Implementing IPSC Voice Terminal using Hardware Dongle

M.H.Retzer
Zhou Shaoqun
Professional & Commercial Radio
Worldwide Radio Solutions
Enterprise Mobility Solutions

***Abstract*- A demo voice terminal application for the IPSC network is described. A PC application implements the IPSC protocols. AMBE voice compression is implemented using a USB "Dongle."**

I. INTRODUCTION

There are valid engineering reasons for distributing voice over an IPSC network using the proprietary Advanced Multi-Band Excitation (AMBE+2™) coding standard developed by Digital Voice Systems, Inc. In the intended application of linking radio repeater sites, use of a more generic codec for the network links would entail re-encoding, or back-to-back *tandeming* of multiple compression schemes. This would result in severe voice quality degradation. There are less valid, but at least practical reasons, for burdening the IPSC network protocols with raw DMR radio air link signaling messages. After free registration, the TS 102 361-1, 2, 3 can be downloaded at http://www.etsi.org/WebSite/homepage.aspx. Because of the cost and complexity of implementing both AMBE and the radio air link messages, we set out to provide a reference approach using readily available hardware integrated circuits controlled by a Microsoft Windows terminal application

.

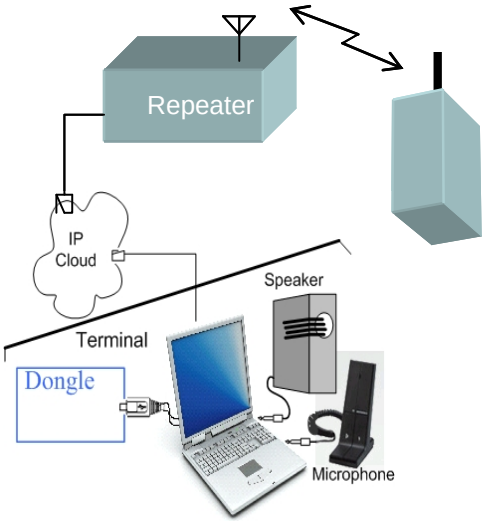


Figure 1 Terminal Hardware

The reference terminal is comprised of a PC supporting an Internet connection to a Mototrbo repeater, a speaker, and a microphone for voice communication. An external hardware *Dongle* connected to the PC by USB implements the DVSI AMBE encoding and decoding. Terminal software running on the PC implements the IPSC protocols, audio processing, scheduling of packets over the IP network, and user interface.

As shown in Figure 2 during voice reception IPSC packets arrive from the Internet connection. The terminal determines if the packets originate from the user selected Talkgroup, and if so the AMBE encoded voice packets are sent through the USB port to the Dongle. The Dongle decodes these packets into PCM

samples, which are returned to the PC over USB. The PC implements jitter buffering on these PCM samples, and supplies them at a uniform rate to the PC sound card, for playback on the speaker.

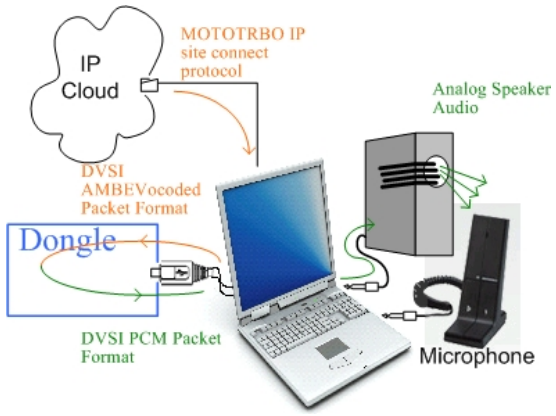


Figure 2: Voice Reception

As shown in during voice transmission, audio from the microphone is sampled at a uniform rate by the PC soundcard. These PCM samples are sent over USB to the Dongle where they are AMBE encoded and returned to the PC. The PC encapsulates these samples along with IPSC control protocol, and transmits them over the IP network to the Repeater.

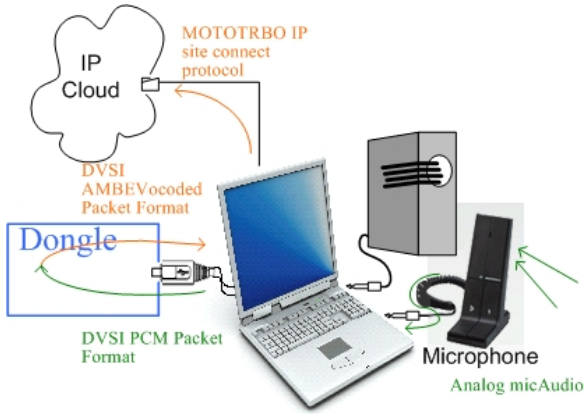


Figure 3: Voice Transmission

This reference design is provided “as-is”; no warranty nor technical support is implied.

II. DONGLE HARDWARE AND FIRMWARE

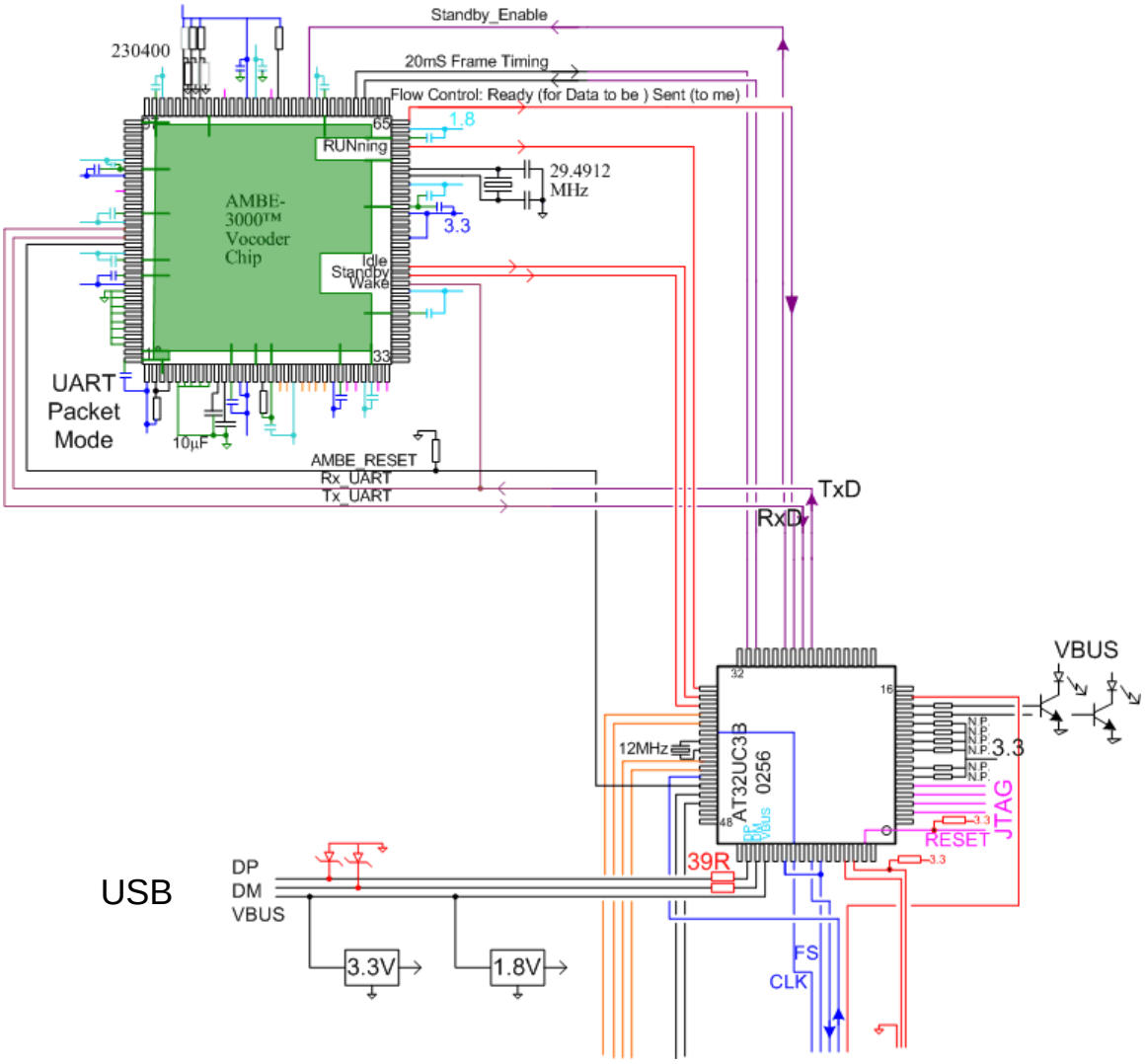


Figure 4: Dongle Hardware

The AMBE-3000™ Vocoder Chip supplied by DVSI is the key hardware component in the Dongle. The chip implements one channel the proprietary voice compression and expansion along with forward error correction and noise suppression, with a relatively easy to use API. Documentation for the chip from DVSI is relatively good. *AMBE-3000™ Vocoder Chip Users Manual* Version 2.2, May, 2010. [http://www.dvsinc.com/brochures/literature.htm].

In principle, one could interface this chip directly to a PC through RS-232 level shifters. In practice, support for serial port communication hardware in PC's is deprecated. The high serial data rate [>230Kbps for one voice channel] is also somewhat problematic. A higher data rate such as USB 12Mbps full-rate, or greater, is preferred.

Strictly motivated by familiarity, we chose to implement the USB 2.0 Full Speed interface using the same Atmel AT32UC3B part used on the Generic Option Board (G.O.B.). This provided some flexibility in

power-up and control sequencing and packet buffering. The capability was there, (though never actually implemented), to use the faster parallel interface into the AMBE-3000™, or attach directly to hardware audio A/D and D/A converters, avoiding the PC soundcard API's.

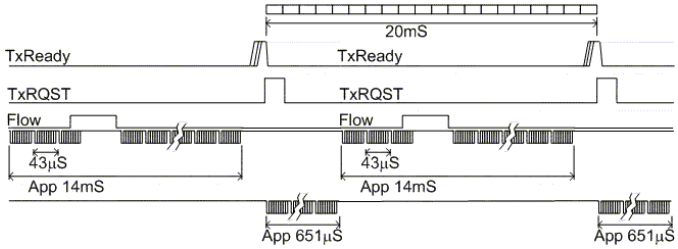


Figure 5: AMBE-3000 to AVR Serial Communication

Using the USART Dongle hardware shown in Figure 5, the critical timing occurs during a voice transfer between the AMBE-3000 chip and the AVR processor. Within a 20mS voice frame, 320 bytes (plus overhead) of PCM data must flow in one direction taking up about 70% of the available time. [During the same frame the 49 *bits* of returned AMBE encoded voice only uses about 3% of the frame.]

Windows Packet arrival jitter is mitigated in the Dongle by processing through in and out circular buffers. This assures that any new packet arriving early does not corrupt any previous packet being processed. Loading and unloading of these circular buffers is facilitated in the AVR by hardware DMA controllers.

The Dongle firmware is loosely based on the Atmel framework USB CDC example. Upon power-up there is some brief hardware initialization, followed by initialization of application states and memory buffers. The high-level (enumeration) USB initialization uses the standard Atmel framework call, *usb_task_init()*. The framework *device_cdc_task_init()* has been slightly modified, removing the local disable/re-enable interrupts and moving this to a common location.

Initialization and use of the DMA controllers was explained somewhat in the earlier paper, "A Software Example using the Generic Option Board," M.H.Retzer, 2009.

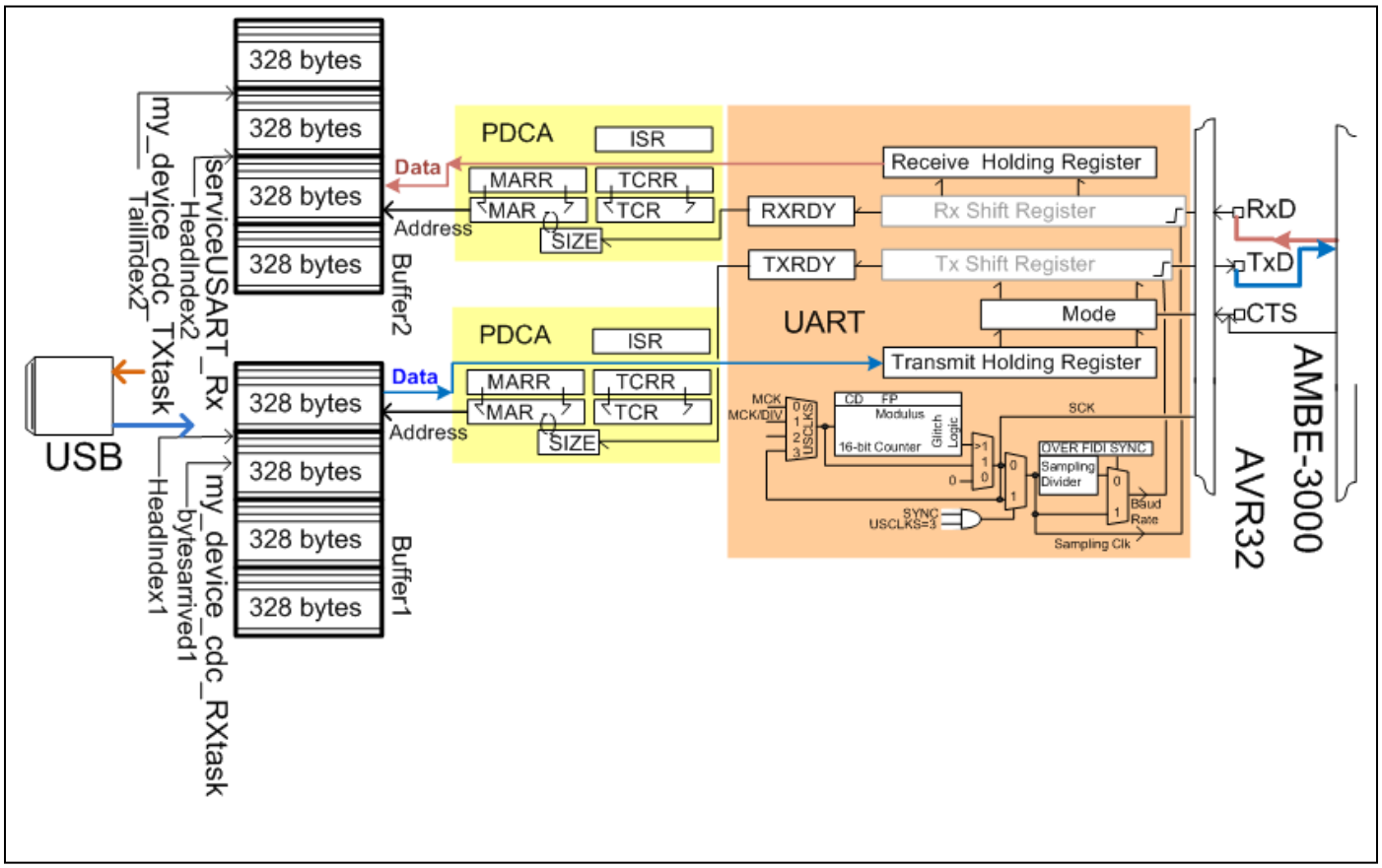


Figure 6: AVR USART DMR Hardware

Once initialized, the Dongle software runs in an infinite loop:

```
while (TRUE)
  {
        usb_task();

        //State Dependent Stuff.
        switch (Dongle_state){
        case WAITINGFORENUM:
           …
           break;
        case WAKINGDVSICHIP:
           …
           break;
        case INITDVSICHIP:
           …
           break;
        case RUNNING:
           serviceUSART_Rx();
           my_device_cdc_TXtask();
           break;
        }

        my_device_cdc_RXtask();
  }
```

The *usb_task()* is from the standard Atmel framework. *my_device_cdc_RXtask()* is polled every loop to handle USB reception. This is modified from the Atmel framework to pass packets efficiently over USB. The standard framework sends/receives only one byte at a time. The modified routine fills entire 64 byte blocks.

Within the state dependent section is code to wake up the AMBE-3000 chip, and to monitor for serial messages from the AMBE-3000 chip, and echo these messages on the USB.

my_device_CDC_RXtask() is always polled (in order to catch messages from the PC).
my_device_CDC_TXtask() is only called once the AMBE-3000 chip is enabled.

serviceUSART_Rx() is a polling routine for delimiting messages from the AMBE-3000 chip. If the USART is receiving a message from the AMBE-3000 chip, the AVR32_USART_CSR_TIMEOUT_MASK in the control and status register will be asserted whenever there is greater than a five character gap following a sequence of characters. This gap indicates the end of a message.

```
void serviceUSART_Rx(void)
{
  if (0 != ( ((&AVR32_USART1)->csr) &
               AVR32_USART_CSR_TIMEOUT_MASK)){
     FlagEvent(2,
               HeadIndex2,
               Buffer2[HeadIndex2].RAW_Words[0]);
     armNextRxBuffer();
  }
}
```

Upon detecting a gap, *serviceUSART_Rx()* calls *armNextRxBuffer()* to point the DMA transfer to the next free buffer. HeadIndex2 is then advanced. When *my_device_cdc_TXtask()* is next called, the mismatch in HeadIndex2 and TailIndex2 will parse the new message and send it out over USB. *depleteTxBuffer()* is called to advance TailIndex2.

my_device_cdc_RXtask() reads in a message from USB, parses it until there is an entire message, and points the USART Tx DMA buffer to automatically echo to theAMBE-3000.

III. PC Application

The IPSC terminal PC application is highly evolved; meaning that it may have started out as an intelligent design, but was patched and fixed over a number of months as new problems were discovered. The surviving changes are what remain today. Still, it provides a useful *working* structure for a first pass. Little effort should be needed to clean it up, or avoid any of its present difficulties.

We had a few ground-rules that determined some of the choices. We use *Windows* computers here, so this was to be a Windows application. Writing to some other OS, with its own unique network, sound, and USB interfaces, may have provided a *different* experience, but not a more obviously relevant one. Writing to some OS-agnostic language, such as Java, may have provided some tangible benefits, but a quick survey of OS *dependent* sound and USB services available for Java caused Fear, Uncertainty, and Doubt. With Windows, we were also constrained to use *old* Windows. We, and many of our 3rd party developers, have not yet upgraded from Win XP. We also had considerable F.U.D. about using the *.Net* environment, in particular *because* we did not want to get tied too closely with some particular OS implementation. So, we stayed pretty much with antique MFC, which has been around at least long enough for most folks to understand. We did not *want* a slick polished user interface; we wanted the IPSC protocols executed correctly, and reasonable sounding audio. There are significant opportunities remaining for 3rd party developers.

The application is dialog based, with the main dialog providing a rudimentary user interface. User operations are limited to initiating connection to the IPSC network, selecting a Talkgroup (and associated TDMA slot), monitoring call events on the IPSC network, pressing and releasing PTT to initiate a Talkgroup call on the network, and monitoring network debugging/diagnostic messages.

The main user interface dialog owns three major classes:

CDirectSound — for audio routing and controlling the PC soundcard. The multimedia sound sampling timer used here also functions as the timebase to schedule serial and network transactions.

CSerialDongle — for USB serial port emulation to and from the Dongle.

CIPSCNet — for maintaining the IP network connection and running the IPSC protocols.

CIPSCNet also maintains two database classes used during a registration session. Presently, this information is not permanently stored.

CIPSCPeer — for maintaining a directory of discovered IPSC peers, and scheduling protocol transactions maintaining the connections to the peers.

CIPSCCall — for maintaining a network call event history, and communicating network events to the user interface.

Each of the three major classes controls one or more worker threads. These worker threads provide autonomous operation for the real-time critical processes, with (minimal) slow-down to each other or to the user interface.

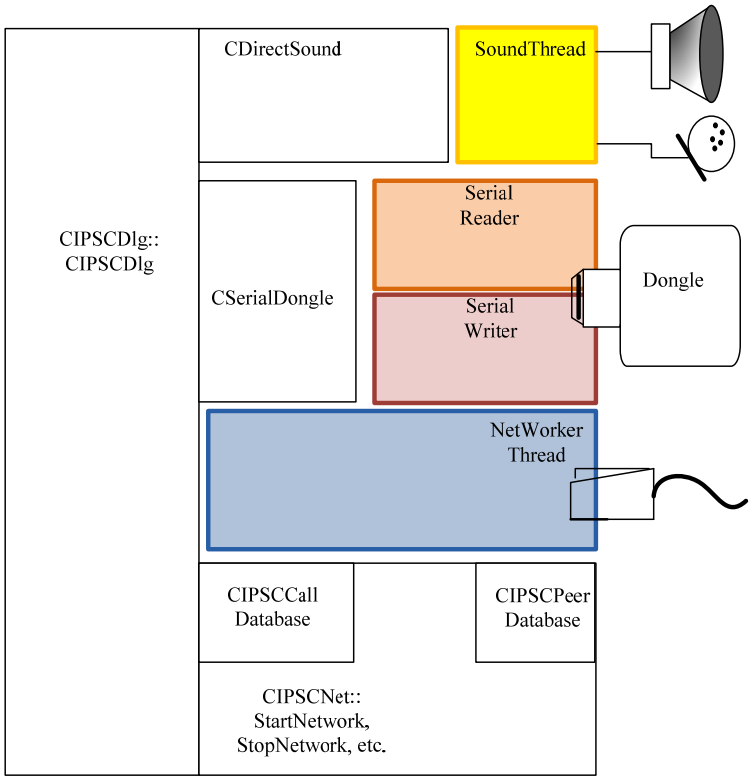


Figure 7: IPSC Terminal Architecture

Once started, the CIPSCNet class establishes connection with the Master repeater, discovers other peers on the network, maintains connection with the Master, and (in the general case) with each of the discovered peers. This is implemented using a state machine. Network and error events are communicated back to the user dialog using normal Windows messaging. The user dialog may then use public methods in CIPSCNet to retrieve peer or call database entries, or to set the current Talkgroup. Most of the knowledge about the IPSC protocols is encapsulated within CIPSCNet.

