Chapter 3

Software Analysis, Modification and Expansion

This chapter describes the state of the software at the time I became involved and my contributions to its expansion. It deals primarily with the message handling aspect of the software. It also states the problems encountered with the solutions to these problems.

The deliverables of the project are mentioned at the beginning and intermediary stages of the project.

3.1 Temporal variations of project deliverables

At the inception of the project, it was deemed necessary for the application software to program the WMI to receive, categorize (by priority), and process messages and, finally, schedule messages to be retransmitted out of either of the two serial ports of the WMI. Message handling included placing messages in an INBOUND queue, checking the message for transmission errors (discarding messages with errors), searching for previous transmissions on the history list – (disregarding duplicate messages), copying valid messages to appropriate priority queues and de-queuing messages according to priority for retransmission. Steven C. Franks, using Microtek Research’s Software Development tools coded most of these requirements in C language [7].

At the time I joined the IVDS project, the goals of the repeater project, as assigned to me, were to complete the software started by Franks and to expand its functionality to include interacting with the WMI via one of the serial ports. The extra software features that were required included: position location through the GPS receiver, converting the polled decoder boards strategy to an interrupt driven mechanism, and changing the message structure. Also, elimination of INBOUND queues for PRIORITY7 (highest user priority) messages, capacity to hold 10000 raw messages and sending out the highest priority messages without queuing or maintaining their history list were some of the other modified code aspects.

These modified specifications required moderate changes in the code written by Steven C. Franks [7] and, as my work progressed, in the kernel code written by Robert A. Brickhouse [1] too.

3.2 IVDS Message Format

The format of the message (revised) is shown in Figure 3.1. For the final format, the message protocol was as mentioned below.

♦ The message would be 23 bytes (184 bits) long. It was subdivided as follows:
  ♦ **Bits 1-24**: The first 3 bytes were the lock or sync bytes. A value of 03H was chosen for this field.
♦ **Bits 25-32:** Byte 4 was the header byte. A value of FFH was chosen for this field. The header byte and the sync bytes were chosen to help the receiving UART to synchronize and latch on to the byte boundaries of the incoming data.

♦ **Bits 33-35:** The 3 high order bits of byte 5 would be the message priority (for priorities ranging from 0 (binary 000) to 7 (binary 111)).

♦ **Bits 36-75:** The 5 low order bits of byte 5 along with bytes 6 through 9 and the 3 high order bits of byte 10 (40 bits in all) were reserved for the Silicon Serial Number - a unique number that identifies the WMI.

♦ **Bits 76-115:** The 5 low order bits of byte 10 along with bytes 11 through 14 and the 3 high order bits of byte 15 (40 bits in all) were reserved for the Product Code - a number that identified the product the consumer desired to purchase.

♦ **Bits 116-121:** The 5 low order bits of byte 15 along with the high order bit of byte 16 (6 bits in all) were reserved for the Consumer Code Version Number - and indicated the software version written for the WMI.

♦ **Bits 122-168:** The 7 low order bits of byte 16 followed by bytes 17 through 21 (47 bits in all) were reserved for the Consumer Code Identification number - a number that was unique to each consumer.

♦ **Bits 169-184:** Finally, bytes 22 and 23 were reserved for the Cyclic Redundancy Code (CRC) calculated on bytes 5 through 21 of the message - these significant bytes included the Priority, Silicon Serial Number, Product Code, Consumer Code Version and excluded the Consumer Code Identification number.

![Figure 3.1 Sample IVDS Message](image-url)
3.3 Historical queuing structures

Before I joined the IVDS project, the message storage and retrieval was implemented through software queues. Memory allocation was dynamic on an as-needed basis. Messages that entered the repeater unit were ultimately received by the motherboard as two message "halves." This was due to the inherent overhead of the protocol established between the 6805s and the 68306 (refer Section 2.2.2). Upon validation, messages were placed into INBOUND queues. From here, messages were moved to the PRIORITY and HISTORY queues based on the message priority and duplicity of messages.

