Chapter 2
Repeater Unit Hardware and Software

2.1 Introduction

This chapter focuses on the hardware and software features of the Wireless Measurement Instrument. It provides the strong foothold that is needed before proceeding into the subsequent chapters. The chapter also elucidates how the WMI™ meets the needs of the IVDS system. Instances where the WMI™ falls short of the IVDS system requirements as well as workarounds and possible solutions are dealt with. The features of the WMI™ listed in this chapter are detailed in the WMI™ reference manual provided by Grayson Electronics® [1].

Henceforth, the terms “WMI™,” "repeater unit," "repeater," and "repeater box" will be used interchangeably to refer to Grayson Electronics® Wireless Measurement Instrument™ unless explicitly mentioned. Also, the term "Grayson" will be used to refer to Grayson Electronics Incorporated®.

2.2 Hardware components of the Wireless Measurement Instrument

A brief introduction of the repeater unit was presented in Chapter 1. This section will discuss the finer aspects of the repeater box.

The backbone of the WMI was called the Interface Control Board (ICB). This motherboard interacted with all the other peripheral devices (like the wireless receiver units, the decoders, the GPS system, etc.) that physically reside inside the WMI. The key aspects of the ICB were the main microprocessor (CPU or central processing unit), memory, ICB bus, ICB backplane and the special purpose devices. Figure 2.1 puts the block diagram of the ICB. The ICB components are elaborated in the following sections.

2.2.1 The CPU Section

This section talks about the core internal modules of the 68306 CPU and explains how these modules are implemented on the WMI.

Overview

The central processing unit (CPU) on the ICB was the Motorola 68306 microprocessor. It was responsible for all the data processing of the peripherals, executing the application and monitor software, etc. Some of the tools that characterized this CPU are explained below:

(a) EC000 core: The 68306 had an EC000 core and belonged to the MC68EC000 family of processors. This meant that the application code was compatible with the MC68000 processor. The addressable space for this microcontroller was 4GB and the peak performance speed of the processor was 2.4MIPS as per its specifications.

(b) UARTs: The 68306 had two independent Universal Asynchronous Receiver Transmitters on board. This allowed the WMI™ to communicate with serial Electronics Industry Association (EIA or RS232C) ports with modem control. Two accompanying baud rate
generators provided a means of varying communication speed for the two serial ports.

(c) Chip selects: Eight programmable chip selects were present to interface the CPU to other microprocessors or master/slave devices for a multiprocessor environment.

(d) Interrupt handlers: There was a programmable interrupt controller that could be handled in software to manage and control interrupt processes flexibly.

(e) Address/data lines: The 68306 processor had 24 address lines and 16 data lines. This enhanced its performance when it talked to the various microcontrollers that controlled the peripherals.

(f) Crystal frequency: The maximum crystal frequency was 16.67MHz. This was further subdivided to provide an internal clock for the instruction cycles and other subsystems (like the serial and parallel port modules, for example).

Implementation

(a) Data lines: The data lines for parallel port communication were implemented using Port A of the CPU.

(b) IRQ4: The IRQ4 pin was connected to a test point on the ICB. This was used, during software development, as a hardware break point. Shorting this test point to circuit common on the ICB triggered an interrupt to the processor, halting software execution.

(c) Both the UARTs were buffered through MAX232 chips to provide external EIA ports.

