CHUCK FORDYCE

Tektronix®
COMMITTED TO EXCELLENCE

USING GPIB
OSPI

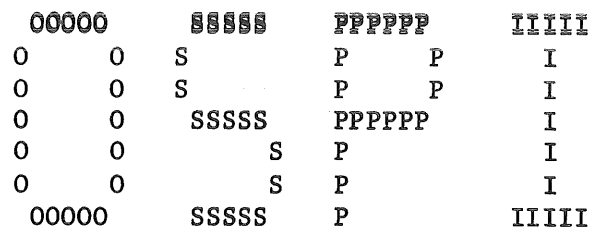
INSTRUCTION MANUAL

Tektronix, Inc.
P.O. Box 500
Beaverton, Oregon 97077

Serial Number _____

June 23, 1981

CHUCK LORRAINE



OPERATING SYSTEM FOR PROGRAMMABLE INSTRUMENTS

Date: May 5, 1981
Robert Bretl
Robert Fitzsimmons
Carl Hovey
Steve Tuttle
x 1118 or 1104

An introduction to this document "Designing With GPIB" is available from Terry Lang, Ext. 1164 WR, Del. Sta. 92-716. I also have more information on using GPIB in your instruments. Please feel free to call me anytime.

This is a preliminary document and is subject to change. A record of changes to the design is maintained and we encourage everyone that implements OSPI to call us to find out about changes since this distribution.

Note:

Sections of this document which have been modified since the last distribution are flagged in the table of contents and index with the string "#####".

$

Introduction

This document is intended to describe the internal operating system designed for the TM500 programmable instruments. The design of this system is based upon the following premises:

1. The instrument internal operating system must never ignore other responsibilities because it is busy waiting for some event to occur. For example, it should not ignore inputs from the front panel or GPIB interface because it is monitoring some hardware function.

2. The instruments must be easy to operate in current programmable instrument systems.

   a) They must use protocols similar to those of existing instruments.

   b) The instrument language takes into consideration existing and future controller capabilities to generate the language constructs.

3. The system must provide the user with a set of primitive operations which provide control over the basic functions of the instrument. These primitive operations are defined so that the user can combine them to perform more complex operations and in some cases macros are provided which make certain sequences of operations easier to specify.

4. The design of the operating system is intended to be modular and general enough to apply to most instrument design problems. It provides a framework for instrument firmware in that it can be reused by changing device dependent portions (command tables, command handlers, hardware drivers, etc.) without entirely redesigning the system.

External features of the operating system are described in a related document titled "Programmable Instruments Firmware Features" (PIFF). It is assumed that the reader of this document has some knowledge of these features and the specifications for GPIB compatable instruments as described in the "IEEE Standard Digital Interface for Programmable Instrumentation" (IEEE Std. 488-1975) and the Tektronix "Codes and Formats Standard for the General Purpose Interface Bus (GPIB)".

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

$

1.1 Control Structure

The firmware in a programmable instrument is responsible for performing a number of functions. Examples of these are: Input and execute commands from the GPIB interface, Monitor hardware operations to maintain performance specifications, determine when front panel buttons are pushed and perform the required functions, etc.. To perform these functions it is often necessary to wait for some event to occur before processing is continued, and this waiting interferes with the performance of other functions which must continue to occur. For example, the fact that the GPIB interface did not transmit a byte should not prevent the processor from recognizing that the operator has pushed a front panel button.

THE PRIMARY GOAL OF OSPI IS TO PROVIDE A STRUCTURE IN WHICH SEVERAL FUNCTIONS CAN BE PROCESSED IN A WAY WHICH PREVENTS THE FUNCTIONS FROM INTERFERING WITH EACH OTHER.

To achieve such a system, the functions to be performmed are partitioned into **Tasks**. The partitioning is done on the basis of whether or not the functions must wait for an event to occur, and if so whether the waiting interferes with the performance of another function. The result of the partitioning is several groups of functions (**Tasks**) and each of these tasks must be performed concurrently.

In the operating system for the TM500 programmable instruments, there are five basic tasks. They are:

1. The **Message Processor** is responsible for decoding and executing all commands received over the GPIB interface.

2. The **Key Processor** task works in conjunction with the front panel interrupt handler (if the interrupt handler is implemented) to decode the front panel button pushes and execute the instrument functions.

