu-Law expansion.
netsim.c – bent pipe network simulator.
sym2bit.c – debug code to convert symbols to bits.
tifgen.c – a collection of routines to take a bit stream and convert it to a .tif formatted file.
bufmgr.c – buffer manager code.
modemif.c – The modem interface routines that link the t30 protocol to the modems.
relay2.c – Variable declarations and high level calls to the relay components.
t30.c – The T.30 state machines arrays and action routines.
sequence.c – The sequencer, state machine arrays, and timer calls.
t4.c – Tiff file generation routines.
ipnet2.c – IP network interface code for MESi native protocol.
t38.c - IP network interface code for T.38 protocol.

14 State Transition Diagram



15 Appendix I – Acronyms and Abbreviations

ATM	Asynchronous Transfer Mode
CED	Called Terminal Identification
CFR	Confirm to Receive
CI	Calling Indicator
CNG	Calling Tone
Demod	Demodulator
DIS	Digital Identification Signal
DSP	Digital Signal Processor
ECM	Error Correction Mode
EOM	End of Message
FCF	Facsimile Control Field
FCS	Frame Check Sequence
FTT	Failure To Train
HDLC	High-Level Data Link Control
ITU	International Telecommunications Union
IFP	Internet Facsimile Protocol
IP	Internet Protocol
LSB	Least Significant Bit
mS	milliseconds
MSB	Most Significant Bit
PPS	Partial Page Separator
PSTN	Public Switched Telephone Network
Re-mod	Re-modulator
RNR	Receive Not Ready
SLIC	Subscriber Line Interface Circuit
TCF	Training Check
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VSAT	Very Small Aperture Terminal