2.2.2 The Memory Section

This section discusses the memory devices present on the ICB motherboard. The memory devices on-board the ICB motherboard were:

2.2.2.1 EEPROM

The ICB had 256KB of EEPROM (Electrically Erasable Programmable ROM). This was implemented using two chips of 128KB each in DIP packages. These EEPROM devices were UV-erasable and re-programmable. A chip-select line from the 68306 would select these devices and the data from them would be read out through strobe signals. These EEPROMs were socketed and labeled as U20 and U21 on the ICB. They were physically below the GPS receiver. The EEPROMs contained the code required to boot the ICB, the main application software routines, the kernel support code, and the debugging monitor code. The EEPROMs were programmed with the odd-number address bytes in one EEPROM and the even-number address bytes in the other EEPROM. The MCU read from both the chips to form a single 16-bit word.
2.2.2.2 RAM

The ICB had 1MB of RAM (Random Access Memory). This was implemented as four blocks of 256KB. Each block was further sub-divided into 2 chips of 128K each. Chip select line CS1* enabled each RAM chip. Each of the 256K blocks was decoded using a DS1211 two-to-four decoder with address lines A18-A19 as select lines. The DS1211 also provided automatic switching to battery backup for all RAM chips when the main supply power was either removed from the ICB or fell below a threshold. When the battery backup was being used the DS1211 disabled its entire chip select line.

The RAM was used to hold a version of the kernel software (beneficial when the EPROM based kernel was faulty), as a backup. It also held the handheld

---

Figure 2.1: ICB Block Diagram
application code, and the data needed by kernel and/or handheld application software.

2.2.2.3 PCMCIA Card

The ICB had a PCMCIA card built onto the board. This slot was designed to allow the main processor read/write privileges to an extended RAM card inserted in the PCM/CIA slot on the WMI™. Options were provided for the 68306 to toggle between regular and attribute RAM of the PCM-CIA card, and to monitor the write protect switch on the card. The card-detect line of the PCM-CIA slot was connected to the IP2 pin of the 68306.

The needs of this card were monitored and processed by an additional 6805 processor on the ICB bus. If the internal battery of the external RAM card needed to be replaced, the 6805 sensed it. The 6805 would then toggle the BVD2 line on the 68306 that would appropriately process this fault condition.

2.2.3 The ICB Bus

The ICB bus was a multiprocessor bi-directional data bus that allowed communication between the 68306 and up to seven 6805 microcontrollers. The 6805s controlled the special purpose devices of the WMI like the GPS receiver, decoder boards etc. The 68306 polled the 6805s for information using the ICB system bus. The IRQ line was pulsed high to initiate a one-byte data transfer on the bus. R/W* was set high if the 68306 wanted to read a byte from the 6805 and R/W* was set low if the 68306 needed to write a byte to the 6805 specified by a device address. ACKN* was the 6805s response line. The 6805 signaled this ACKN* line low when the data had been sent to/received from the data bus.

2.2.4 The ICB Backplane

The backplane section was the interface between the ICB and the special purpose devices. It consisted of four decoder board connectors, four receiver connectors, a GPS connector and two 24-pin connectors that connected directly to the ICB. It also consisted of a backup battery that powered the ICB RAM and the real-time clock (see Section 2.1.5.2) when external power was removed from the ICB. Once the time on the real-time clock device had been initialized (using application software), the battery backup would ensure that the clock was on even though the WMI powered down.

2.2.5 The Special Devices Section

The ICB interfaced to special purpose devices on board the WMI™. These devices were the receivers, decoder-boards, a GPS unit, a cooling fan, a thermostat etc. The motherboard was built to continuously monitor the health of the WMI™ while processing messages that came in from the wireless receivers. This section briefly describes all these special purpose devices.
2.2.5.1 The Digital Thermostat (DS1620)

The DS1620 thermostat chip, when enabled, measured the temperature inside the WMI™. It had the ability to turn on/off two low-current fans to reduce the internal ambient temperature. The DS1620 was initialized with the safe lower and higher temperature bounds (over a clocked data line). This thermostat chip automatically turned on the fans when the internal ambient temperature of the WMI™ rose above the defined higher temperature limit. Likewise, the fans would be turned off when the temperature fell below the defined lower temperature limit. The DS1620 was controlled by a dedicated 6805 processor (#4 in the bank of 6805s on board the WMI™). The controlling 6805 would issue initialization commands (to set the lower and upper temperature limits), read the current temperature, issue fan turn on/off commands etc. It interfaced with the DS1620 processor.