HISTORY queues helped to discard duplicate messages. Messages were sent from the PRIORITY queues to the OUTBOUND queues where they were transmitted out to a PC terminal or the handheld display unit [7].

3.3.1 Modification needed to Franks’ queuing structure

After I began work on the project, some of the older queuing strategy was redefined. Most of the queuing structure prepared by Franks remained the same in functionality. The form changed as described below:

1. Valid messages with priority 7 (highest) were always retransmitted out of the repeater unit as soon as they were received. There were no PRIORITY or HISTORY queues associated with this message priority. Duplicate messages with priority 7 were not discarded. It was assumed that lost messages with priority 7 would be handled by the retransmission scheme at the transceiver end.

2. Messages were received in full, rather than in message halves. This was accomplished by using bit stuffing at the transmitter and bit "unstuffing" at the receiver (see Section 3.9).

3.4 Modification needed in kernel code

To have the GPS location of the unit integrated into the message structure, the GPS kernel routines were reviewed. The GPS kernel routines needed some modification to tune the kernel to decipher specific format messages.

The 68306 polls the 6805s to check for received messages. There are instances when the application software requires substantial processing time for message handling. It is possible during this time for the 6805 buffers to be either overwritten or for the existing messages to be lost. One of the project requirements was to absolutely not lose messages transmitted from the audio link. This suggested an interrupt driven scheme where a 6805 indicates to the 68306 via an interrupt source that the 6805 message buffer is full. The 68306 then responds to the requesting 6805 by extracting the message in the 6805 so that the 6805 then prepares to receive a new message.

Receiving messages on an interrupt basis implies that during heavy transmission traffic, all messages should be received and processed by the 68306. This means that there is a need for a buffer to store all the rapidly incoming messages while they were being processed. At one point, it was decided to have the capability to store
10,000 messages. This was gauged by the sponsor to be several (10 to 20) times more
than the expected customer "worst-case" response during peak seasons for sales, sports
e tc. This capability was coded and successfully tested in the kernel.

These features required that the system variables and the kernel code be
modified to suit the application needs.

3.5 GPS software – kernel expansion

GPS capability was needed in the system and thus was incorporated into the
IVDS software. It was more convenient from a code point of view to place this section of
code in the kernel GPS routines. A brief discussion on some GPS formats follows:

3.5.1 GPS sentence formats

GPS information was transmitted in the form of coded GPS "sentences." Each
sentence had varied information such as latitude, longitude, altitude, time etc. A field
separator (a comma) separated each field.

The kernel recognized two types of formats - NMEA (National Marine
Electronics Association) and TAIP (Trimble ASCII Interface Protocol). The kernel was
modified to choose the NMEA mode as the default mode for the GPS receiver. This
format and its sentence structure are discussed in detail below [3].

3.5.1.1 NMEA format: The NMEA sentence format began with a “$” symbol. It was
followed by a two letter “talker” (talkers are transmitters, listeners are receivers)
mnemonic (e.g., LC, GP etc.), a three letter sentence code (GLL, GGA, etc.), some data
fields and was terminated by a <CR><LF> (Carriage Return – Line Feed combination).
The maximum number of characters was 80, including the “$” and the <CR><LF>.

A sample NMEA sentence follows:

$LCGLL, 4001.74,N, 07409.43,W<CR><LF>

In the above sentence,

$ => Start of NMEA sentence
LC => Loran-C receiver
GLL => Present position in Latitude/Longitude
4001.74,N => Receiver’s latitude was 40 degrees 1.74 minutes, North
07409.43,W => Receiver’s longitude was 74 degrees, 9.43 minutes, West

GLL data was always displayed in the above format.