3. The **Hardware Monitor** task performs the device dependent hardware support functions. For example, the "bit diddler" in the Function Generator, determining the regulation of the Power Supplies, reading a measurement from the DMM hardware.

4. The **Hardware Settings** task is activated by either the Message Proccessor or by the Front Panel task. The function performed by this task is to copy the pending settings to the current settings and to the hardware as well as update the front panel display.

5. The **GPIB Task** performs those functions required to drive the GPIB interface which cannot be performed in the interrupt handler due to execution time constraints. The primary activity here is to determine when the instrument should enable its talker function to transmit the "talked with nothing to say" message.

Each task is an independent process and shares the processor resource within the instrument. Because there is no clock in the instruments to arbitrarily decide how long any task should execute before giving up the processor to another, the tasks in OSPI must give up the processor resource voluntarily. This is in contrast to most operating systems in which the monitor is in complete control.

When a task reaches a point in its execution at which it must wait for an external asychronous event (such as waiting for the user to press a button), the task saves its status and passes control to the monitor. This passing of control has been named **SUSPEND**, and may be thought of as a return to the monitor. However, on a structure chart, it appears as a call to the monitor since the monitor returns control back to the point of the suspend in the task. For the 6800 processor the saving of status and passing of control is accomplished with the software interrupt (explained further below) and although it may be referred to as "return to monitor" or "call the monitor" or "suspend" or "software interrupt", the important concept is that A TASK EXECUTES UNTIL IT REACHES A CONVENIENT POINT, THEN SAVES ITS STATUS AND PASSES CONTROL TO THE MONITOR.

In OSPI, the **Monitor** (the routine which controls which task is executing) is called by the tasks and this is normally done when the task encounters some event it must wait for. Those tasks which do not have to wait for some event, are implemented so that they give up the processor at least once in their main loop.

Each task then is an independent process and is written as a "loop forever" module. Because it is an independent process and must suspend its execution temporarily, EACH TASK IS ASSIGNED ITS OWN STACK AREA. This stack is used in the normal fashion, and maintains the state of the task while it is suspended. A well known use of the stack for this type of operation is to handle interrupts. When an interrupt occurs, the current state of the process is saved on the stack and can be resumed when the interrupt handler has completed its operation.

The mechanism used to call the **Monitor** in OSPI is very similar and is called the Software Interrupt (SWI). The SWI was chosen because it automatically saves the processor state on the task stack and simplifies the function of the monitor.

To switch execution from one task to another the monitor maintains a **Stack Table** containing the stack pointers for each task and a pointer to the **Active Task**. The tasks call the monitor by executing the software interrupt instruction (**SWI**) which sets the interrupt mask and saves the processor state on the task stack. The monitor then saves the task stack pointer in the table entry indexed by the **Active Task Pointer**. The next task to be activated is determined using a simple "round robin" scheduling algorithm, and its stack pointer is fetched and loaded into the processor stack. The activation of the next task is accomplished by executing a return from interrupt (**RTI**) instruction.