2.2.5.2 The Real Time Clock Device (RTC-72423B)

The RTC-72423B was the real-time clock chip. Once initialized by its controlling 6805 (#5), it would maintain the current date/time with an accuracy of one second. In order to initialize the clock the kernel function `set_rtc()` could be invoked by the application software. The controlling 6805 would set/read the date/time values and interface with this RTC chip. As mentioned before, the backup battery was connected to the RTC-72423B chip in order to maintain the date/time of the WMI system even when the main power was removed.

2.2.5.3 The Expansion Slot

The expansion slot was a 15-pin connector that allows other devices (besides those already present on the WMI™) to interface with the WMI. This slot was left unimplemented.

2.2.5.4 The Reference Oscillator (NTC-1150)

This oscillator chip provided a stable 14.4MHz signal to all the installed wireless receivers. It was controlled by another 6805 (#4). This was the same 6805 that controlled the thermostat device (Section 2.1.5.1). This 6805 interfaced with this oscillator device and issued commands to calibrate it.

2.2.5.5 The Volume Control Device (X9MME)

An internal speaker was present on the WMI™. The X9MME volume control chip controlled the volume of this speaker. This X9MME chip interfaced with another dedicated 6805 (#6). This 6805 processor could issue volume raise/lower commands to the speaker.

2.2.6 Decoder Boards on the WMI

The four wireless receivers present in the WMI™ interfaced to four independent receiver-decoder boards. These decoder boards would extract the digital information sent over the wireless channel. A 6805 was on each decoder
board and monitored the health of the decoder board, processed the incoming messages, interacted with the 68306 processor and performed other tasks. These decoder boards were physically connected to the ICB backplane.

Each receiver could be tuned to one of four IVDS channels within the 900MHz-frequency band. Packetized messages were transmitted using spread-spectrum modulation techniques to these receivers. Received signals were passed from the receivers to the corresponding decoder boards. An STEL-2000A processor, present on each decoder board, converted the incoming wireless signal to a digital bit-stream that represented the transmitted data. Each decoder board supported an EEPROM that contained the application software for controlling and monitoring the decoder board. The processed messages were buffered inside the corresponding 6805 memory and eventually sent to the 68306 processor upon a message request.

After the WMI was powered on, application software ran through a self-check to test its attached devices. A part of this routine consisted of a power-up sequence for the four decoder-boards. The sequence was to momentarily power up the decoders and then to shut them off. This sequence was executed to test if the decoders were working on power up. The application software written for the 6805s on the decoder-boards would act in a predictable manner in this power-up sequence stage. There was a communication protocol that was followed for any read/write operations between the 6805 and 68306, depending on which processor initiated the communication. This is described further in Section 2.2.2 of this chapter.

2.3 Software features of the WMI™

The WMI™ came with built in kernel software. This kernel consisted of interface routines to interact with the special purpose devices, protocol implementations between the 6805s and the 68306, interrupt handlers, etc. These features are further explored in the following sections.

2.3.1 The WMI kernel core

This consisted of the kernel software functions, the memory map (ROM, RAM, PCM-CIA memory segmentation), exception handlers, and global variables.

2.3.1.1 The kernel software

Robert A. Brickhouse, from Grayson, wrote the WMI kernel in the C language. Software development tools from Microtek were used to develop the application software for the IVDS project. Using these tools, an in-circuit monitor code was created to supervise the working of the ICB board. After developing the software in stages, the application code was then compiled to compatible binary files and burned into the two socketed EPROMs (U20 & U21 on the ICB).