Since GPS sentences begin with a talker mnemonic (such as "GP" for GPS
receiver), the WMI was programmed to listen to "GP" sentences. In particular, the WMI
could listen to sentences that began with “$GPGGA.”
An example of such a sentence follows:

$GPGGA,170834,4124.8963,N,08151.6838,W,1,05,1.5,280.2,M,-34.0,M,- ,*75

Here,

$GPGGA => GPS fix data
170834 => Universal Coordinated Time (or GMT) is 17:08:34 or 5:08:34 PM.
4124.8963,N => Receiver’s latitude was 41 degrees, 24.8963 minutes, North
08151.6838,W => Receiver’s longitude was 81 degrees, 51.6838 minutes,
West
1 => Fix quality
05 => Number of satellites in view
1.5 => Horizontal dilution of precision (HDOP)
280.2,M => Receiver’s altitude above sea level (in Meters)
*75 => Checksum

In order for positional data from the satellites to be valid, the GPS receiver needed to get data from at least 4 satellites. The quality (or validity, in this case) of data was monitored by the “Fix quality” data field which was a ‘0’ for invalid data (fewer than 4 satellites in view) and ‘1’ for valid data in normal mode (4 or more satellites were in view).

The software that processed GPS data checked for both the fix quality as well as the number of satellites in view before accepting the received data. The code specifically extracted only the latitude and longitude information from the GPGGA sentence format. If this information did not change over a period of time, blank fields were transmitted where the data remained constant. For example, after the above sentence was repeated, say for 10 minutes, and the position of the GPS receiver did not change, then the transmitted sentences would resemble the following format:

$GPGGA,171834,,,,,1,05,,,,,,,*82

From the above it was observed that the latitude and longitude information was not being transmitted any more because the receiver did not change its position in a while. The data was still valid and there were still 5 satellites in view of the receiver, as can be studied from the values of the “fix quality”, and “number of satellites in view” fields.

The data that was transmitted from the WMI to the display devices contained the latitude and longitude as part of the packet. If no valid information was obtained (fix quality = 0), then the diagnostic “No fix” was displayed in place of the latitude/longitude fields.

### 3.6 Message structure modification

The structure of the IVDS messages underwent minor changes. The Product Code (40 bit wide) field and the Customer Code (47 bits wide) fields were swapped. This was done to help the transceiver team with their software algorithms.

### 3.7 Message handling and queuing

IVDS messages could be categorized according to their priority field (which is 3 bits wide). Priority7 was the highest message priority and Priority0 was the lowest message priority. Priority7 messages were treated differently from messages with lower priorities.

#### 3.6.1 Priority7 message handling

Priority7 messages were validated, by checking the CRC bytes, as soon as they were received. Upon receiving good messages, the messages were time stamped. Time stamping was achieved by reading the value of the real time clock (RTC) on board the WMI. These messages were not queued up for scheduling at a later time. They were
directly retransmitted out of the serial port of the WMI. Also duplicate messages received were not discarded under this priority. This was chosen to let priority 7 be used in specific applications like emergency, system testing, etc.

3.6.2 Non-Priority7 message handling

Messages that have priority less than 7 were validated using the CRC algorithm as soon as they were received. Once validated, messages were time stamped (similar to the priority7 messages) and placed in appropriately titled PRIORITY queues. There was a priority queue for each of the different message priorities (0 through 6, excluding 7, as mentioned in the previous paragraph). There was also a HISTORY queue for each priority (0 through 6, excluding 7). This queue held a copy of every new message as it was received. If the message received had, say, priority 5, then it was compared for duplicity in the HISTORY5 queue. If no duplicate was found, then it was stored in the HISTORY5 queue as a new message. If there already was an identical message in the HISTORY5 queue, then the received message was assumed to be a duplicate of what has already been received and was thus discarded. The messages in the HISTORY queues were discarded after being held for 60 seconds. This interval made sure that duplicate messages received within a 60-second window were rejected after the first correctly received copy of the message was place in the queue. It must be borne in mind that when the audio link transmitted messages, it did so in bursts, hence only the first valid message received was preserved.

Messages in the PRIORITY queues were retransmitted every 5 minutes. Changing the appropriate constants in the header files could vary this retransmit interval. It was a good practice to have this activity as frequently as possible so as to preserve memory space. At the same time, re-transmitting messages from the WMI as often as possible could increase latency in the system. This was because the transmission of messages via the serial ports was a slow process due to the kernel software architecture.