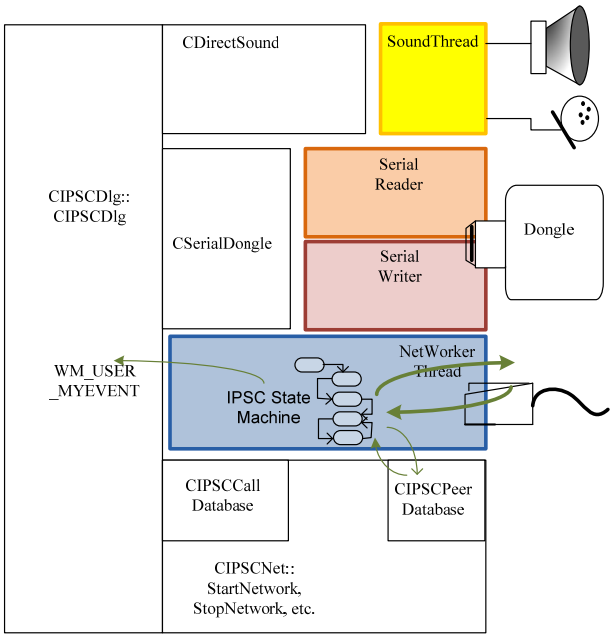


Figure 8: Join the IPSC System

All incoming voice call events are recorded in a finite length revolving CIPSCCall database. Incoming events are also signaled to the user dialog. The user dialog may retrieve the associated database information, and display this information or use it for call control [to force a de-key of the local terminal]. When a call comes in on the pre-selected Talkgroup, CIPSCNet extracts the AMBE encoded voice packets and begins filling a circular buffer queue for sending to the Dongle.

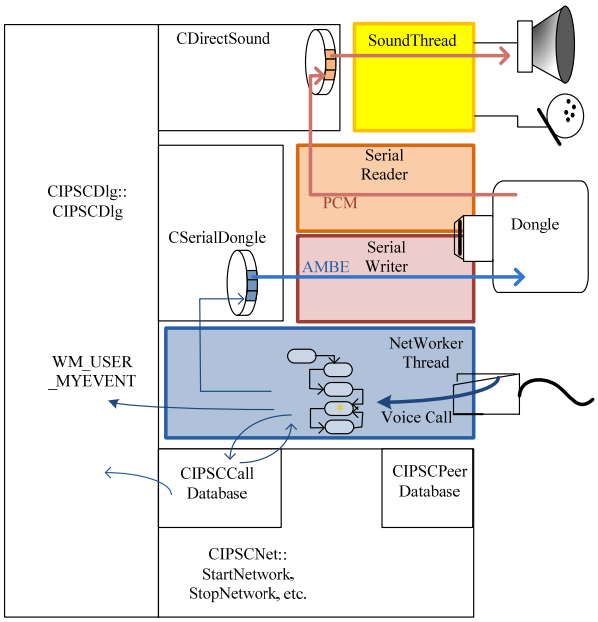


Figure 9: Receiving a Voice Call

Every 20mS, [based on events in the mostly accurate multimedia sound thread], the Dongle send queue is polled. If a new AMBE voice packet is available from the network, the Serial Writer worker thread will send it out the USB link to the Dongle.

Whenever the Dongle receives an AMBE packet, it (eventually) returns a PCM voice packet. The Serial Reader worker thread queues this PCM to a software buffer maintained by CDirectSound. The Sound worker thread periodically feeds the Microsoft sound API's with this stream to the speaker. At the start of a new call reception several PCM samples are buffered into the queue prior to starting the stream to the speaker. This effectively *jitter buffers* the arriving network packets and mitigates small errors between the Repeater and local clocks. Eventually, the depletion of this queue terminates the speaker audio. The number of pre-buffered samples is one of the critical (presently hard-coded) parameters affecting voice reception delay. More effective jitter and lost packet mitigation here could yield better performance.

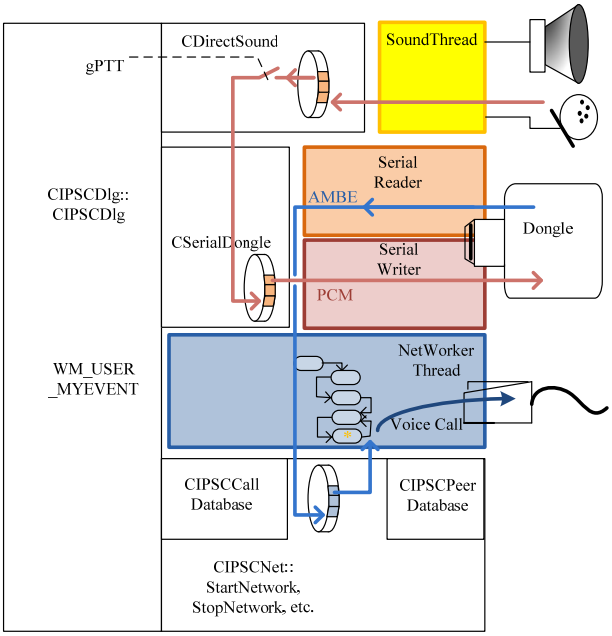


Figure 10: Transmitting a Voice Call

The Sound worker thread is continuously collecting PCM voice packets from the microphone at a (more or less) constant 20mS rate. When the terminal application is idle these packets are simply discarded, [and other periodic tasks are run]. During these periodic tasks the Sound worker thread may detect that the user dialog is requesting transmission on the pre-selected Talkgroup. The worker thread then begins to queue PCM packets to the Dongle. A call is also made to a public method in CIPSCNet to initialize the parameters for a new call.

AMBE packets returned from the Dongle are assembled into a 60mS IPSC network frame and queued for transmission. During this assembly the correct Header frames are scheduled, and the *A,B,C,D,E,F* Turbo airlink formatting is applied.

At any time the user dialog may un-assert its request for PTT. This may happen as a result of a user request, or may automatically happen in response to a higher priority network call event. Terminating a call must happen in an ordered sequence. Upon detecting loss of PTT the Sound thread will continue feeding the Dongle with silence samples while monitoring the network transmission status. This loss of PTT will also be detected by the network packet assembler. The packet assembler will then continue to schedule transmissions, perhaps including silence, until the entire *A,B,C,D,E,F* Turbo airlink superframe is completed. Then it will schedule the voice Terminator transmission, and un-assert its transmission status. The Sound thread then stops feeding silence to the Dongle.

Once voice frames are scheduled into the network transmit queue they are handled autonomously by the network worker thread. In the general case, each network voice frame must be sent to every peer discovered in the CIPSCPeer database. The hooks for this are present in the state machine, however we have not as yet implemented replication to multiple peers.

Operation of an IPSC Terminal requires some *a priori* information, such as the IP Address and port number of the Master peer, and a *pseudo*-IndividualRadioID for the terminal. There are also some long-term settings like COMM Port or preferred sound card. These may be entered by the user, however we felt it more convenient to provision this information through an external text *ini* file. Any real application would likely store other information, such as a mapping between Talkgroup ID's and Names, TDMA Slots, call logs, etc. The exact mechanism for this would be highly application dependent.

IV. IPSC STATE MACHINE

A state transition diagram of the IPSC protocol is shown on the next page. Details of this protocol may be found in *MOTOTRBO™ IP Site Connect Interface Protocol Specification* , [Numerous versions, but I used mostly Version 01.01 November 19, 2009]. This state machine is implemented within the worker thread of CIPSCNet. Three event objects and a timeout are defined to wake this thread from the `WSAWaitForMultipleEvents`:

`m_EventArray[TICKLEWORKERINDEX],` used by public handlers within CIPSCNet, (but called from different threads), to wake the NetWorkerThread. Used to inform NetWorkerThread that a voice frame is ready to be sent, or to request a graceful shutdown of the thread.

`m_EventArray[TXCOMPLETEINDEX],` mapped into the Winsock I/O completion event. Asserted upon completion of a sending operation on the socket.

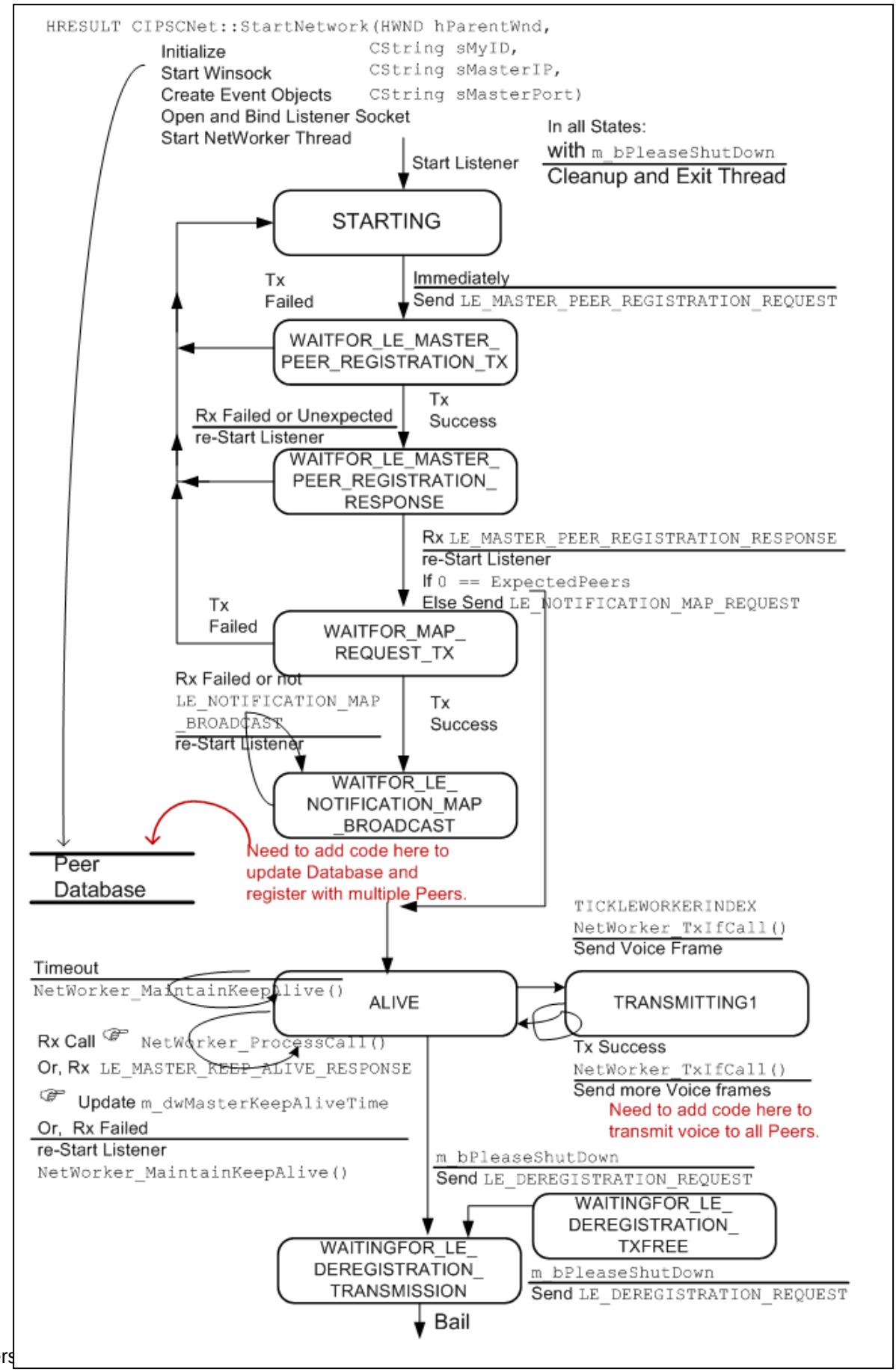
```
m_TxOverlapped.hEvent = m_EventArray[TXCOMPLETEINDEX];
```

`m_EventArray[RXAVAILABLEINDEX],` mapped into the Winsock I/O completion event. Asserted upon completion of a receive operation on the socket.

```
m_RxOverlapped.hEvent = m_EventArray[RXAVAILABLEINDEX];
```

Note that the timeout event is not the period of any keep-alive transmissions. The actual frequency of these transmissions is determined by the Windows *TickCount* and the Master (and multiple peer) database. It is only necessary for the thread to wake up sufficiently often, for any event, to check the database using the helper routine *NetWorker_MaintainKeepAlive.*

```
void CIPSCNet::NetWorkerThread()
{
  int    rc;
  DWORD  EventIndex = FIRSTPASS;

  //Code for starting Listener.
  rc = StartReceiver();

  while(TRUE)
  {
    switch (m_IPSCStatus){
    case STARTING:
      NetWorker_STARTING(EventIndex);
      break;   //End of case STARTING

    case WAITFOR_LE_MASTER_...:
       NetWorker_WAIT…(EventIndex);
       break;  //End of case WAITFOR_LE_MASTER…

    //… Other State dependent handlers.

    }

    EventIndex = WSAWaitForMultipleEvents(
                                  MAXEVENTS,
                              m_EventArray,
                                     FALSE,
                          NOACTIVITYTIMEOUT,
                                     FALSE);
    if (WSA_WAIT_FAILED == EventIndex){
       //Something really bad happened.
       rc = WSAGetLastError();
       return;
    }


    switch (EventIndex){
    case WSA_WAIT_IO_COMPLETION:
      //This should not occur.
      EventIndex = ILLEGAL;
      break;

    case WSA_WAIT_TIMEOUT:
      //No activity for long time.
      EventIndex = TIMEOUT;
      break;

    default:
      //Processes Event Object
      EventIndex -= WSA_WAIT_EVENT_0;
    }
  }
}
```

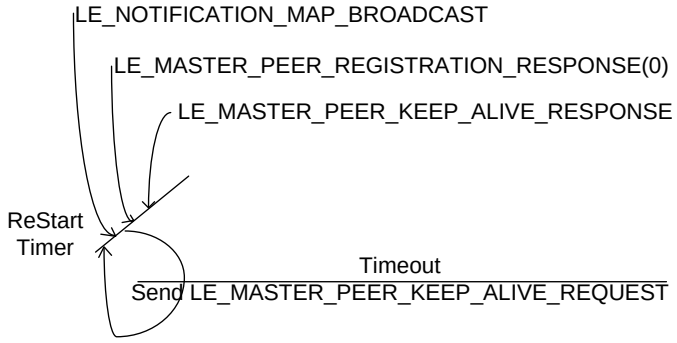

HRESULT CIPSCNet::StartNetwork(HWND hParentWnd,
CString sMyID,
CString sMasterIP,
CString sMasterPort)
Initialize
Start Winsock
Create Event Objects
Open and Bind Listener Socket
Start NetWorker Thread
Start Listener
In all States:
with m_bPleaseShutDown
Cleanup and Exit Thread
STARTING
Tx Failed
Immediately
Send LE_MASTER_PEER_REGISTRATION_REQUEST
WAITFOR_LE_MASTER_PEER_REGISTRATION_TX
Rx Failed or Unexpected
re-Start Listener
Tx Success
WAITFOR_LE_MASTER_PEER_REGISTRATION_RESPONSE
Rx LE_MASTER_PEER_REGISTRATION_RESPONSE
re-Start Listener
If 0 == ExpectedPeers
Else Send LE_NOTIFICATION_MAP_REQUEST
Tx Failed
WAITFOR_MAP_REQUEST_TX
Rx Failed or not LE_NOTIFICATION_MAP_BROADCAST
re-Start Listener
Tx Success
WAITFOR_LE_NOTIFICATION_MAP_BROADCAST
Peer Database
Need to add code here to update Database and register with multiple Peers.
TICKLEWORKERINDEX
NetWorker_TxIfCall()
Send Voice Frame
Timeout
NetWorker_MaintainKeepAlive()
ALIVE
TRANSMITTING1
Rx Call
NetWorker_ProcessCall()
Or, Rx LE_MASTER_KEEP_ALIVE_RESPONSE
Update m_dwMasterKeepAliveTime
Or, Rx Failed
re-Start Listener
NetWorker_MaintainKeepAlive()
Tx Success
NetWorker_TxIfCall()
Send more Voice frames
Need to add code here to transmit voice to all Peers.
m_bPleaseShutDown
Send LE_DEREGISTRATION_REQUEST
WAITINGFOR_LE_DEREGISTRATION_TXFREE
WAITINGFOR_LE_DEREGISTRATION_TRANSMISSION
m_bPleaseShutDown
Send LE_DEREGISTRATION_REQUEST
Bail

The interesting states are when the machine is ALIVE, ready to receive or start transmitting voice frames, or TRANSMITTING1, actively transmitting queued voice frames (to multiple peers). Each of the other states are transitory; used only in establishing or taking down IPSC connections with the Master or other peers.

The machine is relatively simple; sequentially performing the tasks of:

- Sending, or re-sending a Master peer registration request. This message is used to register with the Master Peer.
- Waiting for Winsock to successfully send the registration request.
- Waiting for a Master peer registration response, or retrying the request if no response is received. This response contains the current operating modes and supported services of the Master peer. The receiving peer uses this information to update its local database with information about the Master peer.
- If multiple peers are indicated in the above response, request a notification map. The peer map is only requested when updated information about the state of the linked peers is needed. [This demo does not completely implement the updating of this map as multiple peers dynamically enter and leave the network.]
- Waiting, if necessary, for successful Winsock transmission of the map request.
- Receiving a notification map broadcast and updating the local peer database. Note the System Map in the broadcast does not contain an entry for the Master peer.
- Monitoring the successfully connected IPSC network; responding to any received voice calls or requests to transmit a call. Periodically send keep-alive requests to the Master (and other peers).
- Attempting to gracefully shut down the IPSC network connection upon user termination by sending a deregistration request.

While ALIVE, in addition to handling call transactions the network state machine is responsible to maintaining connectivity to the Master, (and in the general case to all of the multiple peers). CIPSCNet keeps track of a keep-alive schedule variable, *e.g.* m_dwMasterKeepAliveTime. While ALIVE, upon any network reception or timeout event the *NetWorker_MaintainKeepAlive* routine is called to determine if any new keep-alive messages are needed.



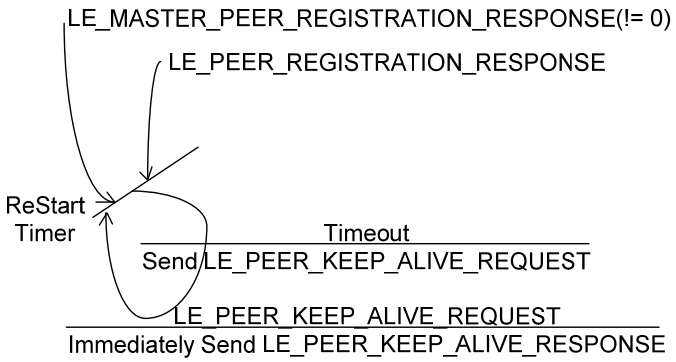


Figure 11: Conditions for Sending Keep-Alive Messages

The public methods used to interface with CIPSCNet are:

```
CIPSCNet();
virtual ~CIPSCNet();
HRESULT StartNetwork(HWND hParentWnd,
                     CString sMyID,
                   CString sMasterIP,
                 CString sMasterPort);
void StopNetwork(void);
void SetAuthenticationKey(CString theKey);
void SetMessagingDelayFloor(int theDelay);
void SetCallHangTimer(int theTime);
void SelectSingleTalkgroup(Cstring selgp,
                           int TxSlot);
unsigned __int32 GetSelectedTalkgroup(void);
void GetSelectedTalkgroup(char* pID3);
int  GetTxSlot(void);
void SetMyRadioID(unsigned __int32 MyID);
unsigned __int32 GetMyRadioID(void);
void GetMyRadioID(char* pID3);
unsigned __int32 GetMyPeerID(void);
void GetMyPeerID(char* pID4);
PSOCKADDR GetpPeerAddressByIndex(int Idx);
void NetStuffTxVoice(unsigned char pVoice);
void NetTx(bool Start);
```

Of these, *NetTx()* is used to initialize new voice calls, and *NetStuffTxVoice()* is used to queue network voice transmissions.

V. IPSC NETWORK VOICE PACKETS

UDP/IP is used as transport layer protocol for IPSC network messages. All IPSC messages have a one-byte opcode that specifies the type of the message, and a four-byte ID of the peer that is sending the message. The payload is entirely dependent on the opcode. Opcodes 0x80 through 0x84 contain voice call content, [0x80 is the code for an ordinary Group call].

Voice content frames [defined in Section 6.4 of *IP Site Interface Protocol Specification*] contain additional information at the beginning of the payload.

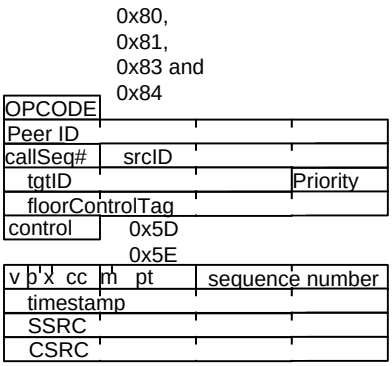


Figure 12: IPSC Header and Additional Voice Frame Information

There are six types of IPSC voice call content frames with a somewhat similar format: The Header and Terminator frames used at the start and end of a transmission, and the A,B,C,D,E, and F frames carrying the actual AMBE encoded voice samples. During a call the sender sources a frame nominally every 60mS. The A ,B, C, D, E, and F frames thus each carry three 20mS AMBE voice packets. The first 31 bytes of the Header, Terminator, and A, B, C, D, E, and F frames use a common format. Some of the fields may be pre-computed at the start of the call, with the Sequence Number, Time Stamp, PT [6.4.2 RTP Header], and Call Control Information [6.4.1.3.5 CallControlInformation] simply updated for each 60mS frame.