Since a copy of the kernel was stored in the RAM, it was possible for the kernel to start up in either the ROM-based or the RAM-based mode. Upon power up, the WMI software had to perform some required duties. These included powering up in the last known kernel mode (RAM- or ROM-based), restart the
handheld application in RAM (optional), finish pending transactions with all its peripheral devices, process commands received over serial port B, and send data over serial port B, if necessary. If during the last power up state, the handheld application was being executed, it would be restarted the next time the WMI unit powered up. Also, if a PC was controlling the WMI through serial port B, the WMI would send all the peripheral data out through this port to its master (PC). Alternatively, if the handheld controller was controlling the WMI through serial port B, the peripheral data remained unprocessed. Besides these start-up routines the WMI software also allowed for recovery from any 68306 exceptions and tracked the time/data since its last power up mode.

Once this startup routine was satisfactorily completed, the application software was executed by the WMI.

2.3.1.2 The kernel memory map

The 68306 had 4GB of directly addressable address space, of which only a fraction was used. The address space was divided into three sections - a RAM section, a ROM section and a PCM-CIA card section. The chip select line CS1* configured the lower memory ($000000 - $0FFFFF) as ICB RAM. The chip select line CS0* provided access to the ICB ROM ($300000 - $3FFFFF), and the CS2* line selected the PCM-CIA card and configured its memory ($400000 - $5FFFFF). This memory configuration was done by the kernel routine, init_icblow(). Figure 2.2 shows these three sections and their internals.

2.3.1.3 The kernel global variables

1KB of RAM was reserved for the kernel global variables ($400-$7FF) which held the status of the WMI software. Care should be taken when modifying these kernel variables, as they are needed at a later point in time.

2.3.1.4 The kernel jumptable

1KB of RAM was reserved for the kernel jump table. This jump table was an array of function pointers. The application software could invoke the functions present in this jump table. There was a header file (kernel.h) that had the prototypes for these kernel functions.

Additional custom-built functions could be included in the jump table. In order to do this, the new function name would be added to the jump table array (in the INCLUDE\JUMPTAB.INC file). The appropriate prototype for this custom-built function would be added to the kernel.h header file.

2.3.1.5 The kernel exception handling

All 68306 software exceptions that occurred after kernel initialization were handled by the default kernel exception handling routines (located in the CODE\INTERRUP.C file). Software exceptions are illegal interrupts that were not supposed to occur during the software flow. These exception routines
restarted the kernel code after updating the global variables with the software exception information.

2.3.2. Communications and packet protocol between the 68306 and 6805s

All inter-processor communications were performed over the ICB bus. This section describes the protocol about how data is transferred between processors (68306 and 6805s). The 68306 processor initiates all data transfers.

2.3.2.1 Data transfer from a 6805 to the 68306

The packet protocol required when a 6805 sent data to the 68306 is shown in Table 1. Every data transfer began with an STX and ended with an EOT. If an ENQ was sent, the next byte sent was the error number. If an ETX was sent, it indicated that the buffer was temporarily empty (to retry after a few microseconds).

The ICB bus timing required to transfer a byte from a 6805 to the 68306 is shown in Figure 2.3.

Figure 2.2: WMI Kernel Memory Map
2.3.2.2 Data transfers from the 68306 to the 6805

The packet protocol required when the 68306 sent data to a 6805 is shown in Figure 2.5. The control codes shown in Table 2 may only be sent as part of a data block if prefixed with NULL. The ICB bus timing required to transfer a byte

<table>
<thead>
<tr>
<th>Protocol Character</th>
<th>Byte Representation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL</td>
<td>$00</td>
<td>Not used</td>
</tr>
<tr>
<td>SOH</td>
<td>$01</td>
<td>The next byte is literal data</td>
</tr>
<tr>
<td>STX</td>
<td>$02</td>
<td>Begin data transfer from 6805 to 68306</td>
</tr>
<tr>
<td>ETX</td>
<td>$03</td>
<td>Not ready, buffer empty, try again</td>
</tr>
<tr>
<td>EOT</td>
<td>$04</td>
<td>End data transfer from 6805 to 68306</td>
</tr>
<tr>
<td>ENQ</td>
<td>$05</td>
<td>Next byte is an error code</td>
</tr>
</tbody>
</table>

Table 1: Packet Protocol for Data from 6805 to 68306

Figure 2.3: Timing Diagram for Data Transfers From 6805 to 68306
from the 68306 to a 6805 is shown in Figure 2.4. Also, Table 3 shows a list of 6805 command bytes.