3.8 Message validation

A cyclic redundancy check (CRC) algorithm was used as part of the message stream to validate incoming IVDS messages. More details on this algorithm can be found in Matt Kurtin's thesis [10]. This algorithm helped minimize errors in the message due to noise.

3.9 Bit stuffing and decoding messages

The 68306 accessed the 6805s to retrieve the messages received by the radios on the WMI. The protocol between these two microprocessors proved to be a minor hindrance. Due to the way the protocol was implemented internally, the 23-byte long IVDS message was transmitted to the 68306 in two unequal message “halves.” The first “half” was 11 bytes long and the second “half” was 12 bytes long. The reason the IVDS message was transmitted in two message parts was because the 68306 to 6805 protocol required the transmission of certain bytes to be preceded by a NULL byte. For e.g., to transmit the SOH command byte as a data byte, the SOH byte should be preceded by a defined NULL byte. Also since the message buffer size for each 6805 was limited, the IVDS message had to be sent in two parts. While this did not pose a problem in the initial
stages of the project, it did prove to be a bottleneck during the later stages. When the robustness of the repeater unit was being tested, it was found that, during heavy incoming message traffic, a 6805 could efficiently receive all incoming bytes without losing any data. It could even buffer up to a certain number of data bytes (limited to approximately 1000 bytes).

In order for the 6805 to “transmit” messages to the 68306, it needed to be polled by the main processor. Receiving messages in two message “halves” slowed down the rate at which messages could be removed from the 6805s. This problem was overcome by retrieving the data packets in their entirety rather than in two message parts. In order to do so, the in-built protocol between the 6805 and the 68306 had to be fooled. This was achieved by using a technique called as bit stuffing.

Bit stuffing was accomplished at both the ends of the repeater unit -- 6805 and the 68306. The incoming 23 bytes (184 bits) were split into 26 sets of 7 bits each and one set having only 2 bits. To each set of 7-bits a “1” was appended on as its most significant bit, making it an 8-bit number whose value was always greater than or equal to 80H. The last set of 2 bits was also converted into an 8-bit number by setting its most significant bit as “1” and the second through sixth most significant bits as “0”. For e.g.,

Received data stream…

40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 55 56
which is the same as…
01000000 01000001 01000010 01000011 … 01010100 01010101 01010110
when broken into sets of 7-bits each…
01000000 01000001 01000010 … 01010100 01010101 01010110
↑ ↑ ↑ ↑ ↑ ↑ ↑
can be re-arranged as…
0100000 0010000 0101000 … 0100110 1010100 0101010 1010101 10
which, after bit stuffing…
10100000 10100000 10100000 10100000 … 11010100 11010100 11010100 11010100 10000010
now becomes…
A0 90 A8 A4 9A 91 8A C6 A3 D2 89 D2 AD 98 CD A7 93 EA 85 8A C9 A6 D4 AA D5 82

At the other end, the 68306 decodes or performs bit “unstuffing” on the new bit stuffed message to retrieve the original information bytes received by the 6805 [8]. Thus, bit stuffing reduced the bottleneck between the 6805 and the 68306.

3.10 Message volume capacity in WMI

Since the queues essentially were pointers to the copies of the messages received, the amount of messages that could be dynamically stored depended on the RAM available (1MB RAM was available on the ICB).

At a later point in the project it was suggested that during heavy message traffic there should be in the WMI the capacity to store statically about 10,000 IVDS messages. This was directly incorporated into the kernel routines that dealt with processing the decoder units. The 1MB RAM space helped achieve this memory requirement.
3.11 Message retransmission from WMI

Messages were retransmitted from the WMI after being validated and queued at fixed retransmission intervals. Retransmission occurs when there were too many messages waiting to be dispatched in the OUTBOUND queue, when priority 7 messages were received or every message transmission period, whichever happened first. This period was variable and user adjustable.