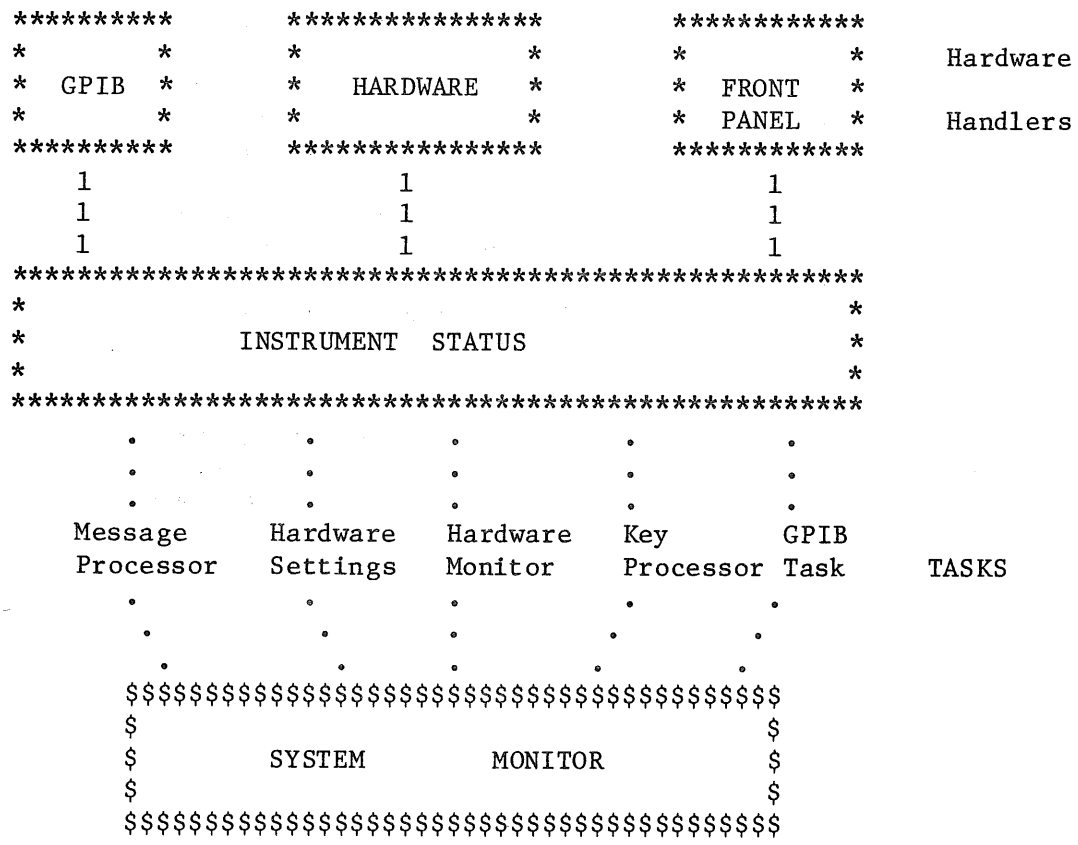
The other major components of the operating system are the **Hardware Handlers** and the **Instrument Status** area.

The **Hardware Handlers (Drivers)** are the modules that interface the hardware to the functional tasks in the system. These modules may or may not be interrupt driven, depending upon the device dependent requirements. They provide the system with information from three primary sources: the GPIB interface, the device dependent hardware, and the instrument front panel. In addition, these drivers also handle the transfer of data from the Instrument Status area to the output device. For example, the GPIB driver puts characters into the **Output Buffer**, changes the Instrument Status (whether Talked, Listened, etc.) and handles the transfer of bytes from the **Output Buffer** to the GPIB chip.

The **Instrument Status** is the data area (in RAM) which defines the current state of the instrument. This data is partitioned into a number of sections which are used to specify the state of a subsystem. For example, Instrument Settings, Front panel display status, results of measurements, Task stacks, and GPIB status are defined as subsections of the Instrument Status.

The **Instrument Status** area is also used to communicate the control information between elements of the system. Below is a block diagram or structure chart which shows the major components of the system and how they are related to each other.

Structure Chart



Note that the tasks may pass control to the monitor from anywhere within its code and that the monitor passes control back to that same point. On the structure chart the passing of control has been shown as a call to the monitor. For the 6800 processor the passing of control is implemented with a software interrupt and is referred to as a **SUSPEND**.

1.2 System Monitor Initialization

The processor powers up with interrupts masked. To start the system, the task Stack Table must be initialized to contain all task stack pointers. Each task stack pointer is set to (top of task stack area) - 7, the task stacks are initialized with the program counter locations set to the task's power-on initialization routine, and the condition code locations are initialized so that interrupts are masked when the task is executed the first time. Then the Active Task Pointer is set to the top of the Stack Table. Finally the System Monitor is activated by loading the stack pointer register with the Stack Table entry pointed to by the Active Task Pointer. Control is passed to the first task by executing an RTI instruction.

Each task is structured so that it:

1. Executes its individual power-on initialization
2. Executes a SWI (Suspend).
3. Executes a CLI (Clear Interrupt Instruction).
4. Executes its main idle loop.

Each task then executes its own power-on initialization and suspends with interrupts disabled. When all tasks are initialized (after the first pass through the monitor), each task unmasks the interrupts (in the second pass) and the system is running.