![Timing Diagram for Data Transfers From 68306 to 6805](image)

<table>
<thead>
<tr>
<th>ADDR</th>
<th>PB0-2</th>
<th></th>
<th>VALID ADDRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>R/W*</td>
<td>PB4</td>
<td>1 0</td>
<td></td>
</tr>
<tr>
<td>DATA</td>
<td>PA0-7</td>
<td>X</td>
<td>VALID DATA</td>
</tr>
<tr>
<td>IRQ*</td>
<td>PB3</td>
<td>1 0 1</td>
<td></td>
</tr>
<tr>
<td>ACKN*</td>
<td>PB5</td>
<td>1 0 1</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 2.4: Timing Diagram for Data Transfers From 68306 to 6805**

<table>
<thead>
<tr>
<th>Command Byte</th>
<th>6805 Address</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10</td>
<td>4</td>
<td>Turn on decoder 0</td>
</tr>
<tr>
<td>0x11</td>
<td>4</td>
<td>Turn on decoder 1</td>
</tr>
<tr>
<td>0x12</td>
<td>4</td>
<td>Turn on decoder 2</td>
</tr>
<tr>
<td>0x13</td>
<td>4</td>
<td>Turn on decoder 3</td>
</tr>
<tr>
<td>0x15</td>
<td>4</td>
<td>Turn off decoder 0</td>
</tr>
<tr>
<td>0x16</td>
<td>4</td>
<td>Turn off decoder 1</td>
</tr>
<tr>
<td>0x17</td>
<td>4</td>
<td>Turn off decoder 2</td>
</tr>
<tr>
<td>0x18</td>
<td>4</td>
<td>Turn off decoder 3</td>
</tr>
<tr>
<td>0x1B</td>
<td>4</td>
<td>Read temperature</td>
</tr>
<tr>
<td>0x30</td>
<td>5</td>
<td>Read real time clock</td>
</tr>
<tr>
<td>0x31</td>
<td>5</td>
<td>Set real time clock</td>
</tr>
<tr>
<td>0x90</td>
<td>4</td>
<td>Return # of 2.5 ms ticks counted since startup</td>
</tr>
</tbody>
</table>

**Table 3: 6805 Commands**

<table>
<thead>
<tr>
<th>SOH</th>
<th>Command Byte</th>
<th>Supporting Bytes (if any)</th>
</tr>
</thead>
</table>

**Figure 2.5: 68306 to 6805 command packet format**
2.3.2.3 The PC protocol

The WMI kernel dedicated serial port B of the 68306 for PC communications. The WMI executed received commands and shuttled data back to the PC over this serial port. All data sent and received was packetized and followed the protocol described in Section 2.2.2. The PC connected to the WMI was given the choice of becoming either a master (taking over the WMI activities) or a slave (dummy display terminal).

Figure 2.6 shows the format of packets received by and sent from a PC application. Table 4 lists all the devices capable of receiving packets sent over serial port B.

All packets received with an address nibble of 0x0A were routed to the WMI kernel command buffer. This command buffer enabled a PC to perform special WMI functions such as volume control and Doppler port control. WMI kernel command packets have the format shown in Figure 2.7.