Initially, messages were retransmitted according to a binary schedule mechanism, which involved scheduling $2^\text{priority}$ messages to the OUTBOUND queues for transmission out of the WMI. This made sure that messages with highest priorities were retransmitted in larger volumes than the lower priority messages.

For example, when there were 100 Priority7 messages, 100 Priority6 messages and no other priority messages in the other PRIORITY queues, all the Priority7 messages would be retransmitted (since 100 is less than $2^7 = 128$), followed by 64 ($2^6$) Priority6 messages during the first retransmission period. On the second retransmission period, the rest of the 36 Priority6 messages would be sent out of the WMI (assuming no more Priority7 messages were received between these two retransmissions).

Let’s consider a different scenario – if there were 150 Priority7 messages and 50 Priority6 messages, there would be two transmissions to flush out all the messages. The first transmission would send out 128 Priority7 messages and all Priority6 messages. The second transmission would transmit the rest of the 22 Priority7 messages. It was seen that Priority7 messages were being treated unfairly compared to the Priority6 messages. This meant that consumers who paid more for choosing Priority7 messages were at a disadvantage compared to those who paid lesser for choosing Priority6 messages based on their time of response. To resolve this situation, the retransmission algorithm was reworked.

The new strategy involved re-transmitting all the messages in a given PRIORITY queue before moving on to the next lower priority queue. This did not apply to the Priority7 queue, however. All Priority7 messages were validated and immediately retransmitted without being stored in HISTORY queues. Every message transmission period, all the messages in the PRIORITY6 queue would be retransmitted first, followed by all the messages in the PRIORITY5 queue, followed by all Priority4 messages, etc.

This scheme of retransmission worked truly in the favor of descending priority.

3.12 Message retrieval from decoders

Originally, the four decoders in the WMI were polled sequentially to retrieve messages from each decoder. Each decoder’s messages were stored in separate decoder buffers [1]. Since it was not important as to which decoder the messages came from, it was not necessary to have four separate buffers for each decoder. This part of the kernel was restructured so that all incoming messages were stored only in one buffer, regardless of the decoder the message came from.

Initially, the four decoders were being polled by the 68306. During heavy message traffic with all the four decoders receiving messages, if the polling was not frequent enough, the message bytes would be overwritten inside the 6805 buffers. This happened despite the circular buffer coded into the 6805s. Retrieving messages from the 6805s through an interrupt driven mechanism rather than a polled mechanism
circumvented this issue. The interrupt driven mechanism was, however, not a true interrupt process. During a true interrupt process, the slave device notified its master through an interrupt port line that it had data to transfer to the master. Upon receiving this interrupt call, the master typically stopped whatever it was doing, depending on the priority of the interrupt, and processed the slave by retrieving its data. From this, it was seen that for a true interrupt mechanism, the hardware on the WMI would need to be modified. This was not an option.

To mimic such a process, I utilized a software programmable interrupt source that generated interrupts on a periodic basis. The period of the interrupt could be programmed as per convenience. Whenever this interrupt triggered the 68306, an ISR (Interrupt Service Routine) would be executed which vectored the code to process the decoders in a polling sequence. This way, by increasing the frequency of polling the decoders, it was ensured that no messages were lost even during heavy traffic. Selecting the interrupt period was more experimental. It would be ideal to have it as short as possible, so as not to lose any messages. If the period selected was too short, most of the time would be spent polling the decoders, making the message handling functions take up most of the CPU time. This period is currently set at 50ms.

3.13 Software interrupts on the WMI

In order to use the programmable interrupts to poll the decoders, I studied the interrupt registers and options available on the 68306 [9]. The kernel code needed modification to incorporate an additional interrupt source. The 68306 microprocessor has an independent timer that can be programmed to generate interrupts.

To test the timer generated interrupts, I wrote some test code that would flash a cursor on the handheld display unit. This test code crashed whenever the interrupt service routine was entered. After some effort, it was realized that the timer was already being used to process one of the serial ports to talk to the monitor program. The chosen baud rate to talk to the monitor was 57.6Kbaud [1]. At this baud, it was necessary to use an interrupt driven source to process the serial port on the WMI. Hence, the timer could not be used for any other purposes, unless it was freed up.