1.3 Interrupt Handling

Because the SWI is used to suspend tasks, interrupts become just another way to suspend task execution. The only difference between an interrupt suspending a task and the task suspending itself is that the interrupt is asynchronous. The result is that tasks can share subroutines (if the subroutine does not suspend) without the subroutine being re-entrant. Interrupt handlers can only share subroutines with tasks if they are re-entrant.

Interrupt handlers need the capability to reset certain tasks (the GPIB handler resets the **Message Processor** on DCL). Resetting a task to its power on state is accomplished by resetting the task stack pointer and storing the task's initialization entry point in the location where the PC is saved for recovery by the **RTI** instruction. Because the interrupt handler may need to reset the stack of the task it interrupted, it cannot use the task stack for its operations. To solve this problem we have allocated another stack, the **Interrupt Stack**. All interrupt handlers are written so that they:

1. Save the processor stack in the **Stack Table** entry indexed by the Active Task Pointer.

2. Load the base address of the interrupt stack pointer into the processor.

3. Perform the interrupt handlers function.

4. Recover the current task stack.

5. Execute an **RTI** instruction.

Notes:

The overhead required by this system is that each task's stack area must contain sufficient space for its deepest level of subroutine nesting, plus the seven bytes required to save the processor state when either an interrupt occurs or a task is suspended.

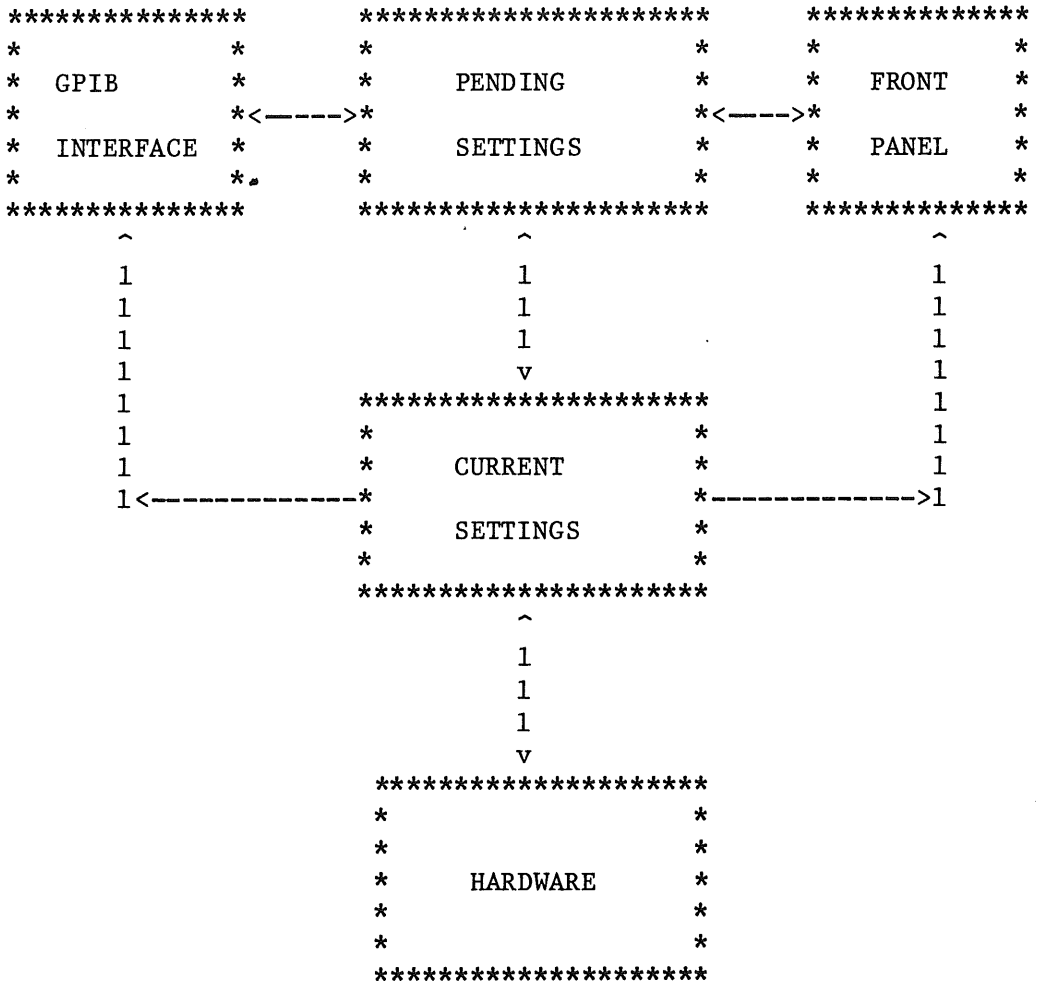
The interrupt handler definition implies that there is only one level of interrupt service -- a new interrupt event cannot interrupt another service routine. This approach is used to save **RAM** required for nesting the interrupt service routines since there is no need in the TM500 programmable instruments for the extra capability.