The various *uncommon* fields used in the Header, Terminator, and at the end of the A,B,C,D,E, and F frames also remain constant during a call, and may be pre-computed.

The type of frame is designated in the common RepeaterBurstDataType field.

RepeaterBurstDataType	Value
DATA_TYPE_VOICE_HEADER	0x01
DATA_TYPE_VOICE_TERMINATOR	0x02
DATA_TYPE_VOICE	0x0A

Fortunately, for many terminal *receive only* applications, such as scanners, loggers, or voice storage, it is not necessary to parse all of these *uncommon* fields. The AMBE encoded voice bits are packed consistently in the A, B, C, D, E, and F frames. In principle, a receiver need only look at the Opcode [first byte] to determine the call type, the tgtID [bytes 10-12] to determine the call destination, and the repeaterBurstDataType to determine start and end of a call.

Call reception in the demo application proceeds fairly simply:

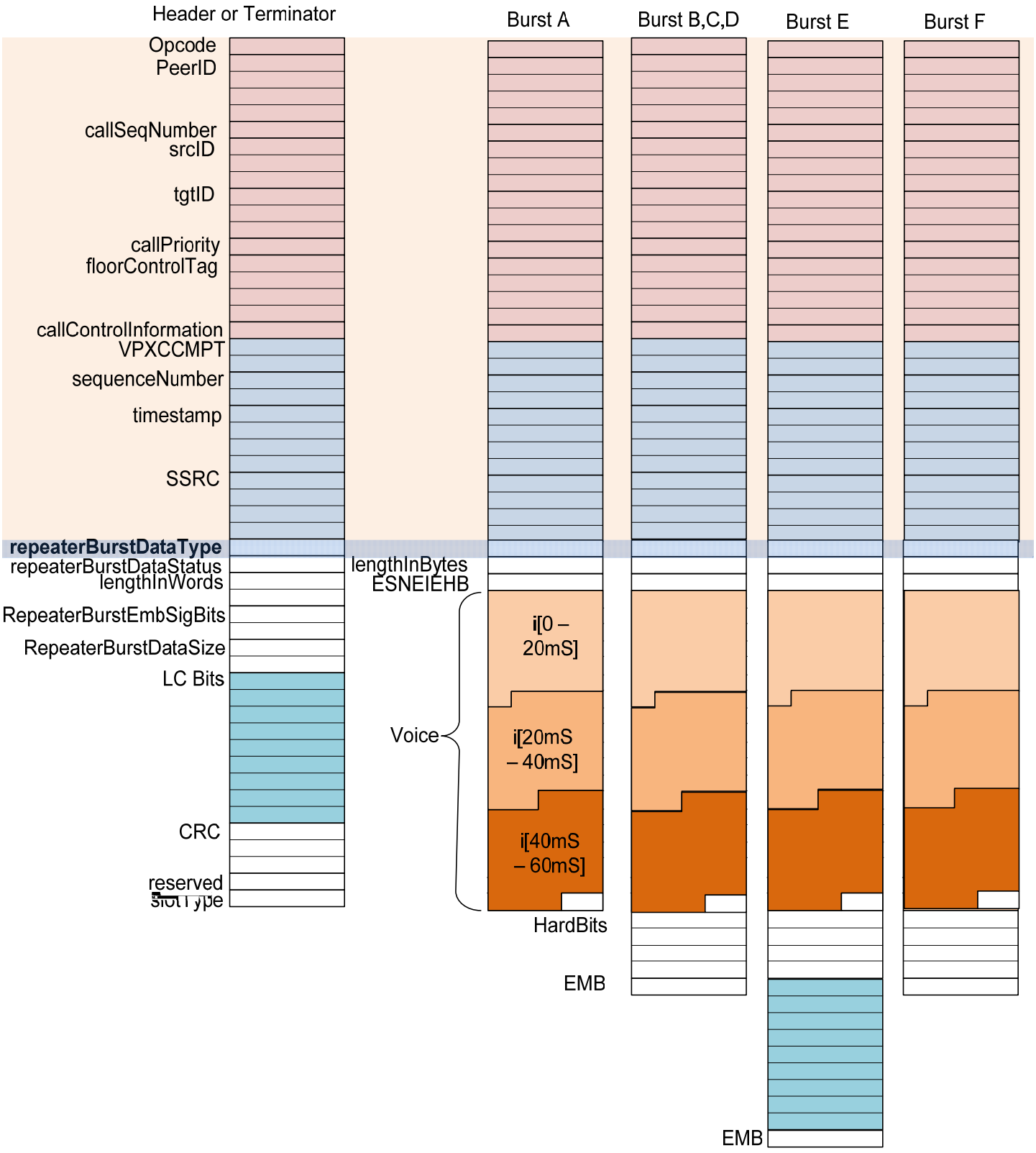
When IPSCNet state machine is ALIVE, and a UDP datagram is successfully received from the network, the code first parses the first byte [Opcode] field. At this point the call could be directed to *any* Talkgroup, and it may be the start of a new call or the continuation of one already in progress.

```
switch (*(m_CurrentRecvBuffer.buf))
{
case IPSC_PVT_VOICE_CALL:
case IPSC_GRP_DATA_CALL:
case IPSC_PVT_DATA_CALL:
case IPSC_GRP_VOICE_CALL:
     NetWorker_ProcessCall();
     break;
case LE_MASTER_KEEP_ALIVE_RESPONSE:
     m_dwMasterKeepAliveTime = GetTickCount();
     break;
//Following code for multi-peer implementation.
case LE_PEER_KEEP_ALIVE_REQUEST:
     break;
case LE_PEER_KEEP_ALIVE_RESPONSE:
     break;
case LE_NOTIFICATION_MAP_BROADCAST:
     break;
case LE_MASTER_PEER_REGISTRATION_RESPONSE:
     break;
default:
     break;
}
```

NetWorker_ProcessCall() extracts the call parameters from the incoming message, and queries the history database by calling *NetWorker_NewCall()*. This updates the call history database. Application dependent information may then be signaled to the user dialog. If the message contains voice, the tgtID matches the selected Talkgroup, and we are not already transmitting, *NetWorker_ExtractVoice()* is called.

Three, 20mS AMBE voice frames are carried in each IPSC network frame. *NetWorker_ExtractVoice()* reserves three AMBE buffers in the CSerialDongle circular store, and copies the bits from the network frame. [This operation is *not* pretty, as each of the three frames is bit-shifted differently.]

To receive a call, CIPSCNet is done until the next message comes in 60mS later. Processing of the received AMBE encoded voice is handled autonomously by the serial and sound worker threads.


Header or Terminator
Burst A
Burst B,C,D
Burst E
Burst F
Opcode
PeerID
callSeqNumber
srcID
tgtID
callPriority
floorControlTag
callControlInformation
VPXCCMPT
sequenceNumber
timestamp
SSRC
repeaterBurstDataType
repeaterBurstDataStatus
lengthInWords
RepeaterBurstEmbSigBits
RepeaterBurstDataSize
LC Bits
CRC
reserved
Slot Type
lengthInBytes
ESNEIEHB
Voice
i[0 – 20mS]
i[20mS – 40mS]
i[40mS – 60mS]
HardBits
EMB
EMB

Call transmission requires somewhat more effort in that all the fields in the Header, Terminator, and A,B,C,D,E, and F frames must be populated correctly. The frames must be also transmitted over the network in the correct order. [Also, what can sometimes be a very significant problem, the frames must be transmitted at a nominal 60mS average rate, with little jitter. Long pauses in the multimedia stream will drop the call.]

When the Sound worker thread detects that the user dialog is requesting PTT, it calls public method *NetTx(TRUE)*. This flags that the transmission process has begun, initializes the burst sequence and number of voice frames collected within the burst, and calls *InitialCallRecord()* to pre-compute the call fields. The Sound worker thread immediately begins queuing 20mS PCM voice samples to the Dongle. The Dongle will return more-or-less periodic 20mS AMBE frames.

```
void CIPSCNet::NetTx(bool Start)
{
  if (Start){ //New start of transmission.

               g_bTX           = TRUE;
               m_TxBurstType   = VOICEHEADER0;
               m_TxSubCount    = 0;
               InitialCallRecord();
               return;
  }else{

...
```

Whenever the Serial Reader worker thread receives a 20mS AMBE encoded frame from the Dongle it calls *NetStuffTxVoice()* to pack the network frame. These bits get packed into one of three fields depending on the m_TxSubCount value. [Again, this operation is not pretty, as each of the three fields is bit-shifted differently.] When all three frames are collected, *NetTx(FALSE)* is called.

NetTx(FALSE) is called whenever an on-going network voice transmission is in progress and there is another 60mS burst to send.

The Sound thread collects periodic 20mS blocks of PCM.

☞ Which get sent to the Dongle at a mostly uniform 20mS rate.

☞ Which get returned from the Dongle at a mostly uniform 20mS rate.

☞ Which get stuffed into a 60mS IPSC voice burst.

The simple state machine in *NetTx()* builds the correct transmission sequence: Starting with (several) copies of Header, the A,B,C,D,E, and F frames in rotating sequence, and completing an entire A,B,C,D,E, and F sequence prior to terminating the call, finally sending Terminator. During the call de-key, the Sound thread will continue to feed silence until `g_bTX` is released.

FillIPSCFormat() is called to fill in the burst dependent fields. The queue head-pointer is advanced (which will indicate to the CIPCSNet worker thread that a new frame is available for transmission. And the thread is tickled to wake it up.

```
  }else{
  //Depending on what was sent LAST time:
      switch (m_TxBurstType){
      case VOICEBURST_A:  //m_TxBurstType += 1;
         m_TxBurstType   = VOICEBURST_B;
         break;
      case VOICEBURST_B:
         m_TxBurstType   = VOICEBURST_C;
         break;
      case VOICEBURST_C:
         m_TxBurstType   = VOICEBURST_D;
         break;
      case VOICEBURST_D:
         m_TxBurstType   = VOICEBURST_E;
         break;
      case VOICEBURST_E:
         if (!g_bPTT){
             last  = TRUE;
         }
         m_TxBurstType   = VOICEBURST_F;
         break;
      case VOICEBURST_F:
         if (g_bPTT){
          //Start transmitting next superframe.
            m_TxBurstType   = VOICEBURST_A;
         }else{
        //Last superframe has been transmitted.
            m_TxBurstType   = VOICETERMINATOR;
            g_bTX           = FALSE;
         }
         break;
      case VOICEHEADER1:
           m_TxBurstType   = VOICEHEADER2;
           break;
      case VOICEHEADER2:
           m_TxBurstType   = VOICEBURST_A;
           break;
      case VOICEHEADER0:
           m_TxBurstType   = VOICEHEADER1;
           break;
      case VOICETERMINATOR:
 //Shouldn't get here, but if does, do nothing.
      case PREAMBLECBSK:
      default:
           return;
      }
  }

   FillIPSCFormat(&IPSCTxBuffer[m_TxHeadIndex],
                                 m_TxBurstType,
                                        last );

   m_TxHeadIndex = (m_TxHeadIndex + 1) &
                    MAXIPSCTXBUFFERSMASK;

  WSASetEvent(m_EventArray[TICKLEWORKERINDEX]);
}
```

Getting all of the fields correct in IPSC transmissions can be a challenge as this requires interpreting several different documents. A major goal of this terminal demonstration is to supplement rough documentation with working code. The code supplied should not be taken as efficient, hardened product-ready libraries, but rather as a reference to the correct implementation of the IPSC protocols. These fields are again briefly described here.

<table>
<tr><th>Field</th><th>Document Reference</th><th>Applies To:</th><th>Description</th></tr>
<tr><td>Opcode</td><td>6.4.1.3.1 of IP Site Interface Protocol Specification [IPSIPS]</td><td>Byte 0, common to all bursts.</td><td>This field indicates the type of the IP Site Connect Call message.
This is the 0x80, 0x81, 0x83, or 0x84 mentioned above.</td></tr>
<tr><td>peerID</td><td>6.2.7.6. [IPSIPS]</td><td>Byte 1-4, ctab</td><td>The ID of the peer where this packet originated. This is a unique ID for the terminal supplied by the IPSC network administrator.</td></tr>
<tr><td>callSeqNumber</td><td>6.4.1.3.2. [IPSIPS]</td><td>Byte 5, ctab.</td><td>The sequence number of this call. The sequence number is generated by the peer that sources the call and is incremented every time a new call originates. The number is set to zero when it crosses the maximum value of Uint8. Once a sequence number is assigned to a call, all the audio packets in that call will carry the same call sequence number. Here, a “call” refers to a single PTT attempt; continuing the conversation with subsequent PTT’s, or answering the conversation from another terminal will be a new call, with a new sequence number. Terminals independently assign sequence numbers.</td></tr>
<tr><td>srcID</td><td>6.2.7.8. [IPSIPS]</td><td>Byte 6-8, ctab.</td><td>The ID of the subscriber that initiated the call. This will be a unique number assigned to the terminal by the IPSC network administrator.</td></tr>
<tr><td>tgtID</td><td>6.2.7.8. [IPSIPS]</td><td>Byte 9-11, ctab.</td><td>The target ID. For a group call this is the ID of the Talkgroup.</td></tr>
<tr><td>callPriority</td><td>6.4.1.3.3 [IPSIPS]</td><td>Byte 12, ctab.</td><td>Specifies the priority of the call.
<table>
<tr><th>Call Priority Value</th><th>Allocation</th></tr>
<tr><td>0x00</td><td>RESERVED</td></tr>
<tr><td>0x01</td><td>Data</td></tr>
<tr><td>0x02</td><td>Voice</td></tr>
<tr><td>0x03</td><td>Emergency</td></tr>
<tr><td>0x04 to 0xFF</td><td>RESERVED</td></tr>
</table>
Floor Control shall allow emergency calls to interrupt and take over ongoing non-emergency calls. Data calls are not allowed to be taken over by any calls, including emergency.</td></tr>
<tr><td>floorControlTag</td><td>6.4.1.3.4. [IPSIPS] 3.2.1.1. IP Site Connect Development Guide[IPSCDG]</td><td>Byte 13-16, ctab.</td><td>The floor control tag in the IP Site Connect voice/data message header is a pseudo random number generated for each floor request.

At the beginning of a session in an IP Site Connect network, more than one transmission may start in a very short time interval, e.g. two subscribers at different sites initiate a call. Due to the nature of the Internet, it is not possible to ensure the peers receive the transmissions in the same order, which may cause different repeaters to select different transmissions to start their sessions. The IP Site Connect system can only support one call at a time in each WAC. To ensure all the peers are synchronized to the same call, each peer is required to implement the same call decision or floor arbitration algorithm before starting a session and during a session.

The rules involving Emergency callPriorities [above] are [mostly] clear. Rules for using the floorControlTag do not seem to be written down anywhere. The intent is that the random number provides some fairness during the contention. Best</td></tr>
</table>

<table>
<tr><th>Field</th><th>Document Reference</th><th>Applies To:</th><th>Description</th></tr>
<tr><td></td><td></td><td></td><td>guess is that smallest number wins.

This terminal demo has not fully implemented floor control. While our terminal is transmitting, the user dialog continues to receive event notifications of incoming call attempts. The user dialog can determine call priority, and choose to release the PTT request at any time.</td></tr>
<tr><td>callControlInformation</td><td>6.4.1.3.5. [IPSIPS]</td><td>Byte 17, ctab.</td><td>This field identifies the call control information for the active call session.
<table>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>secure</td><td>last</td><td>slot</td><td colspan="5">Reserved</td></tr>
</table>
The “last” bit indicates whether this packet is the last audio packet for this call or not. A zero value indicates that this is a regular packet and a one value indicates that this is the last packet. This field must be dynamically updated during the call. [This is the last packet containing the call Terminator.]</td></tr>
<tr><td>VPXCCMPT</td><td>6.4.2. [IPSIPS]</td><td>Byte 18-19, ctab.</td><td>0x805D All bursts except last.
0x805E Last burst.
This field must be dynamically updated during the call. [This is the last packet containing the call Terminator.]</td></tr>
<tr><td>sequenceNumber</td><td>6.4.2. [IPSIPS]</td><td>Byte 20-21, ctab.</td><td>The sequence number increments by one for each burst data packet sent, and may be used by the receiver to detect packet loss and to restore packet sequence. The initial value of the sequence number is random. This field must be dynamically updated during the call.</td></tr>
<tr><td>timestamp</td><td>6.4.2. [IPSIPS]</td><td>Byte 22-25, ctab.</td><td>The timestamp reflects the sampling instant of the first octet in the data packet. The sampling instant must be derived from a clock that increments monotonically and linearly in time to allow synchronization and jitter calculations. Units are in milliseconds. So in practice the terminal increments this number by 60 each burst. This field must be dynamically updated during the call.</td></tr>
<tr><td>SSRC</td><td>6.4.2. [IPSIPS]</td><td>Byte 26-29, ctab.</td><td>Always 0x0000.</td></tr>
<tr><td>repeaterBurstDataType</td><td>6.4.3.1.5.2. [IPSIPS]</td><td>Byte 30, ctab.</td><td>This field specifies the burst data type. This is the 0x01, 0x02, or 0x0A mentioned above. Thisfield is updated depending on Header, Voice, or Terminator.

Note that Voice bursts A,B,C,D,E, or F use Bit 7 of this field for slot number. 6.4.3.1.5.1. [IPSIPS]</td></tr>
<tr><td>repeaterBurstDataStatus</td><td>6.4.3.2.8.2. [IPSIPS]</td><td>Byte 31 in Header or Terminator.</td><td>Represents the status of the current RepeaterBurstData.
<table>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>slot</td><td>RSSI</td><td colspan="3">Reserved 000</td><td>RS Parity</td><td>CRC Parity</td><td>Embedded LC Parity</td></tr>
</table>
Terminal should set RSSI to 0.
Terminal should set RS Parity to 0.
Terminal should set CRC Parity to 0.
Terminal should set Embedded LC Parity to 0.</td></tr>
</table>

<table>
<tr><th>Field</th><th>Document Reference</th><th>Applies To:</th><th>Description</th></tr>
<tr><td>lengthInWords</td><td>6.4.3.2.8.3. [IPSIPS]</td><td>Byte 32-33 in Header or Terminator.</td><td>The length indicates the number of 16-bit words of the entire message frame that follows this field. It does not include the RepeaterBurstDataType field, the RepeaterBurstDataStatus field and the length field itself.</td></tr>
<tr><td>repeaterBurstEmbSigBits</td><td>6.4.3.2.8.4. [IPSIPS]</td><td>Byte 34-35 in Header or Terminator.</td><td>
<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td></tr>
<tr><td>RSSI Status</td><td colspan="2">Reserved 00</td><td>Burst Source</td><td colspan="4">Reserved 0000</td></tr>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>Reserved 0</td><td>Hard Bits Present</td><td colspan="2">Reserved 00</td><td>Slot Type</td><td>Reserved 0</td><td colspan="2">Sync</td></tr>
</table>
Terminal should set RSSI Status: 0
Terminal should set Burst Source: 0
Terminal should set Hard Bits Present to: 0
Terminal should set Slot Type to : 1 (present)
Terminal should set Sync to “10”.</td></tr>
<tr><td>repeaterBurstDataSize</td><td>6.4.3.2.8.5. [IPSIPS]</td><td>Byte 36-37 in Header or Terminator.</td><td>This field represents the number of bits in the burst data.
Terminal should set this to 96 for Header/Terminator.</td></tr>
<tr><td>LC Bits</td><td>ETSI TS 102 361-2 V1.2.6 (2007-12) 7.1.1.1.</td><td>Byte 38-46 in Header or Terminator.

Byte 56-64 of Voice Burst E.</td><td>
<table>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>PF</td><td>Reserved</td><td colspan="6">FLCO = 000000</td></tr>
<tr><td colspan="8">FID = 00000000</td></tr>
<tr><td>Emergency</td><td>Privacy 0</td><td colspan="2">Reserved 00</td><td>Broadcast</td><td>OVCM</td><td colspan="2">Priority</td></tr>
<tr><td colspan="8">tgtID</td></tr>
<tr><td colspan="8">srcID</td></tr>
</table>
Terminal should set PF to 0.
Terminal should set Broadcast to 0.
Terminal should set OVCM to 0.</td></tr>
<tr><td>CRC</td><td></td><td>Byte 47-49 in Header or Terminator.</td><td>If the application Set the “Burst Source” bit in RepeaterBurstEmbSigBits to 1 (Repeater generated), Repeater calculates the 24-bit RS CRC and also applies the appropriate mask for the Voice Header and the Voice Terminator. While using this alternative, the application populates the 24-bit CRC field in RTP payload with ZEROs.</td></tr>
<tr><td>Reserved</td><td></td><td>Byte 50 in Header or Terminator</td><td>00000000</td></tr>
<tr><td>SlotType</td><td>6.4.3.2.8.6. [IPSIPS]</td><td>Byte 51 in Header or Terminator.</td><td>
<table>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="4">Color Code</td><td colspan="4">Data Type
Voice Header = 0001
Voice Terminator = 0010</td></tr>
</table>
Terminal should set Color Code to “0000”.</td></tr>
<tr><td>lengthInBytes</td><td>6.4.3.1.5.3. [IPSIPS]</td><td>Byte 31 of A,B,C,D,E,F</td><td>The length indicates the number of bytes of the entire message frame that follows this field. It does not include the length field</td></tr>
</table>

<table>
<tr><th>Field</th><th>Document Reference</th><th>Applies To:</th><th>Description</th></tr>
<tr><td></td><td></td><td>Voice burst.</td><td>itself. For example, for Burst A, the length is 20 bytes.</td></tr>
<tr><td>ESNEIEHB</td><td>6.4.3.1.5.4. [IPSIPS]</td><td>Byte 32 of A,B,C,D,E,F Voice burst.</td><td>
<table>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>Embedded LC Parity</td><td>Sync</td><td>NULL LC</td><td>72 Bit EMB LC</td><td>Ignore Sig Bits</td><td>EMB</td><td>EMB LC Hard Bits</td><td>Bad Voice Burst</td></tr>
</table>
Terminal should set Embedded LC Parity to : 0 (Success)
Terminal should set Sync to 1 for Burst A, set to 0 for Burst B-F
Terminal should set NULL LC to 0
Terminal should set 72 Bit EMB LC to 1 for Burst E, set to 0 for Burst A, B, C, D, F
Terminal should set Ignore Sig Bits to 0
Terminal should set EMB to 0 for Burst A, set to 1 for Burst B, C, D, E, F
Terminal should set EMB LC Hard Bits to 0 for Burst A, set to 1 for Burst B, C, D, E, F

Note that Bit 0 is actually the Bad Voice Burst flag of the following voice frame.</td></tr>
<tr><td>Voice Payload</td><td>6.4.3.1. [IPSIPS]</td><td>Byte 33-51of A,B,C,D,E,F Voice burst.</td><td>Each voice superframe contains six bursts: A, B, C, D, E and F. And each burst contains three 20ms vocoder compressed frames. Each 20mS frame consists of a Bad Voice Burst flag (Terminal always sets to Zero), followed be 49 packed AMBE bits. The last three bits of Byte 51 are always set to Zero.</td></tr>
<tr><td>EMB</td><td></td><td>Byte 56 of B,C,D,F Voice Bursts.