Freeing up the timer meant building monitor code that could talk at slower baud rates. This was permissible since shifting to a lower baud did not affect the system performance. Thus, fresh monitor code was compiled to work at 38.4Kbaud. Now the timer could be used to generate programmable timer interrupts. Using the timer to generate interrupts did not seem to be as trivial as I expected. No amount of effort to get the timer to generate interrupts was successful.

Looking for other implementation methods, I realized that the timer could be used as a programmable down counter that could be pre-loaded with a preset. After initialization, this down counter generates an interrupt when the count reaches zero. It then jumps to an ISR to identify the interrupting source. Since there is already a SERIAL_ISR routine in the SERIALIO.C file, I modified part of this routine to check for a counter interrupt. If the interrupt source is indeed the counter, the counter is stopped, the four decoders are polled to retrieve messages, the counter is pre-loaded with the same preset again, and the counter is re-started before exiting the service routine. This preset is
chosen so that an interrupt is generated once every 50ms as discussed in the previous section.

While the counter alternative worked as intended by its application, it still remained a mystery as to why the timer could not be used as is to generate interrupts. This remained even after having contacted Motorola’s Tech Support.

3.14 Where do we go from here?

The obstacles so far had been successfully surmounted. None of the transmitted messages were lost. All messages were received, queued and retransmitted to a temporary display unit for debugging purposes. The next part of the project was envisioned by the sponsor as retransmitting messages to another computer with an IP address through the Internet. This required me to have a thorough understanding of the protocols I would need to follow in order to establish communication through an Internet connection.

What was clearly needed was for the WMI to interface to a modem and dial-up an ISP (Internet Service Provider), get a valid connection, login with a username and password, and establish communications with the peer (ISP computer).

3.15 Serial port capabilities

The two serial ports on the WMI could be configured as dedicated or general-purpose ports. During developmental stages, one port was used to talk to the PC controlling the WMI and the other was used to send test data to another PC "dummy" terminal. The proposal was to use one of the ports to talk to remote hosts (through a dial-up modem). The hardware schematic of these ports shows that some of the communication lines were hard-wired. These signals, if available, would help provide better capability while talking to the ISP. The baud rate used to talk to these ports was 19.6Kbaud.

3.16 Understanding modem communication signals

The RS232 interface cable was designed with a computer-modem interface in mind. The computer (referred to as the Data Terminal Equipment or the DTE) talks to the modem (referred to as the Data Communication Equipment or the DCE) using different flow control mechanisms. A flow control mechanism is used to regulate the flow of incoming and outgoing data to minimize loss of data and optimize communication speed. In general, there are two types of flow control schemes - software flow control (also known as XON/XOFF) and hardware flow control (also known as RTS/CTS). Though software is needed in both these schemes, the primary difference between the two mechanisms is the method of implementation.

Software flow control (Figure 3.2) needs no more than 3 wires of the RS232 cable to be implemented. When the receiver's buffer is full, the receiver sends a XOFF command to the transmitter. The transmitter interprets this command as an indication to stop transmitting data, until the receiver is again able to handle more incoming data. When the receiver is ready to receive data again, it sends a XON command to the transmitter. This is the cheapest method of communicating between two devices. The three wires needed for serial communication are Rx, Tx, and signal ground. One
disadvantage of using this method is if the data link is noisy and the XON/XOFF commands are lost and not received by the transmitter, the receiving software ignores any data that comes in while the receive buffer is full. Another disadvantage is when the transmitter needs to transmit the XON/XOFF bits as data and not as commands. The software needs to mask these bytes and this adds extra overhead on the protocol used. XON/XOFF are defined ASCII bytes specially reserved for the purpose of software flow control.