<table>
<thead>
<tr>
<th>Protocol Character</th>
<th>Byte Representation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL</td>
<td>$00</td>
<td>The next byte is literal data</td>
</tr>
<tr>
<td>SOH</td>
<td>$01</td>
<td>Begin a command. Specific command follows in next byte(s)</td>
</tr>
<tr>
<td>STX</td>
<td>$02</td>
<td>Begin data transfer from 68306 to 6805</td>
</tr>
<tr>
<td>ETX</td>
<td>$03</td>
<td>Not ready, buffer empty, try again later</td>
</tr>
<tr>
<td>EOT</td>
<td>$04</td>
<td>End data transfer from 68306 to 6805</td>
</tr>
<tr>
<td>ENQ</td>
<td>$05</td>
<td>Next byte is an error code</td>
</tr>
</tbody>
</table>

Table 2: Packet Protocol for Data from 68306 to 6805

Figure 2.6: Format of Packets sent/received over PC

Figure 2.7: WMI Command Packets
2.3.3 Kernel in-built functions

Table 4 lists and discusses briefly all the kernel functions/modules that can be invoked by the application software. Most of these source modules had the specific purpose of interfacing to a device or a set of devices on the ICB backplane. The kernel functions within these source files that are associated with interfacing to any device are described below.

`init_?`: (e.g. `init_decoder`) attempted to initialize a device (decoder in this case). The device was initialized by first contacting the 6805 controlling it. If this was successful, a command was issued to the device that caused the device to respond. All device buffers were then reset before the function ended. If healthy, a device would properly respond to the command it receives.

`process_?`: (e.g. `process_decoder`) checked the devices' health (in this case, the decoder by examining its response). This function exited if only partial data was received from the controlling 6805. It attempted to retrieve the rest of the data packet the next time it was invoked.

`write_?packet`: (e.g. `write_decoderpacket`) copied a block of bytes into the kernel’s output to device buffer. All required packet protocol overhead bytes were inserted into this block of data before the packet was placed into the buffer. If this buffer was too full to hold the packet, the packet was not stored in the buffer.

`?_packetavail`: (e.g. `decoder_packetavail`) returned the number of unprocessed packets received by the `process_?` function (currently stored in the kernel’s input from device buffer). A packet was considered to be unprocessed if it was not yet removed from the input from device buffer. Packets were removed from the input from device buffer by the `read_?packet()` function.

`read_?packet`: (e.g. `read_decoderpacket`) extracted a data packet from the kernel’s input from device buffer. All packet overhead bytes were stripped before returning this packet was processed. If the buffer was empty, no data was returned. The returned packet was the oldest packet in the buffer.

`settotest_?`: (e.g. `settotest_decoder`) placed the corresponding special purpose device in test mode.

`?_status`: (e.g. `decoder_status`) returned a pointer to the status structure of the device.
decoder_bufferstatus: (e.g. decoder_bufferstatus) returned the number of bytes in the devices I/O buffer.

flush_buffer: (e.g. flush_decoderbuffer) discarded the contents of the device’s I/O buffers. All buffer pointers were reset.

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MONBASE</td>
<td>68306 initialization code/Microtek monitor startup code</td>
</tr>
<tr>
<td>BOARD</td>
<td>ICB specific definitions and functions required by the Microtek monitor</td>
</tr>
<tr>
<td>BUSIO</td>
<td>ICB Bus interface routines</td>
</tr>
<tr>
<td>DECIO</td>
<td>decoder - ICB interface</td>
</tr>
<tr>
<td>ENTRY</td>
<td>kernel C initialization code</td>
</tr>
<tr>
<td>GLOBAL</td>
<td>kernel global variables</td>
</tr>
<tr>
<td>GPSIO</td>
<td>GPS -ICB interface routines</td>
</tr>
<tr>
<td>HANDIO</td>
<td>handheld controller - ICB interface routines</td>
</tr>
<tr>
<td>ICBKERN</td>
<td>main kernel routines (e.g. main kernel loop, global kernel functions)</td>
</tr>
<tr>
<td>INTERRUP</td>
<td>kernel interrupt service routines</td>
</tr>
<tr>
<td>SERIALIO</td>
<td>68306 serial port interface routines</td>
</tr>
<tr>
<td>XMON68K</td>
<td>Microtek monitor core code (this module supplied by Microtek)</td>
</tr>
</tbody>
</table>

Table 4: Kernel Code Modules

2.3.4 The kernel hierarchy

The hierarchy of the kernel code is illustrated in Figure 2.8. It approximately follows the inverted tree format, where each strata of kernel used the functionality of its lower level kernel.