Byte 65 of E Voice Burst.</td><td>
<table>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>Reserved 0</td><td colspan="3">CC</td><td></td><td>EEEI</td><td colspan="2">LCSS</td></tr>
</table>
Terminal should set CC to : 0
Terminal should set EEEI to : 0
Terminal should set LCSS to : 0
The MOTOTRBO repeater fills the EMB field when sending the bursts over the air</td></tr>
</table>

VI. Mapping IPSC Voice Bits to Dongle Frame

Forty-nine AMBE encoded voice bits fill a 20mS frame. Unfortunately, for reasons that may be known only to DVSI, they chose to use a different bit ordering in the AMBE-3000™ chip, (from their usual strategy of Most Important == Most Significant).

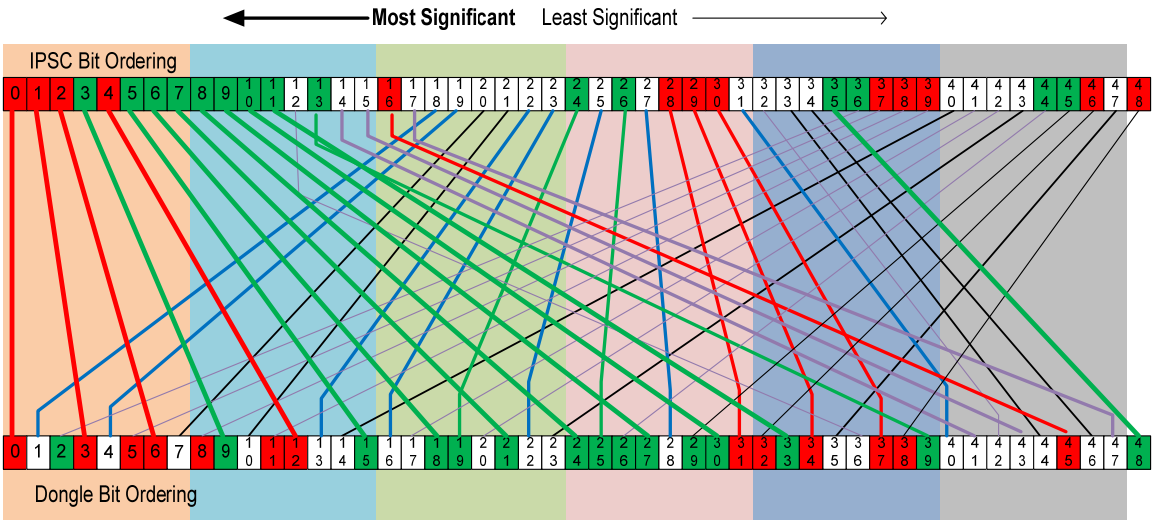


Figure 13: Re-Ordered AMBE Bits

What this means is that if we blindly shift bits between IPSC and Dongle frames, as any reasonable person would try, we will not get recovered voice PCM.

The routine *deObfuscate()* is used to convert between the two bit orderings. Each forty-nine bit AMBE frame received over the IPSC network is first left-justified into a byte array. Then *deObfuscate(IPSCTODONGLE)* is called to return the left-justified byte array expected by the Dongle.

During transmission, each forty-nine bit left-justified byte array coming from the Dongle is passed to *deObfuscate(DONGLETOIPSC)*. The returned left-justified array is then shifted into one of the three voice fields in the IPSC burst. The *deObfuscate()* routine is not particularly elegant, using using a 49 element sorting array.

```
const int IPSCTODONGLETABLE[49] =
//0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10,
11, 12. 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34,
35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46,
47, 48
{ 0,  3,  6,  9, 12, 15, 18, 21, 24, 27, 30,
33, 36, 39, 41, 43, 45, 47,  1,  4,  7, 10, 13,
16, 19, 22, 25, 28, 31, 34, 37, 40, 42, 44, 46,
48,  2,  5,  8, 11, 14, 17, 20, 23, 26, 29, 32,
35, 38 };

const int DONGLETOIPSCTABLE[49] =
//0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10,
11, 12. 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34,
35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46,
47, 48
{ 0, 18, 36,  1, 19, 37,  2, 20, 38,  3, 21,
39,  4, 22, 40,  5, 23, 41,  6, 24, 42,  7, 25,
43,  8, 26, 44,  9, 27, 45, 10, 28, 46, 11, 29,
47, 12, 30, 48, 13, 31, 14, 32, 15, 33, 16, 34,
17, 35 };
```

VII. CALCULATING THE HARD BITS

Burst B, C, D, and E voice bursts carry a four byte field called “LC Hard Bits.” These bits are *Hard* because an IPSC terminal is required to include them, and there is almost no documentation explaining how to do so. These bits are exactly the same information contained in the nine byte LC field, with the addition of error correction, distributed over four of the six voice bursts. [The A-burst never has Hard bits. F-burst has the 4-byte “LC Hard Bits”, but is always set to 0.] One can learn about these by going directly to the DMR standard [ETSI TS 102 361-1 V1.4.5 (2007-12)].

Given the nine LC bytes, the first step is to calculate a five-bit checksum. The calculation for the 5-bit CS is given by formula (B.23) in ETSI TS 102 361-1 V1.4.5 (2007-12) B3.11. A short algorithm is provided below:

```
//Five bit checksum calculation.

// b71 b70 b69 b68 b67 b66 b65 b64
// b63 b62 b61 b60 b59 b58 b57 b56
// b55 b54 b53 b52 b51 b50 b49 b48
// b47 b46 b45 b44 b43 b42 b41 b40
// b39 b38 b37 b36 b35 b34 b33 b32
// b31 b30 b29 b28 b27 b26 b25 b24
// b23 b22 b21 b20 b19 b18 b17 b16
// b15 b14 b13 b12 b11 b10 b9  b8
// b7  b6  b5  b4  b3  b2  b1  b0
const uchar CSUMMASK = 0x1F;
uint16_t checkSumGen( uchar * inputPtr )
{
        int      i;
        uint16_t generatedCrc=0;

        for(i=0;i<9;++i)
        {
                generatedCrc += inputPtr[i];
        }

        //Yes, it really is Mod 31.
        while (generatedCrc > 31){
                generatedCrc -= 31;
        }
        return generatedCrc;
}
```

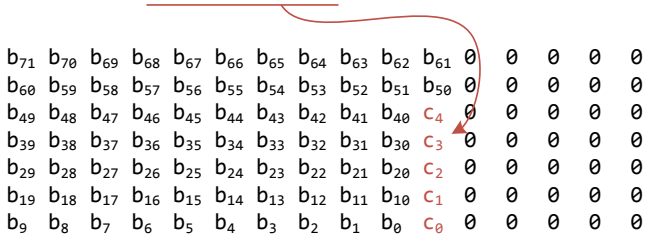
The next step is to re-order the LC bytes into 16-bit rows, stuffing in the five-bit checksum as shown below.

G

$b_{71}\ b_{70}\ b_{69}\ b_{68}\ b_{67}\ b_{66}\ b_{65}\ b_{64}$
$b_{63}\ b_{62}\ b_{61}\ b_{60}\ b_{59}\ b_{58}\ b_{57}\ b_{56}$
$b_{55}\ b_{54}\ b_{53}\ b_{52}\ b_{51}\ b_{50}\ b_{49}\ b_{48}$
$b_{47}\ b_{46}\ b_{45}\ b_{44}\ b_{43}\ b_{42}\ b_{41}\ b_{40}$
$b_{39}\ b_{38}\ b_{37}\ b_{36}\ b_{35}\ b_{34}\ b_{33}\ b_{32}$
$b_{31}\ b_{30}\ b_{29}\ b_{28}\ b_{27}\ b_{26}\ b_{25}\ b_{24}$
$b_{23}\ b_{22}\ b_{21}\ b_{20}\ b_{19}\ b_{18}\ b_{17}\ b_{16}$
$b_{15}\ b_{14}\ b_{13}\ b_{12}\ b_{11}\ b_{10}\ b_{9}\ b_{8}$
$b_{7}\ b_{6}\ b_{5}\ b_{4}\ b_{3}\ b_{2}\ b_{1}\ b_{0}$

Calculate 5 CSUM bits.

$0\ 0\ 0\ c_4\ c_3\ c_2\ c_1\ c_0$



$b_{71}\ b_{70}\ b_{69}\ b_{68}\ b_{67}\ b_{66}\ b_{65}\ b_{64}\ b_{63}\ b_{62}\ b_{61}\ 0\ 0\ 0\ 0\ 0$
$b_{60}\ b_{59}\ b_{58}\ b_{57}\ b_{56}\ b_{55}\ b_{54}\ b_{53}\ b_{52}\ b_{51}\ b_{50}\ 0\ 0\ 0\ 0\ 0$
$b_{49}\ b_{48}\ b_{47}\ b_{46}\ b_{45}\ b_{44}\ b_{43}\ b_{42}\ b_{41}\ b_{40}\ c_4\ 0\ 0\ 0\ 0\ 0$
$b_{39}\ b_{38}\ b_{37}\ b_{36}\ b_{35}\ b_{34}\ b_{33}\ b_{32}\ b_{31}\ b_{30}\ c_3\ 0\ 0\ 0\ 0\ 0$
$b_{29}\ b_{28}\ b_{27}\ b_{26}\ b_{25}\ b_{24}\ b_{23}\ b_{22}\ b_{21}\ b_{20}\ c_2\ 0\ 0\ 0\ 0\ 0$
$b_{19}\ b_{18}\ b_{17}\ b_{16}\ b_{15}\ b_{14}\ b_{13}\ b_{12}\ b_{11}\ b_{10}\ c_1\ 0\ 0\ 0\ 0\ 0$
$b_{9}\ b_{8}\ b_{7}\ b_{6}\ b_{5}\ b_{4}\ b_{3}\ b_{2}\ b_{1}\ b_{0}\ c_0\ 0\ 0\ 0\ 0\ 0$

The embedded signaling is protected using a BPTC consisting of Hamming (16,11,4) row code[ETSI TS 102 361-1 V1.4.5 (2007-12) B2.1].

The generator matrices for the (16,11,4) Hamming code is derived from the (15,11,3) primitive Hamming code. The generator polynomial for the primitive code is as follows:

$G(x) = x^4 + x + 1 = 23_8$

The generator matrices are given in table B.16.

Table B.16: Hamming (16,11,4) generator matrix

1	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1
0	1	0	0	0	0	0	0	0	0	0	1	1	0	1	0
0	0	1	0	0	0	0	0	0	0	0	1	1	1	1	1
0	0	0	1	0	0	0	0	0	0	0	1	1	1	0	0
0	0	0	0	1	0	0	0	0	0	0	0	1	1	1	0
0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1
0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	1
0	0	0	0	0	0	0	1	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	0	1	0	0	1	1	0	0	1
0	0	0	0	0	0	0	0	0	1	0	0	1	1	0	1
0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	1

This generator matrix (seems to be) derived by adding an (even) parity check column:

$=$
1 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1
0 1 0 0 0 0 0 0 0 0 0 1 1 0 1 0
0 0 1 0 0 0 0 0 0 0 0 1 1 1 1 1
0 0 0 1 0 0 0 0 0 0 0 1 1 1 0 0
0 0 0 0 1 0 0 0 0 0 0 0 1 1 1 0
0 0 0 0 0 1 0 0 0 0 0 1 0 1 0 1
0 0 0 0 0 0 1 0 0 0 0 0 1 0 1 1
0 0 0 0 0 0 0 1 0 0 0 1 0 1 1 0
0 0 0 0 0 0 0 0 1 0 0 1 1 0 0 1
0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 1
0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 1

Given an 11-bit input Ä = $[b_{10}\ b_9\ b_8\ b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0]$, the 16-bit codeword,
Ü = $[b_{10}\ b_9\ b_8\ b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0\ h_4\ h_3\ h_2\ h_1\ h_0]$,

is obtained by: Ü = Ä G .

This calculation is performed seven times, once for each of the seven rows.
Left justifying each input row, the following subroutine can be called seven times:
Ü = [b_{10} b_9 b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0 0 0 0 0 0] ,

```
const uint16 Generator[11] = {
        0x13,
        0x1A,
        0x1F,
        0x1C,
        0x0E,
        0x15,
        0x0B,
        0x16,
        0x19,
        0x0D,
        0x07 };
uint16 hamming(uint16 U)
{
        int i;
        uint16 mask = 0x8000;
        uint16 result = 0;

        for (i=0; i<11; i++){
                if (0 != (mask & U)){
                        result ^= Generator[i];
                }
                mask = mask >>1;
        }
        result ^= U;
        return result;
}
```

Resulting in:

b_{71}	b_{70}	b_{69}	b_{68}	b_{67}	b_{66}	b_{65}	b_{64}	b_{63}	b_{62}	b_{61}	$H1_4$	$H1_3$	$H1_2$	$H1_1$	$H1_0$
b_{60}	b_{59}	b_{58}	b_{57}	b_{56}	b_{55}	b_{54}	b_{53}	b_{52}	b_{51}	b_{50}	$H2_4$	$H2_3$	$H2_2$	$H2_1$	$H2_0$
b_{49}	b_{48}	b_{47}	b_{46}	b_{45}	b_{44}	b_{43}	b_{42}	b_{41}	b_{40}	c_4	$H3_4$	$H3_3$	$H3_2$	$H3_1$	$H3_0$
b_{39}	b_{38}	b_{37}	b_{36}	b_{35}	b_{34}	b_{33}	b_{32}	b_{31}	b_{30}	c_3	$H4_4$	$H4_3$	$H4_2$	$H4_1$	$H4_0$
b_{29}	b_{28}	b_{27}	b_{26}	b_{25}	b_{24}	b_{23}	b_{22}	b_{21}	b_{20}	c_2	$H5_4$	$H5_3$	$H5_2$	$H5_1$	$H5_0$
b_{19}	b_{18}	b_{17}	b_{16}	b_{15}	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	c_1	$H6_4$	$H6_3$	$H6_2$	$H6_1$	$H6_0$
b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	c_0	$H7_4$	$H7_3$	$H7_2$	$H7_1$	$H7_0$

The parity check bits (PC) shall be chosen such that each column of the BPTC matrix has an even number of "one" bits.

b_{71}	b_{70}	b_{69}	b_{68}	b_{67}	b_{66}	b_{65}	b_{64}	b_{63}	b_{62}	b_{61}	$H1_4$	$H1_3$	$H1_2$	$H1_1$	$H1_0$
b_{60}	b_{59}	b_{58}	b_{57}	b_{56}	b_{55}	b_{54}	b_{53}	b_{52}	b_{51}	b_{50}	$H2_4$	$H2_3$	$H2_2$	$H2_1$	$H2_0$
b_{49}	b_{48}	b_{47}	b_{46}	b_{45}	b_{44}	b_{43}	b_{42}	b_{41}	b_{40}	c_4	$H3_4$	$H3_3$	$H3_2$	$H3_1$	$H3_0$
b_{39}	b_{38}	b_{37}	b_{36}	b_{35}	b_{34}	b_{33}	b_{32}	b_{31}	b_{30}	c_3	$H4_4$	$H4_3$	$H4_2$	$H4_1$	$H4_0$
b_{29}	b_{28}	b_{27}	b_{26}	b_{25}	b_{24}	b_{23}	b_{22}	b_{21}	b_{20}	c_2	$H5_4$	$H5_3$	$H5_2$	$H5_1$	$H5_0$
b_{19}	b_{18}	b_{17}	b_{16}	b_{15}	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	c_1	$H6_4$	$H6_3$	$H6_2$	$H6_1$	$H6_0$
b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	c_0	$H7_4$	$H7_3$	$H7_2$	$H7_1$	$H7_0$
Calculate Even Parity.															
P_{15}	P_{14}	P_{13}	P_{12}	P_{11}	P_{10}	P_9	P_8	P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0

trix top-to bottom and

Each row of the resulting Transmit Matrix is 32 bits long and is placed in the embedded signaling field of four sequential bursts

b_{71} b_{60} b_{49} b_{39} b_{29} b_{19} b_{9} P_{15}
b_{70} b_{59} b_{48} b_{38} b_{28} b_{18} b_{8} P_{14}
b_{69} b_{58} b_{47} b_{37} b_{27} b_{17} b_{7} P_{13}
b_{68} b_{57} b_{46} b_{36} b_{26} b_{16} b_{6} P_{12}

b_{67} b_{56} b_{45} b_{35} b_{25} b_{15} b_{5} P_{11}
b_{66} b_{55} b_{44} b_{34} b_{24} b_{14} b_{4} P_{10}
b_{65} b_{54} b_{43} b_{33} b_{23} b_{13} b_{3} P_{9}
b_{64} b_{53} b_{42} b_{32} b_{22} b_{12} b_{2} P_{8}

b_{63} b_{52} b_{41} b_{31} b_{21} b_{11} b_{1} P_{7}
b_{62} b_{51} b_{40} b_{30} b_{20} b_{10} b_{0} P_{6}
b_{61} b_{50} C_{4} C_{3} C_{2} C_{1} C_{0} P_{5}
$H1_{4}$ $H2_{4}$ $H3_{4}$ $H4_{4}$ $H5_{4}$ $H6_{4}$ $H7_{4}$ P_{4}

$H1_{3}$ $H2_{3}$ $H3_{3}$ $H4_{3}$ $H5_{3}$ $H6_{3}$ $H7_{3}$ P_{3}
$H1_{2}$ $H2_{2}$ $H3_{2}$ $H4_{2}$ $H5_{2}$ $H6_{2}$ $H7_{2}$ P_{2}
$H1_{1}$ $H2_{1}$ $H3_{1}$ $H4_{1}$ $H5_{1}$ $H6_{1}$ $H7_{1}$ P_{1}
$H1_{0}$ $H2_{0}$ $H3_{0}$ $H4_{0}$ $H5_{0}$ $H6_{0}$ $H7_{0}$ P_{0}

VIII. REAL TIME PROCESSING

There always seem to be colorful debates about the real-time capability of Windows on a general purpose computer. For consumer applications, Windows does a great job at playing audio and even HDTV, with spreadsheets calculating in the background, and interactive game sound effects. What seems to be happening here is that Windows does a great job at these things, and continues to improve from generation to generation, precisely because these are the high profile applications targeted by Microsoft. The advice to developers of these applications is always allocate a HUGE streaming buffer! For non-interactive playback, latency does not matter much. Fortunately this is improving as game developers push for better responsiveness. By some reports there have been substantial improvements in Windows Audio Session API (WASAPI) for Vista/Windows7. Unfortunately, these more modern API's were not within the goals for this demonstration.

An area which seems to not have improved much, and always seems to keep the older API's longer, is in microphone sound capture. The high-end sound production applications have evolved to using special sound capture hardware, with ASIO or kernel streaming drivers. [Or Skype!] We did not want to require specialized high-end audio equipment for this project, nor did we have the expertise to write at the driver level. These remain opportunities for3rd parties. Over the last fifteen years there have been several Microsoft initiatives to address enterprise VoIP communication. These may eventually help with these audio latency problems.

We experimented with numerous variations of the old wave API's, and Direct Sound, with different threading and buffering approaches, in hopes of finding the "magic bullet" which would give stable performance. We eventually got a combination to work. We are not certain this is really the best combination.

The actual speed of the USB transport between the PC and the Dongle did not actually present much of a problem. There is plenty of bandwidth to support even a few more voice channels. What was somewhat problematic was the COMM port CDC emulation mode in which we used USB. The Microsoft API's for USB virtual COMM ports work similar to real serial ports, but not *exactly* similar. We ended up having to play tricks with buffer sizes in order to avoid short packets getting "stuck" for long durations.

Here we will illustrate a couple of the common audio problems we had during this project. Perhaps these can be avoided by other developers. In general, the problems we encountered were not due to lack of processing cycles. Even on our anemic 2GHz Pentium M the application never used more than a few percent of the processing time. The problems always came down to meeting some hard real-time deadline.