A side effect of having an independent stack for the interrupt service routines is that it also saves **RAM** since this space does not have to be duplicated in each task's stack area.

Use of the **SWI** instruction with the 6800 precludes the use of **NMI** since the 6800 has a bug which if NMI and a SWI occur at the same time, the processor incorrectly goes to the **IRQ** vector.

1.4 Data Flow Diagram

The **Data Flow Diagram** below provides a very simplified view of the data flow within the instruments. There are two different types of data being transmitted to/from the GPIB interface or front panel. These are Instrument Settings and for measurement instruments, the measurement data. Measurement data is transmitted from the hardware to storage area in RAM and from there it is tranferred to the front panel and GPIB interface. The important concept portrayed in the diagram is a system philosophy based upon multiple copies of the instrument settings. The instrument hardware itself contains one representation of the instrument settings. Because the hardware is usually implemented as a set of write only registers, a copy of the instrument state is maintained in the **Current Settings** buffer. Another copy of the settings is maintained in what is called the **Pending Settings** buffer. As its name implies, the **Pending Settings** buffer stores settings processed by the instrument operating system (whether they came from the Front Panel or the GPIB interface) until they are executed -- that is, until the settings are copied into the **Current Settings** buffer and the hardware.



The reasons for this approach are:

1. By building pending settings a message (or portion of a message) can be processed and the new state can be evaluated to determine if it is legal before executing. This technique prevents signal sourcing instruments such as the power supply or function generator from entering illegal intermediate states.

2. The ability to scan the message for syntax errors before executing settings is greatly simplified. (This is one of the "friendly" instrument features.)

In normal operation of the instrument, hardware settings are updated when:

1. An entire command is entered from the Front Panel (for multiple keystroke commands this occurs when the **ENTER** button is pressed).

2. The **Message Processor** completes the decoding of the entire message.

3. Before any operational or output command.

For a more complete discussion of when hardware settings are updated, see the section titled **Execution of Settings**.

The **Pending Settings** and **Current Settings** are stored in the image of the hardware register's format. This is done to minimize the response time to **GET**. In some cases, this data format is not convenient for performing **Pending Settings** verification. When this is true some of the settings are also saved in a format more applicable to the data processing required (floating point or unscrambled binary).

To make it easy to identify when **Pending Settings** are lost, a flag is associated with the **Pending Settings** buffer which indicates "new settings pending" **NSP**. This flag is set by any routine which alters the **Pending Settings** and cleared by the **Hardware Settings** task when the settings are copied to the **Current Settings** or by any routine which resets the **Pending Settings** by copying the **Current Settings** to the **Pending Settings** buffer (as is done in **DCL**).

$

2.1 Overview of Message Processor Operation

Functional Description

The **Message Processor** task is responsible for decoding and executing all commands received over the GPIB interface. It gets messages from the **Input Buffer** and parses them to verify their syntax. Commands (message units) are processed sequentially and the message processor suspends itself between the processing of commands to allow other tasks time to execute. As the commands are processed, they are taken out of the **Input Buffer** to allow further input. When an error is detected in a message, the remainder of the message is flushed before the error is reported and the SRQ is asserted.

Command Processing

GPIB command processing includes fetching the characters from the **Input Buffer**, decoding the headers and arguments, detecting the delimiters and reporting the syntax errors detected in parsing a message unit. The commands are scanned sequentially, in the order they are received.