Figure 2.9 displays the packet flow diagram and explaining which kernel functions were responsible for controlling various I/O buffers, interacting with serial ports and dealing with the ICB bus.
2.3.5 The three modes of the kernel software

Kernel software could be compiled in three ways. These were appropriately named as debuggable, downloadable and EEPROM-based kernel code. Of these, the first and the last options were used widely throughout the IVDS project.

Debuggable version of the kernel was used during software developmental stages while the EEPROM-based kernel was used after stages of code completion.

2.3.6 Breakpoints and halting code execution

During developmental stages, the software execution could be stopped at any time by using either software or hardware breakpoints. Software breakpoints could be set/cleared using the development tools prior to code execution. Hardware breakpoints could be initiated at any time during the software execution.

Hardware breakpoints were initiated by shorting the TP1 (test point 1) to ground. As explained previously, the IRQ4* line is connected to TP1 on the ICB. This port line, when shorted to circuit common for a few seconds triggered an unmasked interrupt to the main processor which stopped code execution. Two wires were inserted into the TP1 and circuit common slots on the motherboard.
The same two wires were used to invoke the monitor code in the EEPROM devices. Shorting TP1 to common twice in rapid succession, after switching on the main power switch to the WMS accomplished this dual role.

![Packet Flow Diagram]

**Figure 2.9: Packet Flow Diagram**

### 2.4 Summary and potential bottlenecks of the WMI™

Clearly, the WMI had many desirable features needed by the repeater unit of the IVDS system. It had more than the sufficient number of hardware receivers and accessories, and also was well equipped with in-built kernel. The modularity of the kernel was very useful in creating a custom software application while not necessarily knowing much detail about the lower kernel functionality. For example, it was simple to write an application software segment to initialize, read...
from, and write to any of the decoders. This could be achieved without knowing how the kernel transacted with the decoder board. Though it was initially conceived that the application software would be transparent to the internal kernel operations, some potential bottlenecks of the system surfaced with the projects development time.

The WMI™ was designed for use in throughput analysis and other testing issues related to the cell phone industry. While the existing software architecture of the kernel helped the application it was built for, there were gaps where kernel routines failed to meet the IVDS project requirements. For example, the main processor polled the messages coming out of the decoder boards. This assumed that the processor processed the messages faster than the decoder boards could feed it. However, for the IVDS application, the case was reversed.

Messages could be streaming in voluminously from the decoder boards. The application software written for the decoder boards added minimal delay time between the messages coming in to it and the messages going out from it to the main processor. The main processor, on the other hand, had a lot of message processing to perform (queuing, validation, time stamping, rescheduling, transmitting, etc.) besides emptying the decoder board buffers. Hence, polling was not a good strategy to retrieve messages from the decoder boards. The decoders were eventually addressed on an interrupt basis.

The data processed by the GPS kernel routines assumed that the data from the GPS receivers always came in with a standard format. There are existing GPS "condensed" data formats that are sub-sets of the standard formats. The kernel routines could not handle data of this nature and needed to be modified.

Other shortcomings of the kernel included the lack of processing speed of the serial ports, getting unreliable data from the RTC device etc. These modifications to the kernel were made in the SERIALIO.C, GPSIO.C, DECIO.C, BUSIO.C and ICBKERN.C files. These kernel improvements slowed down the system as the number of interrupts increased.

For more information on the Wireless Measurement Instrument (Pin-Out diagrams circuit schematics) the Product Manual that accompanies the hardware can be referred to [1].