We knew going into the project that ordinary Windows timers used for thread timeouts or *GetTickTime()* were at best quantized to 15mS, and could be delayed by hundreds of milliseconds due to higher priority processes like mouse clicks or disk drive access. We *thought* that the hardware timers used to sample audio, and the delivery of audio packets from the OS to the application thread would be highly accurate. We were encouraged by reports of folks successfully using wave buffers shorter than 20mS. Needing to keep our jitter on the network below 60mS, we felt we had plenty of design margin. We chose the 20mS firing of the sound thread microphone sampler as our master clock. At 8Ksps, the sound thread should collect 160 samples every 20mS.

Boosting our encouragement, our first testing was done on a somewhat elderly desktop PC with integrated sound on the motherboard. Figure 16 plots the time difference between successive 20mS sound thread wake-ups. The variation compared to the 20mS period is very small. In terms of timing, this worked successfully right from the start, receiving and transmitting on the IPSC network!

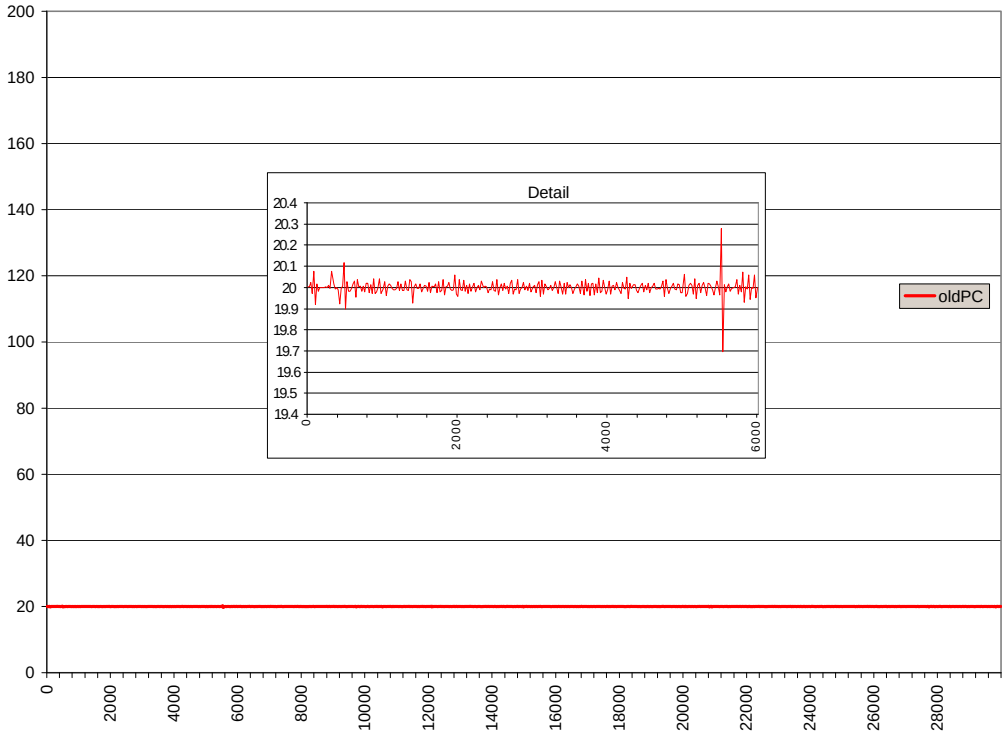


Figure 14: Timing Variation using Elderly PC

The next step, of course, was do try a Demo using a (somewhat more modern) laptop. Receiving from the IPSC network and playing on the speaker continued to (appear to) work. Any transmission attempts would momentarily key the repeater. Voice on a monitor radio sounded like a "sliding squawk." The repeater would soon drop the call.

Some investigation discovered that the laptop was not really sampling the microphone at an 8Ksps rate. The rate was more like 8050sps. The old *telephony* 8Ksps rate is not well supported in modern PC's. Thinking this may be the problem, we changed the microphone sampling rate to 48Ksps, and processed the samples through a 32-tap decimation filter to reduce the sampling rate down to the required 8Ksps.

This indeed did work, and the laptop could then both receive and transmit on the IPSC network. This may have worked, but perhaps not for the reason we thought it had worked. In any case, the notion of running the audio sampler at a higher rate, and decimating down to the 8Ksps rate, is likely the preferred approach. With some improvement in the design this may be the best way to achieve DVSI's tight frequency mask requirements.

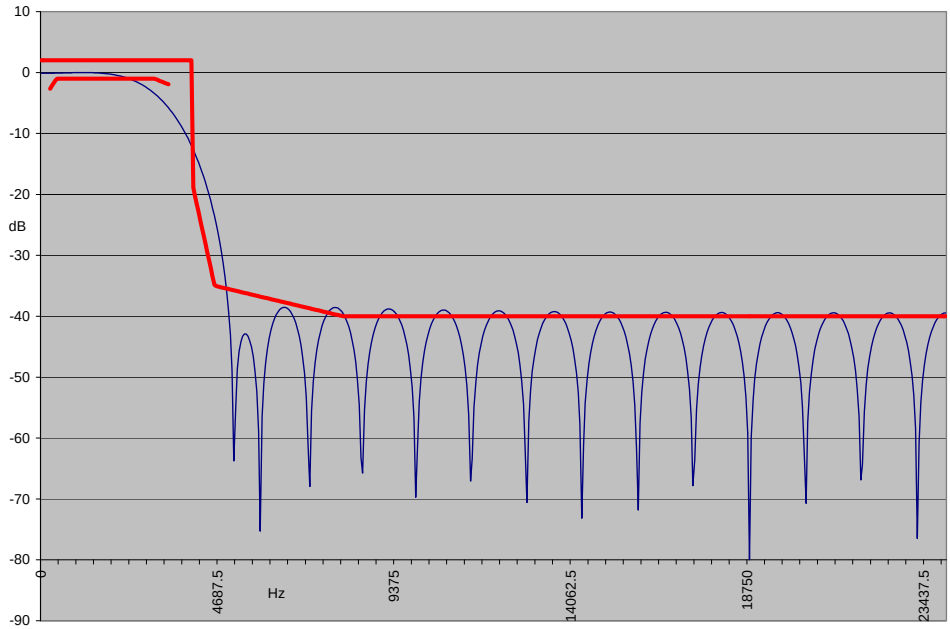


Figure 15: Decimation Filter Frequency Response

Being burned once with performance failures using different PC boxes, we tried the Demo on other hardware models. We found in general that the application did not work on newer model laptops. The problem appeared to be a periodic variation in audio sample delivery. The average rate was correct, but four, five, or six consecutive blocks would arrive a few milliseconds late, followed by one very early block. What ultimately fixed this was improving the robustness of the USB serial communication to the Dongle.

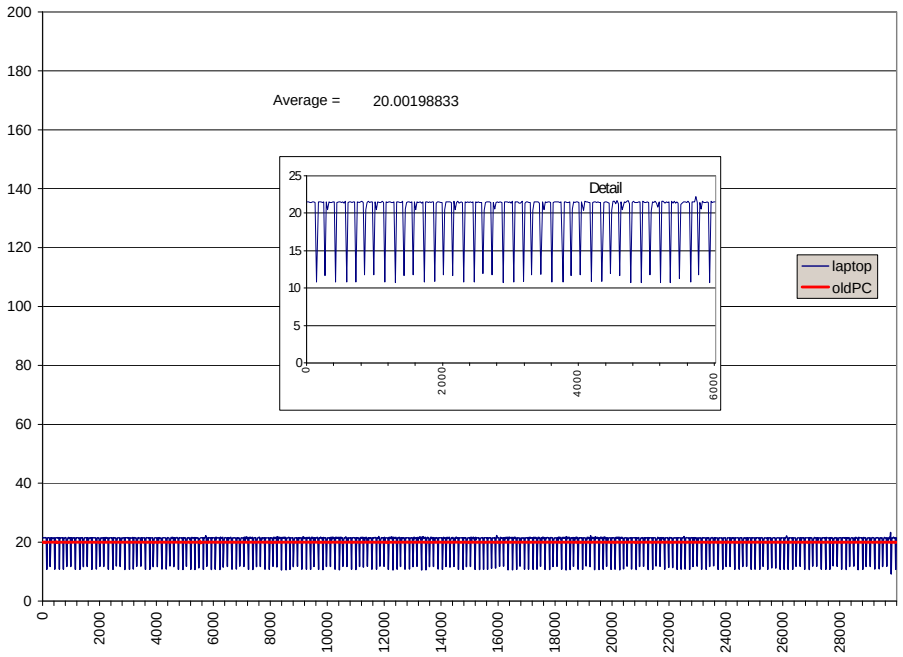


Figure 16: Sample Time Variation of Laptop Computers

After all this, transmitting on the IPSC network seemed to be stable, but receiving would still sometimes fail.

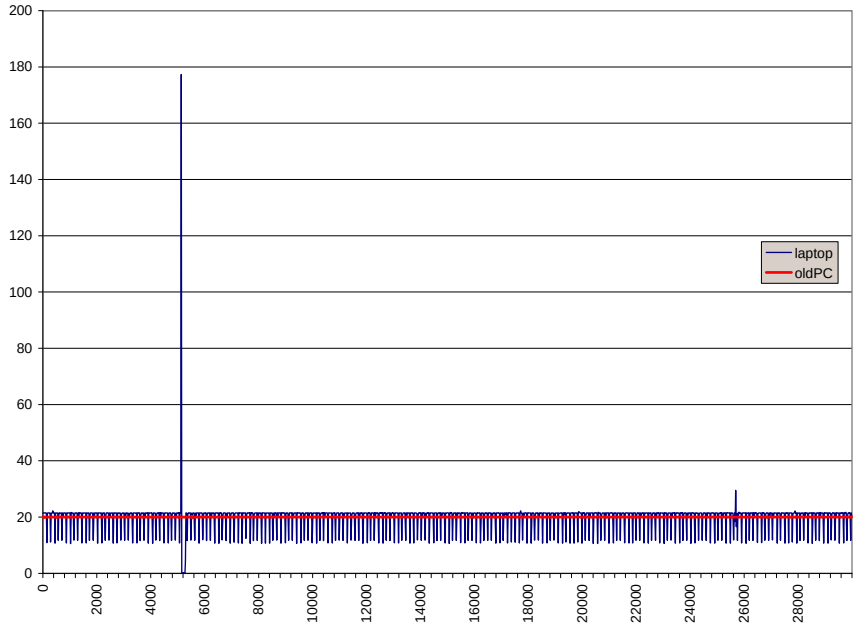


Figure 17: Large Error at start of Reception

We consistently observed, on some computers in some installations, a large >150mS gap in the Sound thread, followed by several queued blocks in rapid succession. This gap occurred immediately at the start of the reception. There would occasionally be shorter gaps, such as the one at the right of Figure 19. Plus, the usual random crackles in Microsoft audio due to moving the mouse around, refreshing windows, etc.

The large gap at the start of the reception was traced to the BlackIce software firewall we are required to have on our business machines.

The large gap, and many of the smaller ones, was somewhat mitigated by adjusting thread priorities and the rules used to mute our speaker. More work remains here in refining these algorithms. In principle, *bandaids* on the receive audio should be fairly effective. It is less clear how to mitigate such gaps *should they occur* while transmitting. This will require a greater in depth understanding the tolerable dropouts at the IPSC repeater.

IX. SOUND PROCESSING

Two sound processing classes are included in the demonstration: The current class CDirectSound; And the (out of date) CSimpleSound. The older class is included for reference as it uses the older wave API's.

Much of the detail of Microsoft sound API's will not be covered in this paper. We've tried to make the comments in this code track closely with the Microsoft developer web site.

The constructor for CDirectSound calls *EnumerateCaptureDevices()* and *EnumeratePlaybackDevices()* to illustrate the process of discovering available sound devices. In the general case this list of devices could be used by the user to select the desired sound device. The first device enumerated is always called the Primary Sound Driver, and the lpGUID parameter of the callback is NULL. This device represents the preferred playback device set by the user in Control Panel, and is the device always used by this demo application.

Sound capture works by passing a buffer to DirectSound and an array of location descriptors within the buffer. After the sound system has completed filling to a location an event is triggered. In our case our buffer is a contiguous array of 32 buffers. Each one holds the number of microphone samples in a 20mS block. [At 48Ksps this is 960 samples.]

```
const int IN_BYTES_PER_SAMPLE = 2;
const int IN_BITS_PER_SAMPLE = 16;
const int NUM_IN_BUFFERS =   32;
const int IN_BUFFER_MASK =   NUM_IN_BUFFERS - 1;
const int IN_SAMPLES_PER_S = 8000;
const int IN_SAMPLES_PER_20mS = 160;
const int IN_BYTES_PER_20mS   =
          IN_SAMPLES_PER_20mS * IN_BYTES_PER_SAMPLE;

#if USING_DECIMATION
const int IN_SOUNDCARD_SAMPLES_PER_S = 48000;
const int IN_SOUNDCARD_SAMPLES_PER_20mS = 960;
#endif

const int IN_SOUNDCARD_BYTES_PER_20mS   =
          IN_SOUNDCARD_SAMPLES_PER_20mS
          * IN_BYTES_PER_SAMPLE;

//960 samples per 20mS @48K, or 1920 bytes per 20mS.
typedef union
{
 unsigned __int8 CharBuf[IN_SOUNDCARD_BYTES_PER_20mS];
 __int16 ShortBuff[IN_SOUNDCARD_SAMPLES_PER_20mS];
}SoundCardPCM;

//160 samples per 20mS @8K, or 320 bytes per 20mS.
typedef union
{
  unsigned __int8 CharBuf[IN_BYTES_PER_20mS];
  __int16  ShortBuff[IN_SAMPLES_PER_20mS];
}InPCM;

//Buffers attached to DirectSound.
SoundCardPCM SoundCardInXferBuffer[NUM_IN_BUFFERS];
//Buffer to pass to Dongle.
InPCM        m_ScratchPCM;
```

The user dialog starts the sound processor running by calling:

```
CDirectSound::StartSound(HWND hParentWnd,
                         int uInDeviceID,
                         int uOutDeviceID)
```

Local member variables and sound buffers are initialized and an event object is created for each input buffer plus one used to tickle the thread. The sound thread is then started in sequence:

```
//Start Sound Thread suspended.
m_pSndThread = AfxBeginThread(SndThreadProc,
                                        this,
               THREAD_PRIORITY_TIME_CRITICAL,
                                           0,
                                CREATE_SUSPENDED);
if (NULL == m_pSndThread){
    //Error Handling
    . . .
}
m_pSndThread->m_bAutoDelete = FALSE;

//DirectSound has to open Output before Input.
result = OpenOutput();
if (0 != result){
   //Error handling.
   . . .
}

//Now try opening input.
result = OpenInput();
if (0 != result){
   //Error handling.
   . . .
}

//Now start thread running.
m_pSndThread->ResumeThread();
```

OpenOutput() follows the Microsoft examples very closely so will not be described here. The only thing somewhat tricky is the format for the Primary buffer is what we want for sampling the microphone *input*, not for playing the speaker *output*. By forcing the Primary Buffer to our input choice (if capable), it assures that our preferred input sample rate will be available, and not require re-sampling by Windows. A Secondary buffer is created using the actual playback format, and zeroed with the pointer starting at the beginning. Playback is not started.

OpenInput() follows the Microsoft examples very closely so will not be described here. The only thing somewhat tricky is the array of location descriptors we pass to DirectSound has 32 positions, corresponding exactly to the array boundaries of our 20mS buffers. Essentially what will happen here is an event will fire every time a new buffer is filled. Sampling is immediately started.

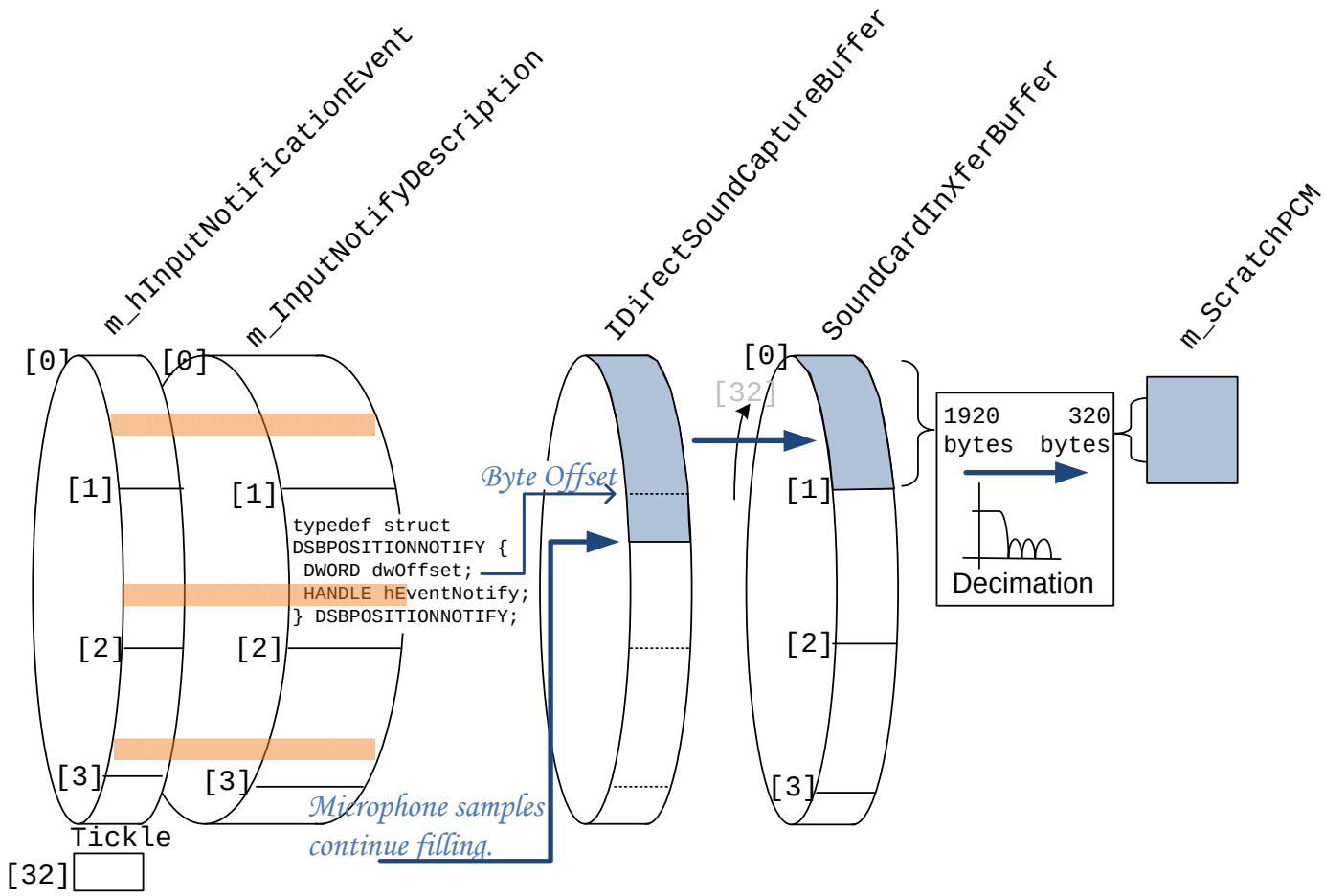


Figure 18: DirectSound Capture

The DirectSound microphone sampler runs continuously, wrapping around the contiguous IDirectSoundCaptureBuffer. As this buffer fills *past* each of the byte offsets specified in m_InputNotificationDescription[] the corresponding m_hInputNotificationEvent[] fires.

```
void CDirectSound::SoundThread()
{
   DWORD result;
   DWORD EventIndex;
   while(!m_ShutDownThreads)
   {
      EventIndex = WSAWaitForMultipleEvents(
                           NUM_IN_BUFFERS+1,
                  m_hInputNotificationEvent,
                                    FALSE,
                                WSA_INFINITE,
                                    FALSE);
      if (WSA_WAIT_FAILED == EventIndex){
         //Something really bad happened.
         result = WSAGetLastError();
         return;
      }

      EventIndex -= WSA_WAIT_EVENT_0;
      if (NUM_IN_BUFFERS != EventIndex){
         m_EventHasFired[EventIndex] = TRUE;
         HandleSoundInput();
         HandleSoundOutput();
      }
   }
}
```

Nominally the SoundThread will wake up at an uniform, predictable 20mS rate after one of the sample buffers is filled. However (during disk drive activity or other random occurrences) the wakeup may be delayed and more than one notification event may have fired. *Long* delays while transmitting are likely unrecoverable, and a hardened application should most likely take down the call and initiate error recovery. Shorter delays during transmit may be recoverable due to the averaging over the 60mS network frame.

The EventIndex returned by *WSAWaitForMultipleEvents()* is the *lowest* numerical value. This lowest numerical value may correspond to the oldest buffer captured. But because of buffer wrap this lowest number may not necessarily be the oldest buffer.

To adjust for this, *HandleSoundInput()* remembers from call-to-call the index of the last buffer processed. *WSAWaitForMultipleEvents()* returned the lowest numerical value, any gaps in the buffer index are immediately apparent. The correct action is to process the [(last+1 Mod32)] buffer, clear its Event Object, and return the buffer to DS. *HandleSoundInput()* copies the oldest available 20mS buffer from IDirectSoundCaptureBuffer to SoundCardInXferBuffer[], and advances m_SoundCard_InHeadPtr to indicate a new set of samples is available.

Subsequent handling of the copied microphone samples is call dependent, using global flags g_bPTT and g_bTX. Some caution must be employed in keeping these thread safe.

g_bPTT — Is a flag which can be set or cleared at any time by the user dialog.

g_bTX — Is a flag indicating that network transmission is in progress. It is asserted by SoundThread when calling *g_MyNet.NetTx(TRUE)*. It is released by the network thread after clean depletion of the transmission.

Snapshots are taken of these flags once every 20mS pass through *HandleSoundInput()*.

```
snapPTT = g_bPTT;  //Catch snapshot at this instant.
snapTX  = g_bTX;
```

During idle, neither flag is asserted and any new microphone samples in SoundCardInXferBuffer[] are just discarded.