First the optional format characters allowed at the beginning of a message unit are skipped. Then the header is decoded and the address of the command handler is determined. The **HWSETR** is checked, and the **Message Processor** suspends until it is clear before dispatching to the command handler. The **Message Processor** suspends here to prevent modifications to the **Pending Settings** while the settings are being updated, to prevent queries from returning settings which are in the process of being modified by the **Hardware Settings** Task, and to prevent operational commands from performing their function during the time th

hardware is in an unstable state.

If the command type is **QUERYONLY**, the **Message Processor** saves the **Current Settings** in a buffer which is used to generate the query response. This is done to prevent conflicting settings from being returned in a query due to the fact that a setting was changed (from the front panel) while the data was being output.

The command handlers decode the arguments and perform the function requested. For **Operational Commands**, this implies performing some operation on the hardware. **Output commands** generate the response from the data in the **Current Settings** buffer, and put it into the **Output Buffer**. The settings are processed by placing their binary representation which defines the new instrument state in the **Pending Settings** buffer.

When the particular command handler has completed its operation, it returns through the dispatcher to the **Message Processor**, which handles the message unit delimiter. If another message unit is in the buffer, the **Message Processor** processes it, otherwise it performs the end of message processing and returns to the idle state waiting for the first

byte of the next message.

Command Execution

The description of command processing above is general in nature and doesn't deal with some of the specific cases which cause difficulty. One of the problems which needs to be resolved is when commands are EXECUTED. The word "executed" means "to follow out", "perform", "fulfill", or "carry into effect". As described above, it is evident that the operational and output commands are clearly "executed" in the order in which they occur. Due to the possibility of introducing invalid intermediate states, it is not desirable to "execute" the settings in this order -- we can make the instruments more friendly by managing this problem for the user. For this reason, the execution of the settings is defined to be the act of copying **Pending Settings** to the **Current Settings** and to the hardware. Since this is not an instantaneous operation, the **HWSETR** flag is set to perform this function and is cleared when it is complete.

As pointed out in the discussion of the **Data Flow Diagram**, the execution of settings normally occurs:

1. When an entire command is entered from the front panel (for multiple keystroke commands, this occurs when the ENTER button is pressed). The reason for doing the execution here is obvious, the operator just entered a new setting and implicitly wants the instrument to reflect the new state.

2. When the **Message Processor** completes the processing of the entire message. Here again an implicit execution of settings is implied, since the end of message indicates that the controller has communicated one complete set of actions to be performed.

3. Before processing of any operational or output commands. In this case, it is assumed that the operator wants to make a measurement, generate some output, or read the new instrument status. If the settings were not executed before these commands were processed, the operational or output functions would be performed using potentially "old" settings.

The time at which the settings are excecuted defaults (at Power On) to the three conditions described above. The operator can change this by using the **Device Trigger** function. This is accomplished by sending the instrument the "DT SETTINGS" command, which explicitly instructs the instrument to ignore its default mode of execution and only execute settings when it receives a **GET** and is in the **REMOTE** state.

Notes:

Query command handlers must obtain all necessary information from the **Current Settings** buffer before suspending (due to **Output Buffer** full, for example). This is because the Key **Processor** may change the **Pending Settings** and set the **HWSETR** flag while the query command handler is suspended.

In **DT SETTINGS** mode, many messages may be processed before a **GET** occurs. In this case, the **Message Processor** must not copy the **Current Settings** to **Pending Settings** each time a new message is processed. If it did, the **Pending Settings** from previous messages would be lost. Instead, the **Message Processor** must assume that **Pending Settings** are in a valid state. For this reason, any event that might cause **Pending Settings** to be invalid must perform (or force another routine to perform) a copy of **Current Settings** to **Pending Settings**. This includes the following events:

Device Clear (**DCL**)

Return to **LOCAL** from the front panel (**RTLPROC**)

Transition to **REMOTE** (**RLC**)

Setting range error (from **Key Processor** or **Message Processor**)

Setting Conflict error (from **Key Processor** or **Message Processor**)

Message syntax error

Message Processor Restart

2.2 Types of Commands

There are several types of commands that are handled by the Message Processor.

Setting Commands

These commands are used to change the instrument settings. They typically consist of a header and one or more arguments, which specify the desired state for that function. The setting commands are sometimes further partitioned into Device Dependent and System Setting commands. The System Setting commands are those commands with common mnemonics which change the way the instrument operates in the GPIB system environment (RQS ON/OFF, DT SETTING, etc.).