![Software flow control diagram](image)

**Figure 3.2: Software flow control**

Hardware flow control uses additional RS232 lines to communicate. The basic level of hardware flow control (Figure 3.3) uses the RTS (Request-To-Send) and CTS (Clear-To-Send) lines. For the DTE, RTS is an output indicating that the computer is ready to send data. Also the CTS is an input line for the DTE. A logic high level on the CTS inputs indicates to the DTE that the communicating equipment is ready to accept data. The DCE receives the incoming RTS signal on its CTS line. If the DCE is ready to receive data it responds to the DTE’s request on its RTS output line (connected to the CTS input of the DTE).

The next level of hardware flow control (Figure 3.4) uses the DTR (Data Terminal Ready) and DSR (Data Set Ready) lines of the RS232 cable. An active DTR signal indicates that the DTE is ready to transmit data while the DSR (for a DCE) signal indicates that the DCE is ready to receive data. This is used when, for example, the null modem ties the RTS output to the CTS input in a loop-back mechanism. The null modem is a serial communication interface that makes the DTE look like a DCE. For example, in order to connect a PC to a serial communication device (like another PC), it is not
necessary to use a modem. Using the null modem creates a loop-back mechanism so that each PC is made to believe that the PC is communicating to a modem.

It is also possible to use the CD (Carrier Detect) line of the RS232 cable to support the hardware flow control scheme. This is not a status signal, i.e., it does not indicate whether the DTE or DCE is ready to communicate. It only informs the requesting unit that the opposite unit is healthy (has a carrier and is operating well).

![Diagram](image)

**Figure 3.3: Hardware flow control using RTS/CTS**

![Diagram](image)

**Figure 3.4: Hardware flow control using DTR/DSR**
Depending on the flow control mechanism and the hand-shaking method used, the null-modem can be chosen or custom-configured to successfully make two devices communicate serially. Handshaking is the protocol used when two devices communicate serially. When only the Rx, Tx and Signal ground lines are used (Figure 3.2), there is no handshaking present (as in the case of simple software flow control). When the RTS/CTS, DTR/DSR and CD lines are internally fed back (Figure 3.5), handshaking with feedback is in action. When RTS/CTS are shorted to the CD line at both ends, and DTR/DSR are actively used partial handshaking (Figure 3.6) is effective. Finally, when all signals (Figure 3.7) of the cable (CD may or may not be used) are used at both ends in the software, full handshaking is being used.

When a PC is used initially to talk to the ISP, a simple RS232 cable with no null modem Is utilized to connect to a modem. This is sufficient since the PC and the modem were configured to use full handshaking. However, with the WMI taking the role of the PC, there was a need to use a null modem. The WMI had its hardware flow control signals internally hardwired. This limited the WMI system from using any kind of hardware handshaking. Since the modem software was expecting to detect the DTR/DSR signals, a null modem was used. The null modem re-routed the DSR signal from the modem back on to its DTR line (loop back mechanism). This ensured that the modem and the WMI were receiving all signals necessary to communicate with each other. This also restricted the WMI-modem system to using only software flow control. Using software flow control increased the protocol overhead and resulted in decreased bandwidth between the WMI and the outside world.

It was clear to see that for the WMI to communicate to a remote host system, the WMI would need to be connected using a null modem to a modem. The modem would then dial the ISP (Internet Service Provider) and establish communications with the modem on the ISP. Once this communication was established, the data sent by the ISP would be re-channeled to the WMI.

Figure 3.5: Feedback handshaking implementation
3.17 Need for Point to point protocol (PPP)

Talking to another computer with an IP address required that the WMI communicate to the peer computer (ISP) using a common language tool that the peer understood. The language that is used as a standard between any two computers on the Internet is the point-to-point protocol, also known as PPP. I realized that I needed to implement the PPP on the WMI to send messages to the Host subsystem.
3.18 Summary

The changes needed for message retransmission, formatting, queuing, GPS software etc. were successfully completed. At various stages, the software was tested with various types of messages, volumes, locations, etc. to ensure robustness and functionality.

The software functionality was also demonstrated to the other teams, the sponsor and the faculty at different stages. The performance was accepted and suggestions were worked upon.