When the user dialog asserts g_bPTT (and g_bTX is released), it is a new PTT request. NetTx(TRUE) is called to initialize the parameters for a new transmission. g_bTX is asserted by this call. FlushAnyAMBEtoDongle() is called to discard any received AMBE samples which may be in the pipeline.

```
if ((!snapTX) && snapPTT){ //New PTT request.
     g_MyNet.NetTx(TRUE);
     g_Serial.FlushAnyAMBEtoDongle();
}
```

```
SomeMicWasSent = FALSE;
while (m_SoundCard_InTailPtr != m_SoundCard_InHeadPtr){

    if (snapTX){ //Transmitting,

       if (!snapPTT){ //User Dekeyed...
         //Zero samples coming from DS.
         for (i=0; i<IN_SOUNDCARD_SAMPLES_PER_20mS; i++){

             SoundCardInXferBuffer[m_SoundCard_InTailPtr]
                                  .ShortBuff[i] = 0;
         }
       }

       DecimationFilter();
       g_Serial.SendDVSIPCMMsgtoDongle(
                          &m_ScratchPCM.CharBuf[0]);
       SomeMicWasSent = TRUE;
     }

  m_SoundCard_InTailPtr = (m_SoundCard_InTailPtr + 1)
                           & IN_BUFFER_MASK;
}

if (TRUE == SomeMicWasSent){
   g_Serial.TickleMicThread();
}else{
   g_Serial.SendAnyAMBEtoDongle();
}
```

For each un-processed buffer, if we are transmitting based on snapTX:

> We first clear the buffer if snapPTT is un-asserted. This will continue to process silence until the network gracefully depletes the transmission. [Should the network thread release g_bTX this silence will get flushed. Should the user dialog re-assert g_bPTT at any time while the network is depleting a previous transmission, we'll just resume sending voice samples, and the call will continue as if it never ended. If the user dialog re-asserts g_bPTT while the network successively depletes the previous transmission, a new call will be initialized at the next pass of SoundThread.]
>
> We run the decimation filter on the new SoundCardInXferBuffer[*n*] buffer, (including transient samples from the previous SoundCardInXferBuffer[*n*-1] buffer). This filter stores the new 160@8Ksps samples in a temporary buffer m_ScratchPCM.CharBuf[]. Any silence placed in the 48Ksps buffer by the previous step is processed like any other 48Ksps sound buffer.
>
> We flag that SomeMicWasSent and call *SendDVSIPCMMsgtoDongle()* to queue the temporary buffer to the serial port. *SendDVSIPCMMsgtoDongle()* handles the Little Endian to Big Endian conversion of the16 bit PCM samples.

We then advance m_SoundCard_InTailPtr to deplete the buffer.

If SomeMicWasSent, we call g_Serial.TickleMicThread() to wake up the serial sender thread.

Playing speaker audio using DirectSound works very nicely, as long as the playback buffer is large, and one knows exactly when the buffer is going to end (so one can swap in a new large buffer with a known end). It is not really intended to handle small buffers, when the streaming stops at unpredictable times. The classic fix for this is to accumulate many small buffers into a larger buffer to pass to DS, but this greatly increases delay.

The biggest challenge with DS speaker playback was to stop the playback near the end; without significant truncation of the received audio, and without unwanted looping over previous samples, while keeping latency low. The problem is related to detection of when a call ends, when delivery of voice frames may be variable or dropped altogether.

Buffer management for playback is conceptually fairly simple; not using any notification events.

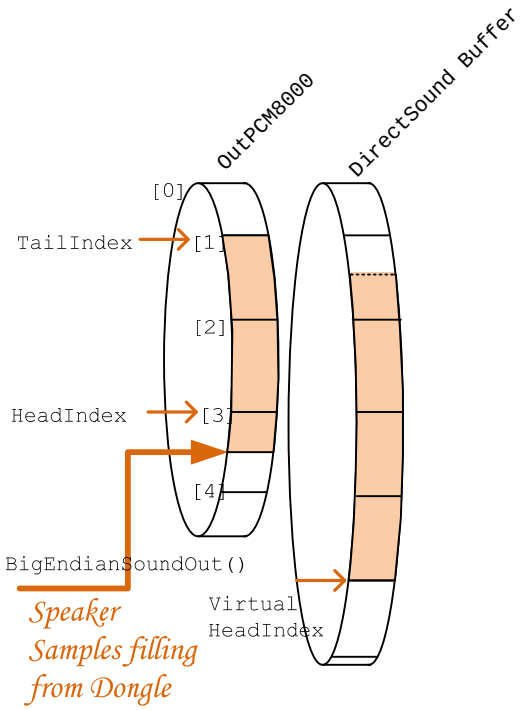


Figure 19: Play Out Buffers

Sound output buffers are defined in a similar manner as were the input buffers. Here, we keep the output sample rate at 8Ksps throughout. This seemed to work OK, but in general an interpolation filter to 48Ksps should not incur detrimental loading, and may improve performance. Thirty-two 20mS buffers, organized as a circular array, are managed by software: OutPCM8000[NUM_OUT_BUFFERS].

Sixty-four 20mS buffers, organized as contiguous bytes, are managed by DS. These are treated by DS as one large buffer, that will wrap around. As 20mS blocks are de-queued from the software managed array, they are written into this DS buffer at virtual array boundaries.

As 20mS PCM frames are returned from the Dongle, the serial reader thread calls *BigEndianSoundOut()* to convert the Big Endian PCM samples to Windows Little Endian, and enqueues the frame to OutPCM8000[]. Newsamples are enqueued at m_OutHeadPtr. This index is only changed by serial reader thread, and is read by the sound thread to determine if new samples are present. The m_OutTailPtr is only

moved by the sound thread when it de-queues a frame, and is read by the serial reader thread to avoid collisions.

Speaker muting is controlled by a state variable, m_bOutPumpIsPrimed, indicating if the speaker is currently playing, and a hysteresis counter, m_OutHysteresisCount.

HandleSoundOutput() is called by the sound thread nominally every 20mS.

```
const int OUT_BYTES_PER_SAMPLE = 2;
const int OUT_BITS_PER_SAMPLE = 16;
const int NUM_OUT_BUFFERS = 32;//Softwaremanaged.
const int OUT_BUFFER_MASK =  NUM_OUT_BUFFERS - 1;
const int OUT_SAMPLES_PER_S = 8000;
const int OUT_SOUNDCARD_SAMPLES_PER_S = 8000;
const int OUT_SAMPLES_PER_20mS = 160;
const int OUT_BYTES_PER_20mS   =
OUT_SAMPLES_PER_20mS * OUT_BYTES_PER_SAMPLE;

typedef struct
{
   unsigned __int8 Buf[OUT_BYTES_PER_20mS];
}OutPCM;

OutPCM       OutPCM8000[NUM_OUT_BUFFERS];

const int NUM_DS_OUT_BUFFERS =  64; //DS managed.
const int OUT_DS_BUFFER_MASK = NUM_DS_OUT_BUFFERS
                               - 1;
const int OUT_BYTES_DS_BUFFER =
        OUT_BYTES_PER_20mS * NUM_DS_OUT_BUFFERS;
const int OUT_HYSTERESIS_LIMIT = 8;
const int MUTING OVERRUN = NUM DS OUT BUFFERS/2;
```

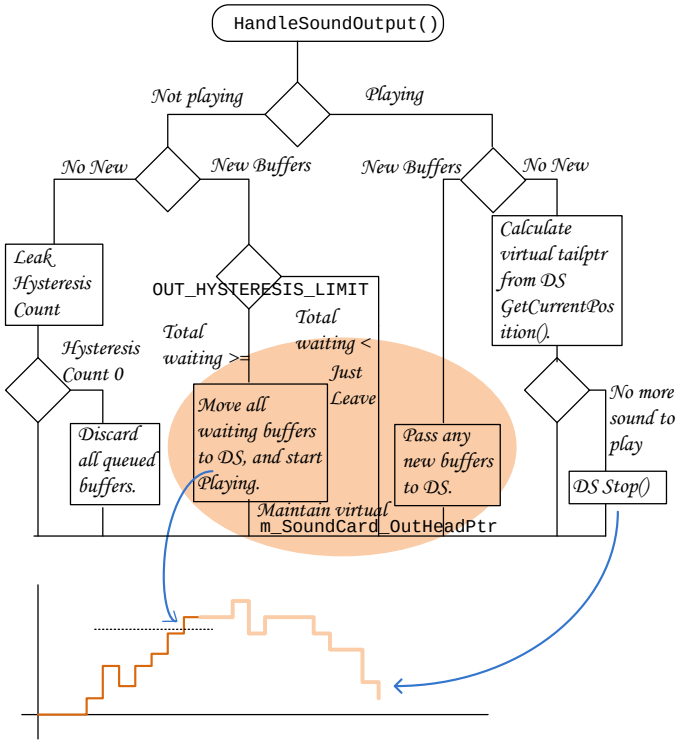


Figure 20: Speaker Muting Decision.

Once started, DirectSound will continue to play in a loop *its entire 64 frame buffer* until told to stop. Due to jitter, lost packets, or slight mis-matches between the repeater and local PC clocks, there may not be a new

frame ready to feed the speaker exactly every 20mS. One way to handle this is to delay the start of the play out until several frames have arrived, then start playing. Continue to add new frames as they arrive; any jitter between the nominal 20mS sound handler and the actual frame availability will be absorbed by the delay of the queued frames. [In the general case, the sequence number can be used *a posteriori* to determine that a previous frame was lost, and a silence or repeat frame substituted.]

There are a couple special cases to handle:

Should a very short reception come in, shorter than the un-muting threshold, there must be some mechanism to deplete any frames stuck in the queue.

Since the sound thread only wakes up every 20ms, and this wakeup may sometimes be delayed by other things going on in Windows, it does not seem possible to stop the playback *exactly* at the end. Getting within a couple 20mS frames seems workable. In particular, the case of buffer wrapping and re-playing the entire 64 buffers (1.3Seconds) should be avoided.

In this example code a value of 8 was used for the OUT_HYSTERESIS_LIMIT, introducing an additional 160mS play out latency. This value may be needlessly too high for some implementations.

X. SERIAL PORT DONGLE COMMUNICATION

The Dongle firmware USB code was based off of the Communications Device Class (CDC) example in the Atmel/AVR software framework. This framework provides the *at32uc3xxx_cdc.inf* file to configure Windows USB to COMM emulation drivers. In practice, this COMM port emulation turned out to be somewhat awkward to use. Not all features of the Windows serial port API's worked quite as expected; in particular, the ability to flush a Tx buffer, receive notification upon a Rx gap timeout, and some of the error recovery calls to abort serial communication, did not seem to work correctly. [Or, they did, but failed due to slight miss-applications of the exact wording of the API's.]

For a hardened product, it would be well to invest the effort into writing dedicated Windows USB drivers.

In general, the first obvious choice of using isochronous transfers may not work out too well due to the dynamic switching of frame size between PCM and AMBE. Plus, we already had enough challenges using Microsoft Sound API's for the microphone and speaker interfaces. It is possible that an isochronous transfer is the best route, but this will require more investigation.

A more fruitful path may be to use interrupt transfers to guarantee latency, with ideally some way to better handle timely delivery of variable size packets. It may be possible to implement this using the WinUSB generic driver.

Serial communication with the Dongle is initiated by the user dialog calling the OpenDongle() public function in CSerialDongle. This call has the user supplied string to the desired COMM port. This initializes local variables, creates event object arrays for the reader and writer threads, initializes COMM port hardware, and starts the two threads running.

Links to the Microsoft documentation never seem to stay current for more than a few days, but detailed tutorials on serial port communication may be found somewhere in the MSDN library [http://msdn.microsoft.com/en-us/library/aa363196(v=VS.85).aspx]. A handle to the COMM port is obtained by calling *CreateFile()* with the user specified string. The local *SetupDongle()* and *PurgeDongle()* functions are intended to follow the Microsoft examples, with the already mentioned caution that ReadIntervalTimeout does not really seem to work as expected. *PurgeDongle()* is also called, with more or less success, during error recovery.

During development we tried several different approaches to the Dongle serial thread handling. It is complicated in that the USB receiver really wants to know exactly how many bytes to expect; if fewer arrive, the short packet will not be delivered in a timely manner. Most of the time, the receiver can know what to expect based on what was sent: Sending PCM should expect to receive 15 byte AMBE frames; Sending AMBE should expect to receive 328 byte PCM frames. This doesn't work for variable size control frames. Trying to sort this out within one thread, with Tx complete notifications and Rx received notifications coming in random orders, led to confusing code. The added complexity of overlapped I/O on the surface does not seem beneficial, however it enabled us to break up the serial communication into two *fairly independent* easy to understand reader and writer threads.

Data is queued to the Dongle through two circular buffers; one for PCM frames and one for AMBE frames.

When the sound thread is transmitting and has PCM audio from the microphone to send to the Dongle, it packages the 320 bytes into a scratch buffer, and calls *SendDVSIPCMMsgtoDongle()*. [Figure 20]

> At the start of a new transmission, prior to calling *SendDVSIPCMMsgtoDongle()*, the sound thread will have called *FlushAnyAMBEtoDongle()*. This sets
> m_bPleasePurgeAMBE to TRUE
> so the writer thread will discard any remaining AMBE frames upon waking up. It also sets
> m_dwExpectedDongleRead to AMBE3000_AMBE_BYTESINFRAME,
> and tickles the reader thread so that it can expect to receive AMBE frames.

When the sound thread has queued any PCM, it calls *TickleMicThread()*. This routine calls *SetEvent(m_hTickleTxSerialEvent)* to wake serial writer.

```
void CSerialDongle::SendDVSIPCMMsgtoDongle( unsigned __int8* pData )
{
  int i;
  int snapPCMBufTail;

  snapPCMBufTail = (m_PCMBufTail - 1) & MAXDONGLEPCMFRAMESMASK;
  if (snapPCMBufTail != m_PCMBufHead){ //No collision.
      //Put new message into circular buffer.
      m_PCM_CirBuff[m_PCMBufHead].fld.Sync = AMBE3000_SYNC_BYTE;
      m_PCM_CirBuff[m_PCMBufHead].fld.LengthH = AMBE3000_PCM_LENGTH_HBYTE;
      m_PCM_CirBuff[m_PCMBufHead].fld.LengthL = AMBE3000_PCM_LENGTH_LBYTE;
      m_PCM_CirBuff[m_PCMBufHead].fld.Type = AMBE3000_PCM_TYPE_BYTE;
      m_PCM_CirBuff[m_PCMBufHead].fld.ID = AMBE3000_PCM_SPEECHID_BYTE;
      m_PCM_CirBuff[m_PCMBufHead].fld.Num = AMBE3000_PCM_NUMSAMPLES_BYTE;
      for (i=0; i< AMBE3000_PCM_INTSAMPLES_BYTE; i++){
          //Endian conversion.
          m_PCM_CirBuff[m_PCMBufHead].fld.Samples[1+(i<<1)] = *pData++;
          m_PCM_CirBuff[m_PCMBufHead].fld.Samples[  (i<<1)] = *pData++;
      }
      m_PCM_CirBuff[m_PCMBufHead].fld.PT = AMBE3000_PARITYTYPE_BYTE;
      m_PCM_CirBuff[m_PCMBufHead].fld.PP = CheckSum((DVSI3000struct *)&m_PCM_CirBuff[m_PCMBufHead]);
      m_PCMBufHead = (m_PCMBufHead + 1) & MAXDONGLEPCMFRAMESMASK;
   }
}
```

During transmission, if *SendDVSIPCMMsgtoDongle()* is able to find a non-colliding buffer in m_PCM_CirBuff[] queue, it appends the Dongle PCM packet header, copies the data to the buffer performing the Little Endian to Big Endian conversion, and appends the Dongle checksum, and advances m_PCMBufHead.

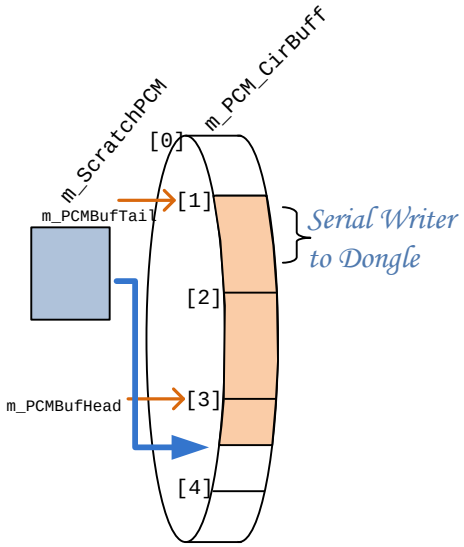


Figure 21: SendDVSIPCMMsgtoDongle()

When *NetWorker_ProcessCall()* detects the start of new reception it calls *FlushAnyPCMtoDongle()*. This sets
m_bPleasePurgePCM to TRUE
so the writer thread will discard any remaining PCM frames upon waking up. It also sets
m_dwExpectedDongleRead to AMBE3000_PCM_BYTESINFRAME
and tickles the reader thread so that it can expect to receive PCM frames.

During reception, *NetWorker_ExtractVoice(),* for each of the received AMBE frames, calls *GetFreeAMBEBuffer().* This routine pre-pends the Dongle AMBE packet header to the Head buffer of m_AMBE_CirBuff, and returns a pointer to the buffer. After filling, *NetWorker_ExtractVoice()* calls MarkAMBEBufferFilled(). This routine appends the Dongle checksum. If there is no collision m_AMBEBufHead is advanced.

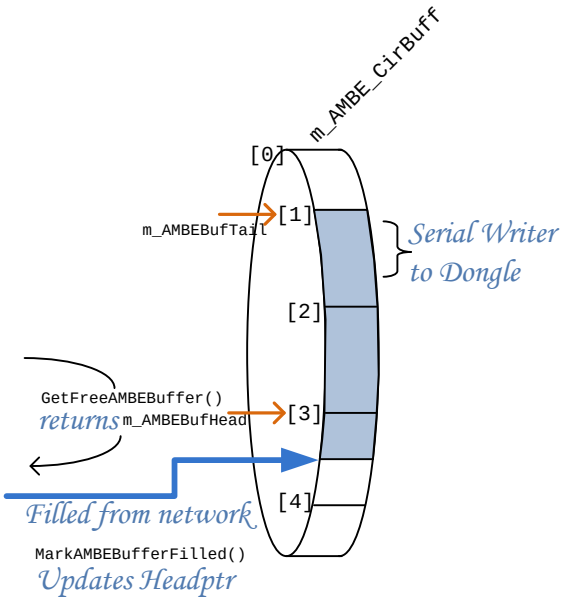


Figure 22: Networker_ExtractVoice()

When the sound thread is not transmitting, it calls *SendAnyAMBEtoDongle()*. This routine tests m_AMBEBufHead and m_AMBEBufTail. If they are unequal, *SetEvent(m_hTickleTxSerialEvent)* is called to wake the serial writer thread.

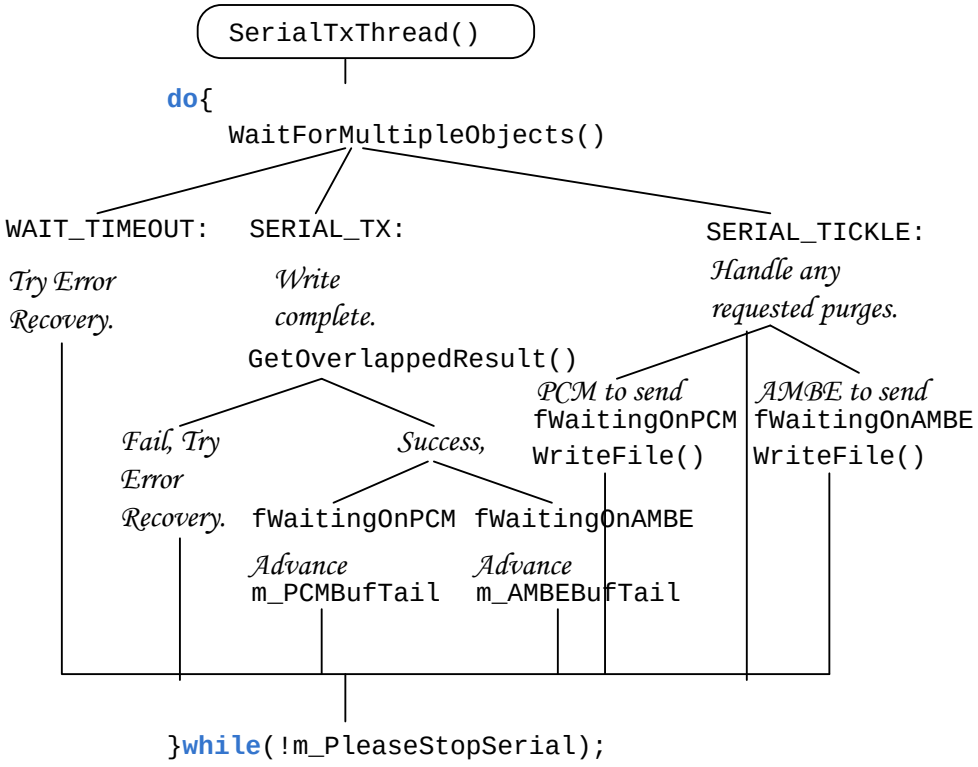


Figure 23: SerialTxThread (partial) Decision Tree

In the present code *SerialTxThread()* sends only one packet to the Dongle every time it is tickled awake. The packet is depleted from the queue when the write successfully completes. More complex schemes, such as allowing any available transmission whenever the previous transmission completes, *should* work, and *may* provide some margin for small Windows delays in the sound thread. Long Windows delays of hundreds on milliseconds have no real mitigation other than detecting the error and re-starting the call. [A scheme that stops and waits until the return packet is received from the Dongle is *not* a good strategy. Total time can be greater than 22mS.]