Examples:

```
VOLTS 5;
LEVEL 247;
SLOPE POS;
```

Output Commands

This second type of commands includes all of the commands which generate output. Most of these are Query commands, which are questions directed to the instrument concerning the settings or some other instrument state. Typical query commands are set command headers to which a "?" has been appended. Another form are Query Only commands, which return some instrument status information that cannot be set (for example, the power supply regulation). The response to all query commands is a header and argument list which defines the state of the queried function.

The other output command used is the SEND command. This command tells the instrument to send a particular measurement. If an instrument has more than one type of value, then a device dependent argument is included to define the particular measurement or value requested.

Examples:

```
LEVEL?
SLOPE?
SEND;
```

Operational Commands

Operational commands are a group of commands which direct the instrument to perform a particular action. These commands typically consist of only a header, and the header describes the action to be performed.

Example:

```
TRIGGER;
ARM A;
START;
STOP;
```

REMOTE ONLY Commands

These are commands which are only executed in the REMOTE state and in LOCAL they generate an error message. They are usually those setting or operational commands which change the front panel state of the instrument. All device dependent setting commands are REMOTE Only.

2.3 Implementation of the Message Processor

Interactions With Other Tasks

1. The Scan Format routine must return with the **GBPTR** pointing to the first non-format character.

2. The Header Table Search routine must return with **GBPTR** pointing to the first character after the header.

3. The command handlers must return an EVENTCOD in a variable called **ERR**. The command handlers should NOT call **NEWEVENT**.

4. The **PUTEOI** routine must not put a message terminator in the **Output Buffer** if the **Output Buffer** is empty. This allows the **GPIB Task** to determine if a measurement result should be placed in the buffer or the "Talked with nothing to say" message.

5. The **ABORTMSG** subroutine must clear the **NSP** flag by calling **COPYC2P**. This prevents the **EXECUTE** routine from asserting the **HWSETR** flag in the case of a message error.

6. All argument processors must return with **GBPTR** pointing to an argument delimiter or a message unit delimiter. Command handlers that do not need arguments can leave the **GBPTR** where the **Message Processor HDRSRCH** routine left it. The string argument processor scans over extra characters at the end of the valid string until an argument delimiter, a message unit delimiter or **EOM** is found. If an invalid argument string is encountered, then the **ERR** variable is set, so it does not matter where **GBPTR** is. The numeric argument processor does a pre-scan on the number, and leaves the **GBPTR** pointing to the first non-numeric character. If this character is not an argument delimiter or a message unit delimiter, then **ERR** is set by either the command handler or the **Message Processor**.

7. Command handlers must not suspend after the **ERR** variable is set since it is a variable shared by the **Message Processor** and **Key Processor**. In addition, the task which calls a command handler must save the value of **ERR** if it needs it after a suspend. This is because the other task might use the variable while the task is suspended.

Message Processor Variables

ERR Used to return the EVENTCOD from the command handlers. ERR = 0 implies that no error occurred.

MPERRCD Stores the EVENTCOD that the Message Processor detects in parsing the message. The ABORTMSG routine reports the error by copying MPERRCD to EVENTCOD and calling NEWEVENT.

CONFLICT An error indicator returned from the Pending Setting Verify Routine. The value of CONFLICT indicates which pair of settings are conflicting. CONFLICT = 0 implies that no conflict exists.

EOM End-of-Message flag, which when true indicates that all of the first message in the Input Buffer has been read. It is set and cleared by the GETBYTE routine.

CMDTYPE The command type variable is returned from the header search routine and indicates whether the command is: a REMOTE ONLY type command, a Setting Command, an Output Command, or an Operational Command. It is derived from the tag byte in the Command Table -- see the section on Command Table Structure.

CMDINDEX The Command Index is returned from the Header Search Routine, and is used as an index into a table of command handler routine addresses.

COMMAND An array of addresses used to dispatch to the command handlers. This array is twice as large as the number of commands, with the first set of entries being those which handle SETONLY commands, the set function of SETQUERYcommands, OUTPUT, and Operational Commands. The second half of the array contains the addresses of the Query Command Handlers. (This was not the original design, but we found that the code could be reduced considerably for instruments with greater than one third of the commands of SETQUERY type by using this table structure.)