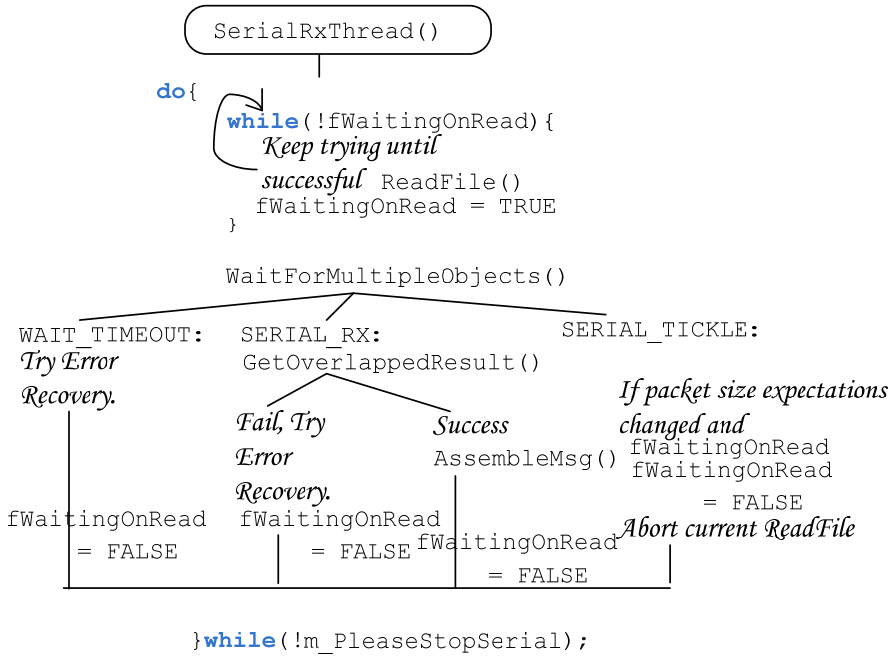


Figure 24: SerialRxThread (partial) Decision Tree

The *SerialRxThread()* is always listening for a packet returned by the Dongle. [We've looked at schemes that only enable the receiver when we *expect* something to be received; this *should* work, and may be somewhat cleaner. Here, if we are listening for something and our expectations for packet size change, we need to abort and re-start the reception.
Upon receiving a packet from the Dongle, *SerialRxThread()* calls *AssembleMsg()*. *ReadFile* will have placed the dwRead received bytes in m_DongleRxBuffer[]. In the general case these bytes may be a single message from the Dongle, multiple messages, or partial messages.

AssembleMsg() is a state machine that parses m_DongleRxBuffer[] character-by-character, first looking for the AMBE3000_SYNC_BYTE. Once this is found, *AssembleMsg()* begins storing the received packet into the beginning of m_RxDVSImsg.

The next byte received (should be) the high byte of length.

The next byte received (should be) the low byte of length. From these the whole message length is calculated. This length is compared with the maximum possible message length to prevent a false message from overflowing m_RxDVSImsg buffer.

The remaining bytes are read in as received. *ParseDVSImsg()* is called to evaluate the received message. *AssembleMsg()* will continue looking for AMBE3000_SYNC_BYTE, in order to handle multiple or partial messages.

ParseDVSImsg() (really should) call *CheckSum()* to verify the received bytes are actually a received message. There could be USB errors, or more likely false detection of AMBE3000_SYNC_BYTE.

ParseDVSImsg() dispatches the received message based on the Dongle format Type field:

AMBE3000_AMBE_TYPE_BYTE: This is an AMBE frame. *deObfuscate()* is called to remove the bit-scrambling. *NetStuffTxVoice()* is called to assemble the 60mS IPSC network burst.

AMBE3000_PCM_TYPE_BYTE: This is uncompressed 8000sps 16 bit data packet from the Dongle. *BigEndianSoundOut()* is called to queue the audio to the speaker.

AMBE3000_CCP_TYPE_BYTE: This is a control response from the Dongle.

XI. PROCESSOR LOADING

Processor loading measurements are shown in Figure 27. Figure 28 breaks the loading down by thread. Data was collected for the four major real-time threads; Sound, Network, Serial Reader, and Serial Writer by recording the performance counter at the exit and start of each *WaitForMultipleEvents()*. The raw time samples were processed to account for any of our threads interrupting each other. The data does *not* account for any other system events suspending our threads. Nor does the data account for any overlapped system processes which are utilized by our code. This only provides a measure of the execution efficiency of our application code, not any underlying system services.

The application was running on an AMD Sempron 3100+ processor (1800MHz clock), on an ABIT motherboard, with 1G of RAM. Realtek AC97 integrated motherboard sound was used.

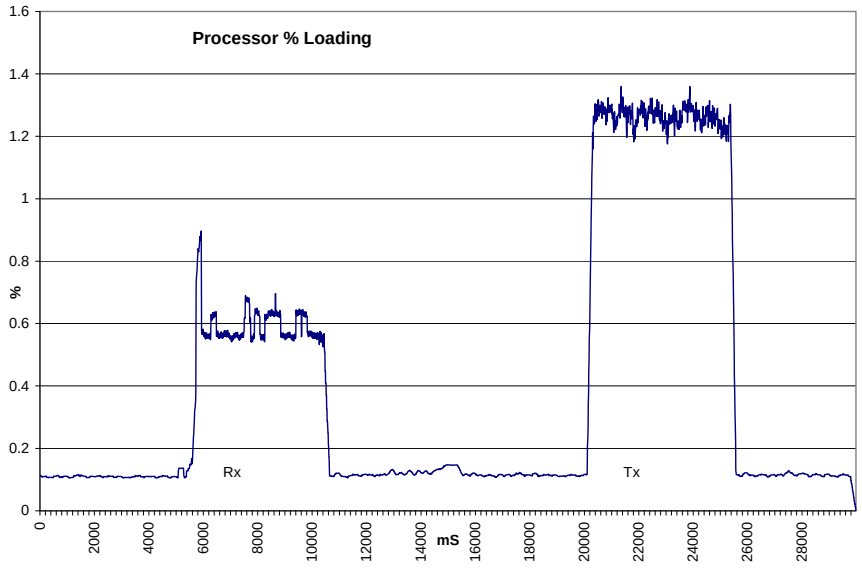


Figure 25: Processor Loading

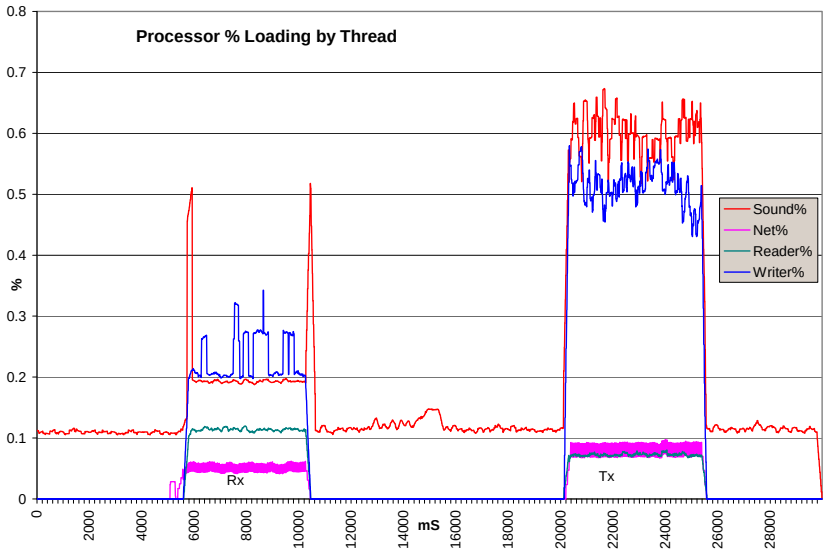


Figure 26: Loading by Thread

	Idle	Rx	Tx
Sound	0.1% 0.03mS peak	0.2% 0.67mS peak	0.6% 0.2mS peak
Network	0%	0.043% 0.046mS peak	0.07% 0.054mS peak
Serial Reader	0%	0.1% 0.028mS peak	0.07% 0.036mS peak
Serial Writer	0%	0.25% 0.19mS peak	0.5% 0.14mS peak
Total	0.1%	0.6%	1.3%

For reliable operation, the peak time spent in a thread is critical. This affects the thread's capability to complete its task, and allow other threads to complete their tasks, within some hard deadline.

The Sound thread is somewhat well-behaved in this implementation except for a 0.67mS spike at the beginning of a reception, and some lesser peaks at the end of reception.

Overlying Network activity (Figure 30), shows that the spike in the beginning occurs somewhat after IPSC frames start arriving from the network. Measurements such as this are useful during development in isolating problems and execution bottle-necks.

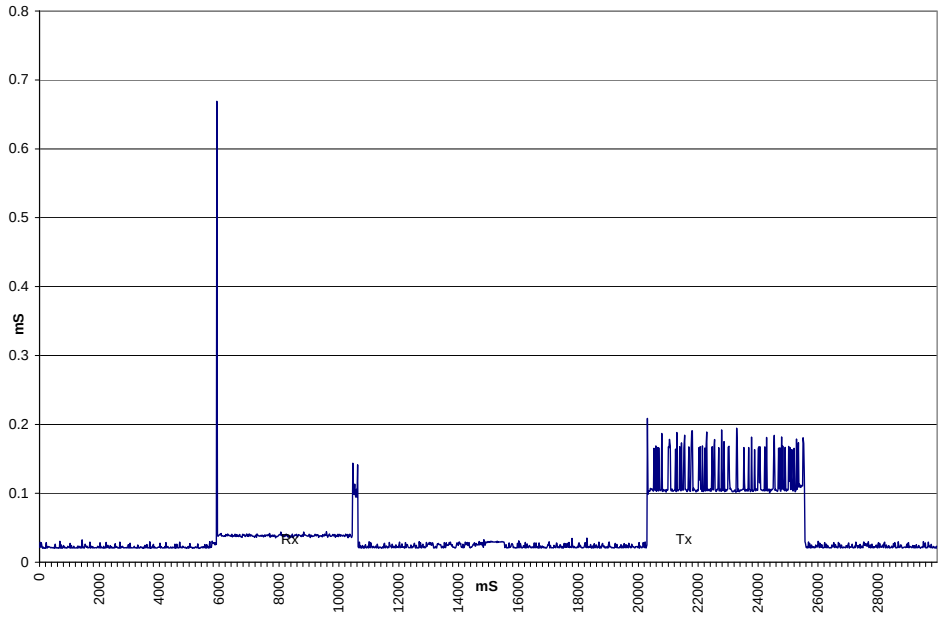


Figure 27: Peak Sound Thread Execution

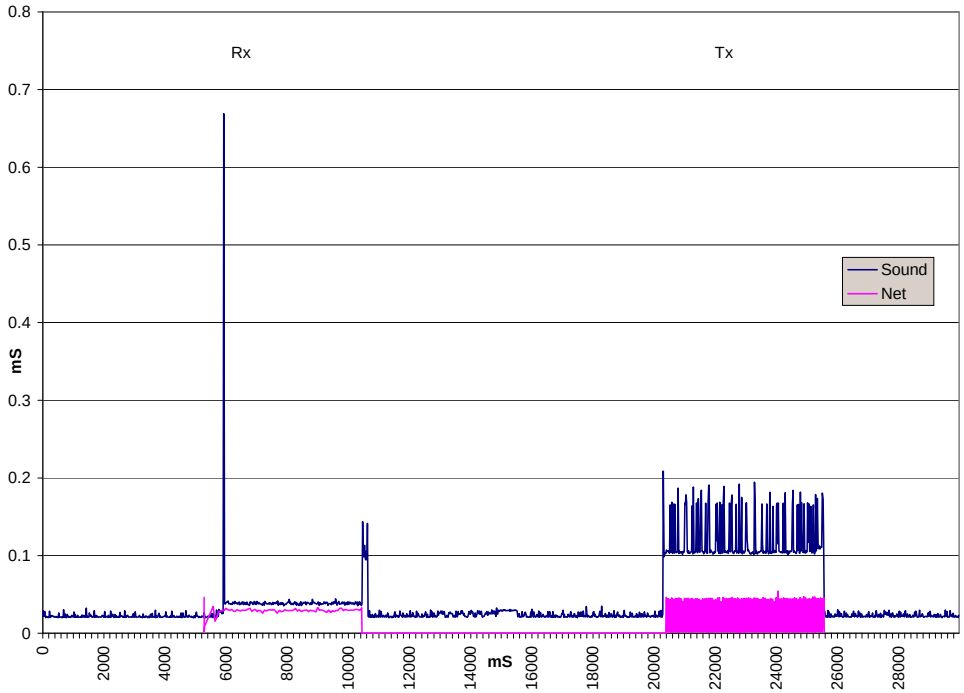


Figure 28: Sound and Network Activity

XII. LESSONS LEARNED

DVSI audio bit mapping between software vocoder and hardware vocoder came as quite a surprise. The software provided here is effective in de-scrambling these bits.

Modern PC's do not well support 8000sps microphone input sampling rates. Software decimation filters (and interpolation filters for Rx) are effective solutions. Built-in Windows rate changing algorithms are not as effective.

The Windows *GetTickCounts()* timer at best has about a 15mS resolution. All critical timing in this application was based off of DirectSound events.

For audio, Windows Audio Session API (WASAPI) is likely the best route going forward for Vista/Windows7.

Thread priorities must be increased to lessen sound dropouts and dropped calls due to windows/mouse activity.

A stop-and-wait strategy is not effective in feeding PCM frames to the Dongle. Independent serial reader and writer threads proved most effective. Developing an interrupt transfer USB driver could eliminate some of the ad *hoc fixes* employed in the present code.

Large (100's milliseconds) delays can occur at the start of receiving an IPSC call. This is due to software firewalls blocking the application threads. The solution for this in a Corporate site may be problematic and impact the viability of some products using general purpose PC's.

Actual processor peak and average loading while running the IPSC application is quite modest. Additional loading to support more audio channels or repeater peers should be feasible.

The present USART interface between the AVR processor and the *AMBE-3000*™ Vocoder Chip is likely inadequate to support multi-channel operation (using the AMBE-3003™). Other parallel or synchronous interfaces would provide higher bandwidth.

XIII. HOW TO BUILD THE PC APPLICATION

Required Software:
* Microsoft Visual Studio .Net 2003 or above (we are using MS Visual Studio 2008 as an example)
* MOTOTRBO IPSC AMBE Demo Package – ipsc.zip

Steps to Build the Application
1) unzip the ipsc.zip file under the Visual Studio Project directory, e.g. D:\DATA\USERNAME\Visual Studio Projects\ipsc_demo
2) Start Microsoft Visual Studio by selecting Start/All Programs/Microsoft Visual Studio 2008/Microsoft Visual Studio 2008
3) Open the ipsc project by selecting File/Open/Project in Microsoft Visual Studio 2008 and then selecting the ipsc project file by browsing in Visual Studio projects\ipsc_demo directory
4) Build the solution by selecting Build/Build Solution in Microsoft Visual Studio. And the executable file ipsc.exe is created under D:\DATA\USERNAME\Visual Studio Projects\ipsc_demo\ipsc\Release directory as an example.

XIV. How to Start the Application

1) Plug in the IPSC dongle to the PC through a USB cable. The In the PC New Hardware Wizard, manually select the *at32uc3xxx_cdc.inf* at D:\DATA\USERNAME\Visual Studio Projects\ipsc_demo\ipsc folder as the USB driver.

2) Copy the *mototrbo_ipsc_voice_demo.ini* file at C:/ directory
   a. *The first line contains the Master Peer's IP address and UDP port number.*
   b. *The second line contains the PC application's Peer ID.*
   c. *The third line contains the PC application's Radio ID.*
   d. *The fourth line contains the COMM port number that the dongle is connected to.*

   Please make sure the Master Peer's IP address and UDP port number match with what are configured in the repeater. Also if you connect the dongle at different USB port, use Control Panel/System/Hardware/Device Manager/Ports (COM & LPT) to find out the new COMM port number and update the file accordingly

3) Change the PC application IP address so that it is in the same subnet as the repeater peer. The PC's IP address can be changed at Control Panel/Network Connections/Local Network Connection/Internet Protocol (TCP/IP)/Properties/Using the following IP address.

4) Start the application by double clicking the ipsc.ext file, e.g. D:\DATA\USERNAME\Visual Studio\Projects\ipsc_demo\ipsc\Release

5) After the application starts, input the selected group ID and slot number in the field next to the "Select", and then click "Select". Separate the group ID and slot number by ":", for example, 2:0 means the selected group ID is 2, and the slot number is 1.

6) Select "Link" at the top menu row, and select "Connect" to join the IPSC system.

7) When the radio initiates a call, the PC application with the selected Group ID, the PC application can automatically play the received audio. When the PC application wants to talk back or initiate a call, click "PTT" button and start to talk. After finishing the talk, press "deKey" button to terminate the transmission.

8) To exit the application, select "File" at the top menu row and select "Exit".

Michael Retzer is a Member of the Technical Staff in the Professional & Commercial Radio group, and has been a software gunslinger for Motorola since 1976. In this role he has had to pull the cat out of the hat on numerous occasions. He holds seventeen patents covering various wireless and signal processing techniques using small processors. Mr. Retzer has been developing embedded software and hardware since paper tape was high-tech: “The languages have changed; the parts have gotten faster; more people overlook the same bugs.” Recent projects have included the hardware and software design for the Generic Option Board, a tutorial on two-way radio Trunking presented in Penang, Malaysia, and a Generic Option Board demo using the accelerometer.

Shaoqun Zhou is a Lead Software Engineer for the MOTOTRBO Application Developer Program Global Engineering Team. She joined Motorola in 1998. She has 8-year experience in Linux application development and Network Management for the Packet Data Gateway of Motorola mission critical tier two-way radios. She has 4-year application development for Motorola Professional and Commercial tier two-way radios. Shaoqun earned a BS in Engineering Physics and MS in Nuclear Electronics from Tsinghua Universiy, China. She also earned her MS in Computer Science from Illinois Institute of Technology.

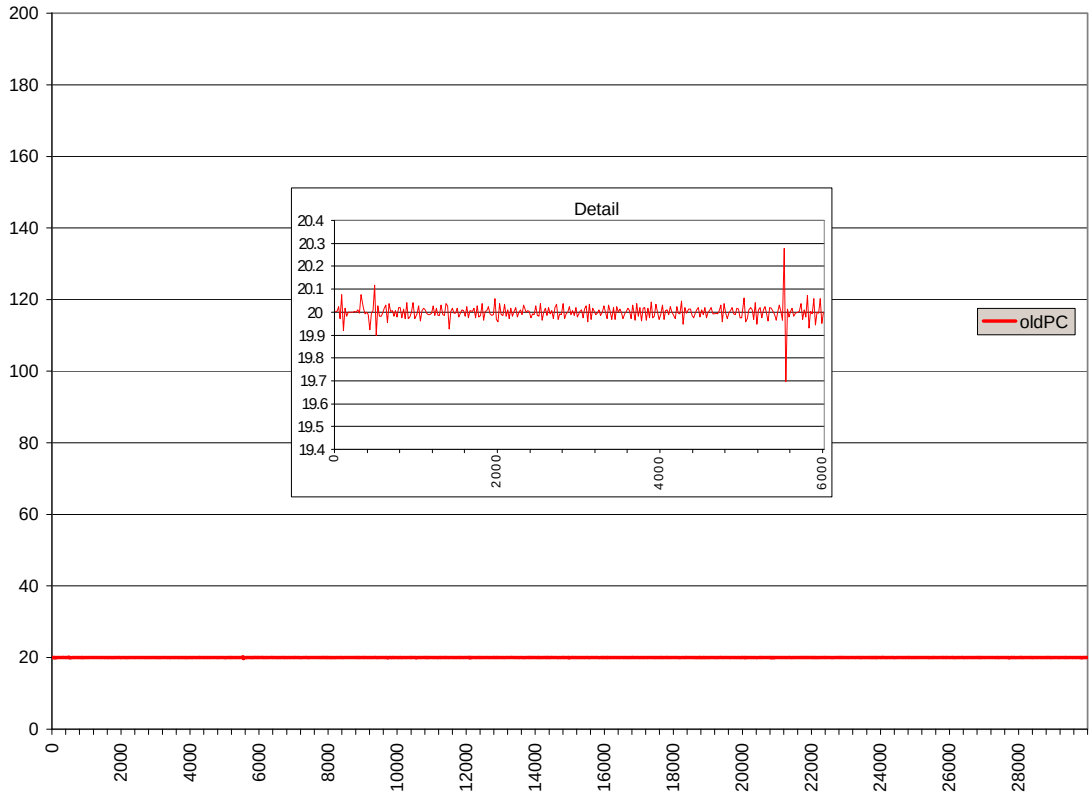


Figure 29: Timing Variation using Elderly PC

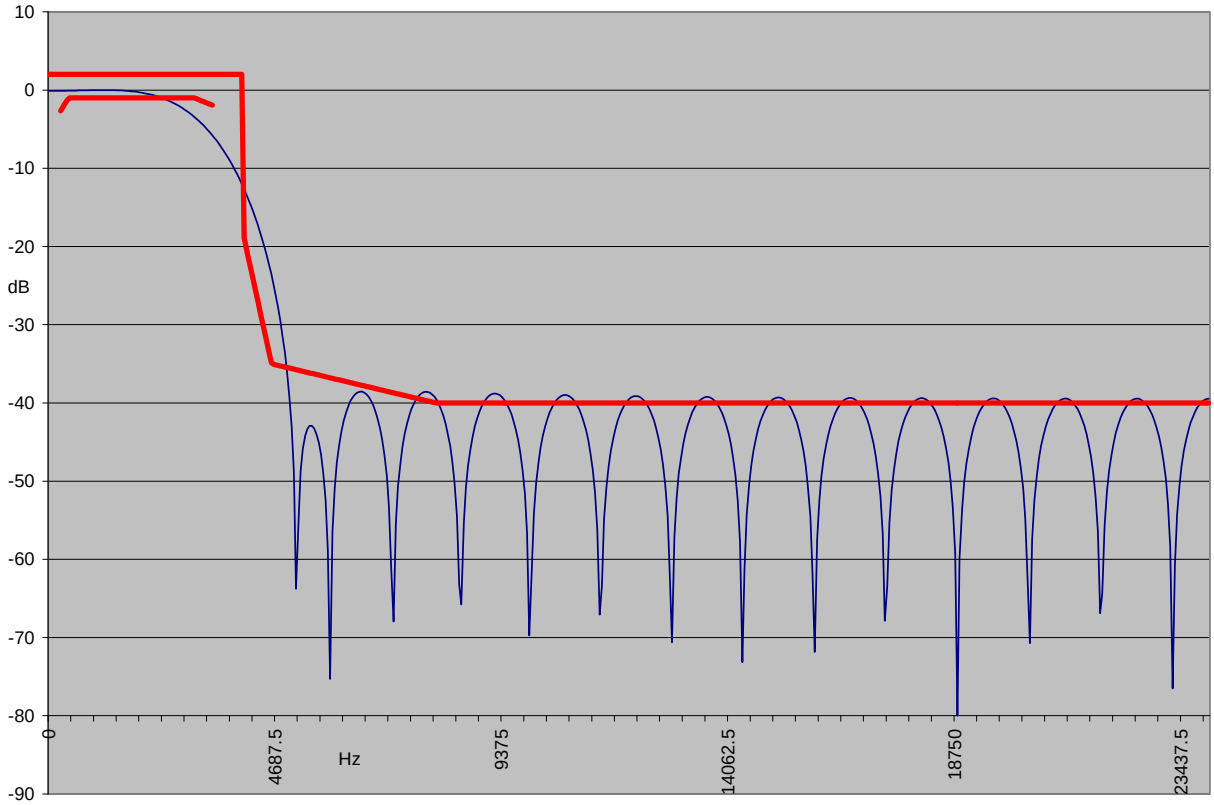


Figure 30: Decimation Filter Frequency Response

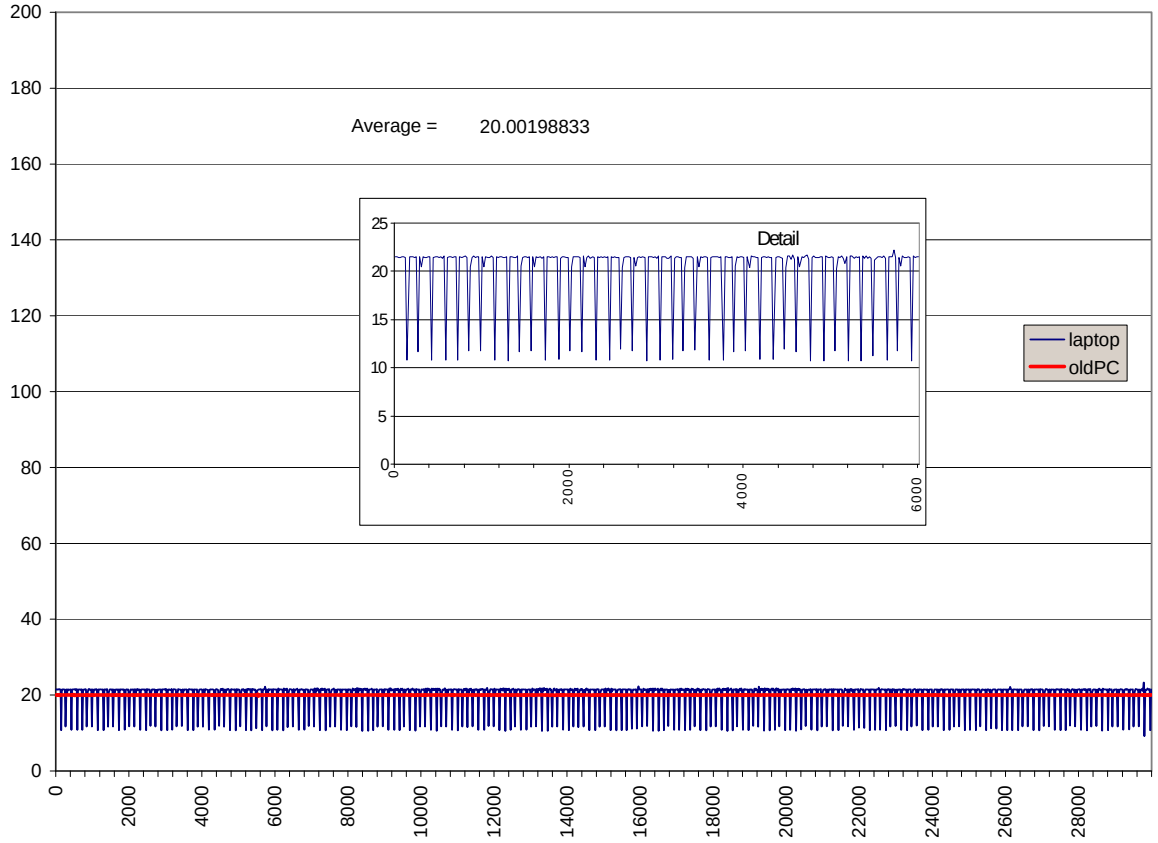


Figure 31: Sample Time Variation of Laptop Computers

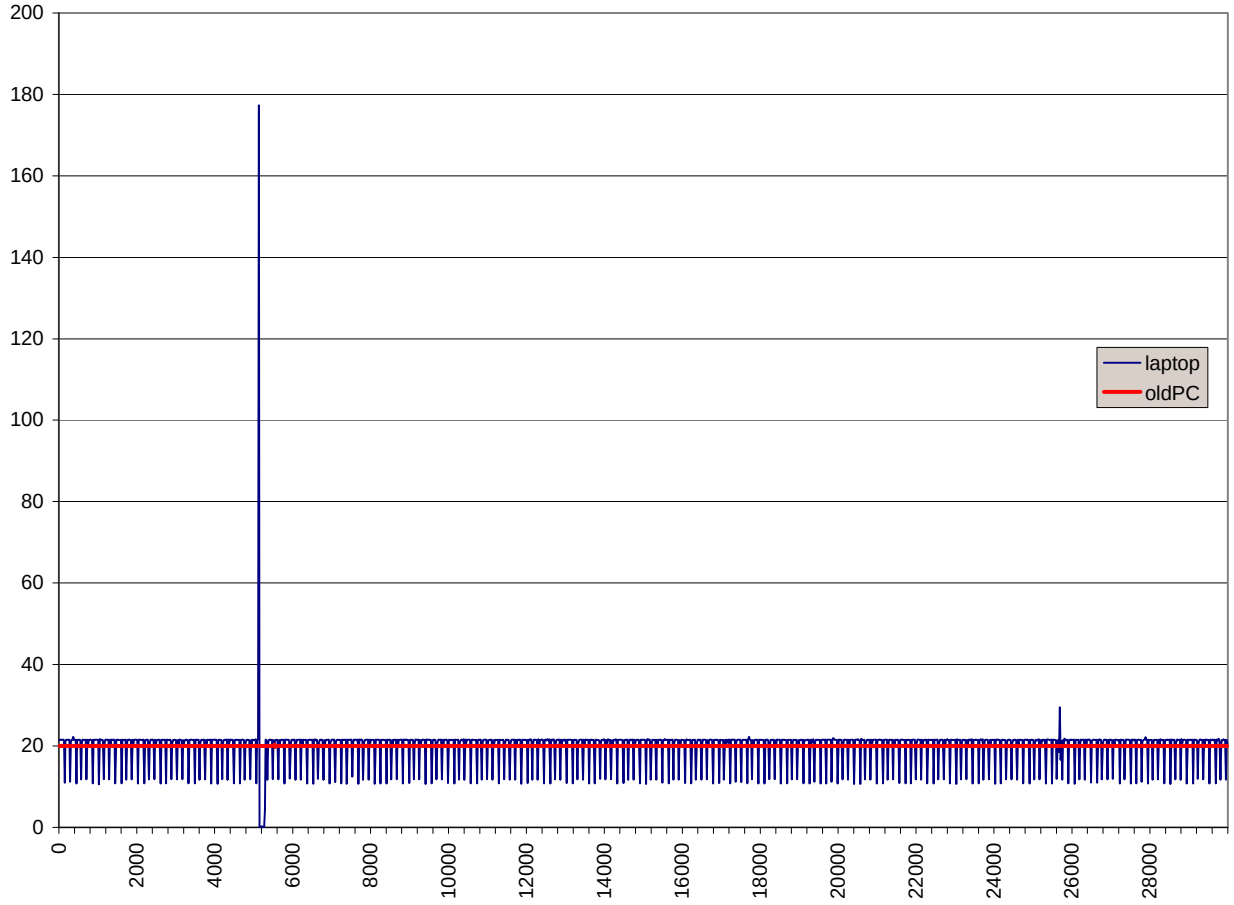


Figure 32: Large Error at start of Reception

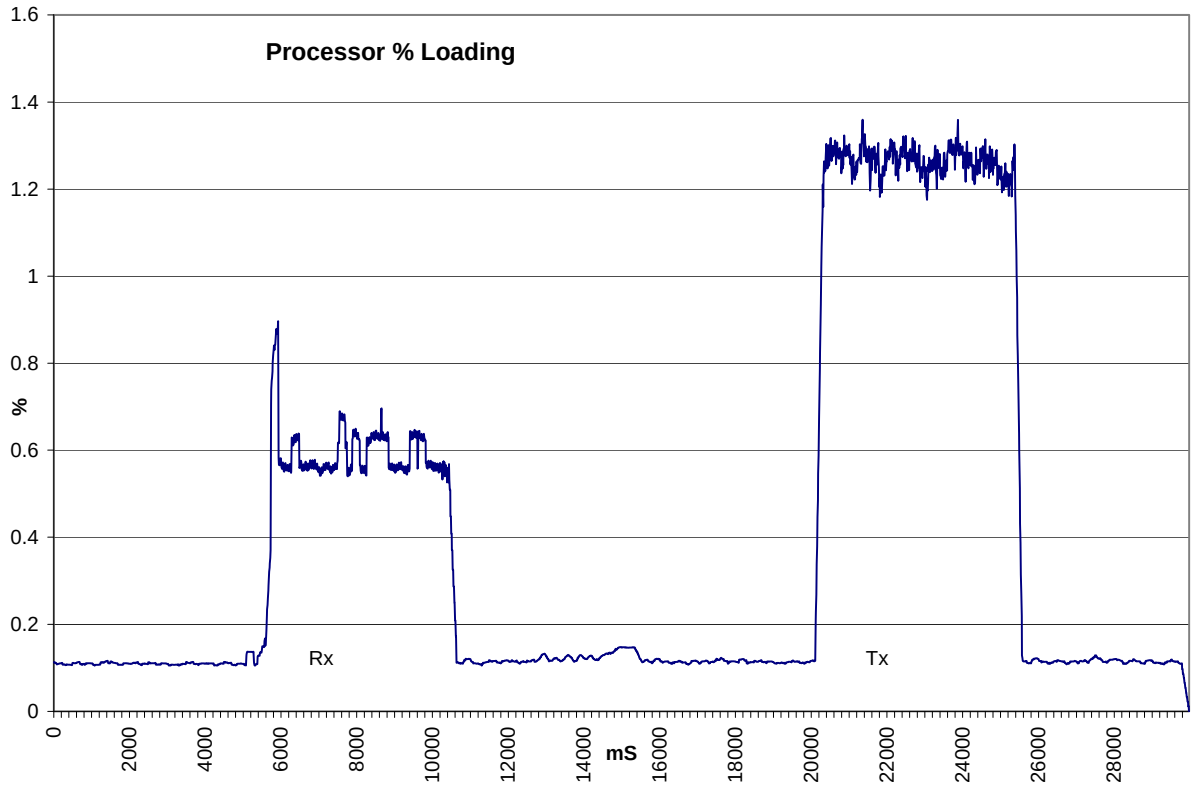


Figure 33: Processor Loading

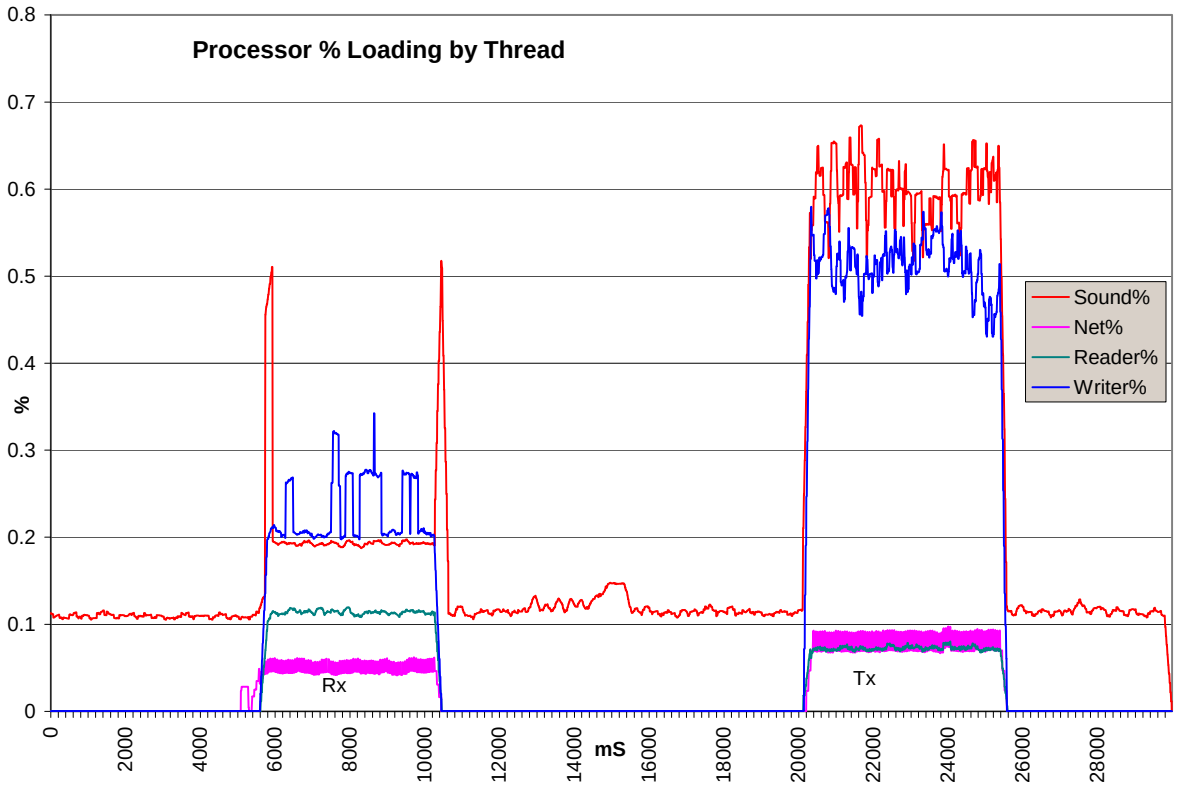


Figure 34: Loading by Thread

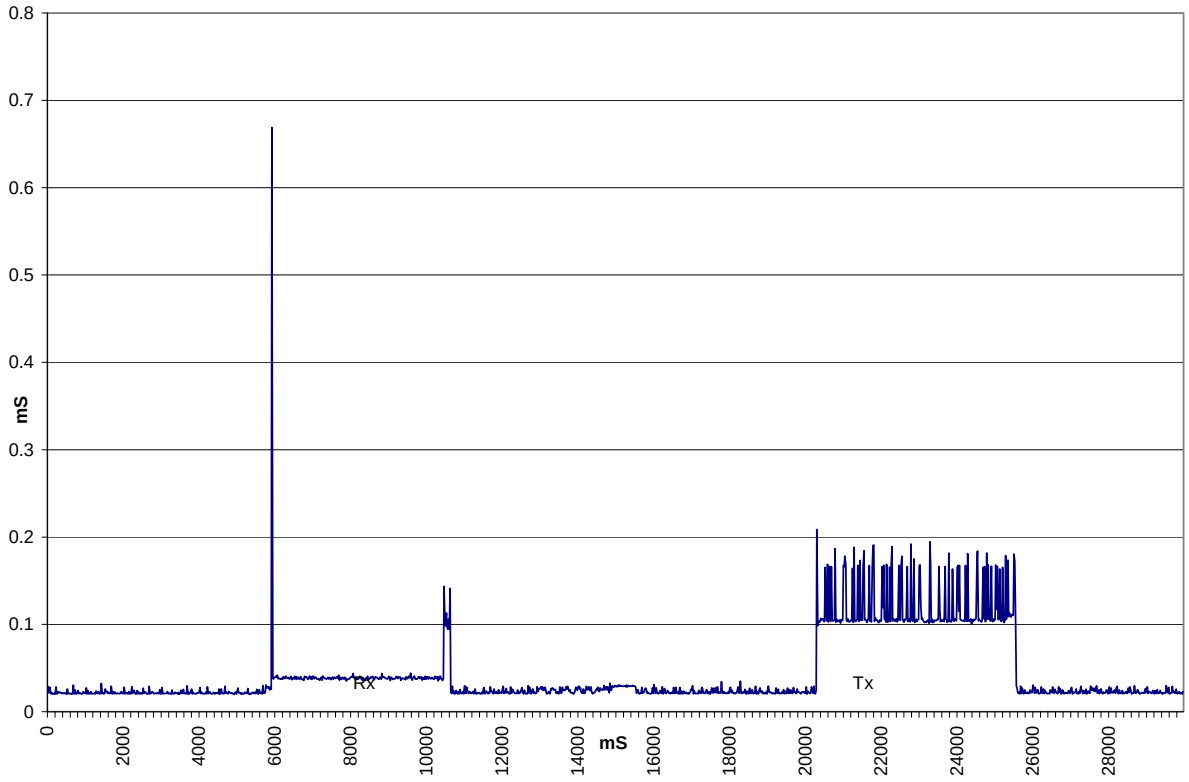


Figure 35: Peak Sound Thread Execution

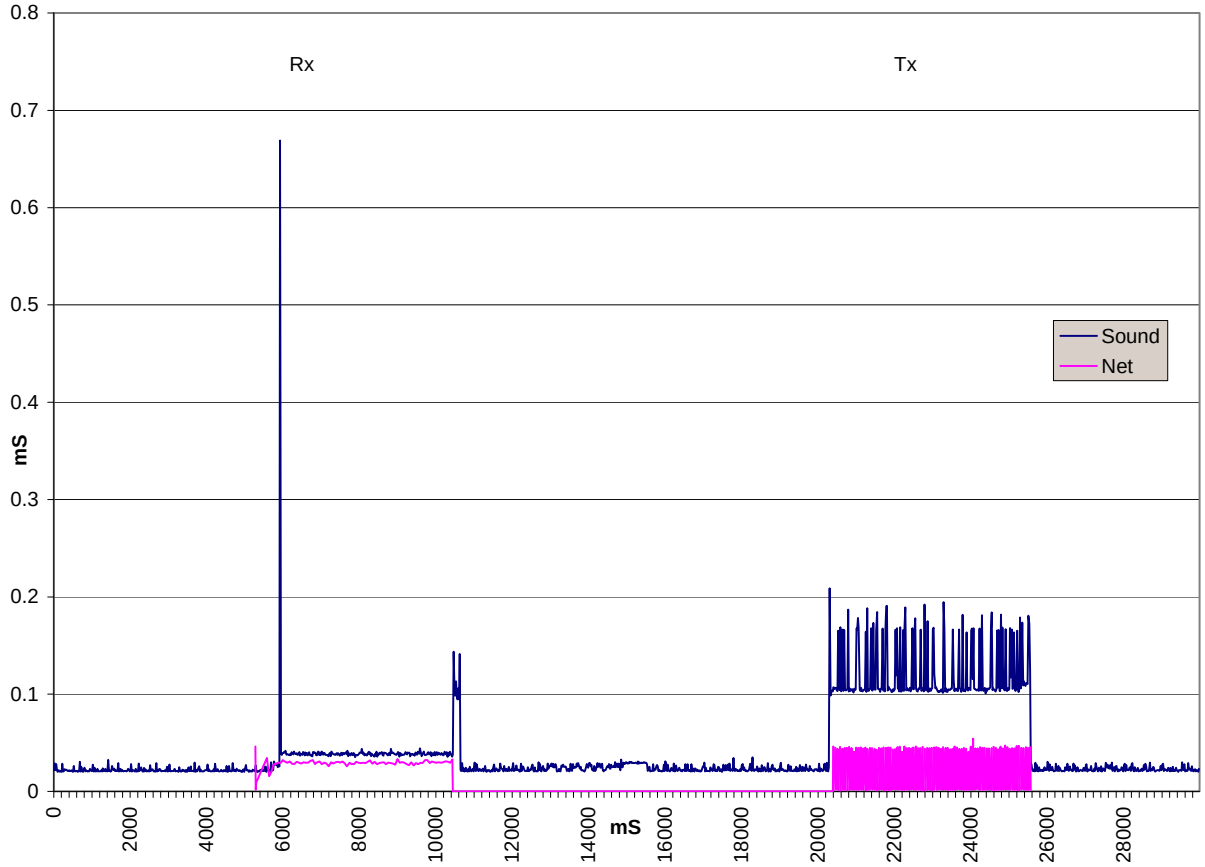


Figure 36: Sound and Network Activity