MSGREM The Message Remote flag, when true indicates that the current message is to be processed as if the instrument is in REMOTE state. It is set by BYTEIN and cleared by RTLPROC and NEXTMSG.

FPCNTRL The Front Panel Control flag, when true indicates that the front panel user has control over the instrument's settings. It is set by **RTLPROC** and cleared by **MSGPROC**. This flag is needed to determine (in the RLC interrupt service routine) whether there are settings pending execution due to waiting for a **GET** from the GPIB interface. **HWSETR**

The Hardware Settings Request Flag. When true, this flag indicates to the **Hardware Settings** task that a **Pending Settings** execution has been requested. It also is used to indicate to other tasks in the system that a setting change is still in progress. The flag is set in **MSGPROC**, **KEYPROC** , and **GET.IH**. It is cleared only by the **Hardware Settings** task.

INCHAR The last character that the **GETBYTE** routine read from the **Input Buffer**.

OUTCHAR The variable in which a character to be output is passed to the **PUTBYTE** routine.

MLHLDOFF Mag-latch holdoff is a time delay counter that is used to limit the duty-cycle of the mag-latch relay coil current in the FG5010.

MSGPROC

Pseudo code for the Message Processor task.

```
MSGPROC

# ROUTINES CALLED: SCANFRMT, HDRSRCH, ABORTMSG, EXECUTE
#    COMMAND HANDLERS, PUTEOI, NEXTMSG

#CALLED BY: SYSTEM MONITOR ACTIVATING THE TASK

     SUSPEND
     CLEAR INTERRUPT MASK

     REPEAT #FOR EACH MESSAGE
          MPERRCD = 0 # NO ERRORS IN MESSAGE

          REPEAT #FOR EACH COMMAND IN THE MESSAGE.
               SCANFRMT #SCAN OVER FORMAT
               EXITR WHEN EOM
               HDRSRCH #IDENTIFY COMMAND HEADER AND TYPE
               IF MPERRCD <> 0 THEN #HEADER SYNTAX ERROR
                    ABORTMSG #FLUSH MESSAGE AND REPORT ERROR
                    EXITR
               ENDI

               IF REMOTE ONLY COMMAND THEN
                    IF MSGREM THEN #PROCESS AS IF IN REMOTE
                         #INDICATE THAT GPIB HAS CONTROL OF SETTINGS
                         FPCNTRL = FALSE
                    ELSE #IN LOCAL SO ISSUE ERROR
                         MPERRCD = REMONLY
                         ABORTMSG
                         EXITR
                    ENDI
               ENDI

               IF ( CMDTYPE = OPERATIONAL ) OR ( CMDTYPE = OUTPUT ) OR
               ( CMDTYPE = QUERYONLY ) THEN
                    EXECUTE #VERIFY AND EXECUTE ANY PENDING SETTINGS.
                    IF MPERRCD <> 0 THEN
                         ABORTMSG
                         EXITR
                    ENDI
               ENDI

               #MSGPROC MUST SUSPEND AT LEAST ONCE BETWEEN COMMANDS.
               REPEAT
                    SUSPEND
                    #WAIT FOR SETTINGS UPDATE TO COMPLETE.
                    UNTIL NOT( HWSETR )
               ENDR

               IF CMDTYPE = QUERYONLY THEN
```

```
                SAVE CURRENT SETTINGS FOR QUERY RESPONSE
           ENDI

           CALL( COMMAND ( INDEX ) ) #DISPATCH TO COMMAND HANDLERS
           IF MPERRCD <> 0 THEN #COMMAND HANDLER DETECTED AN ERROR
                ABORTMSG
                EXITR
           ENDI

      #LOOK FOR MESSAGE UNIT DELIMITER
      UNTIL ( INCHAR <> SEMICOLON ) OR EOM
      THENDO
           IF ( INCHAR = SPACE ) OR ( INCHAR = LF )
                OR ( INCHAR = CR ) THEN
                SCANFRMT #SKIP OVER FORMAT AT END OF MESSAGE.
           ENDI
           IF NOT ( EOM ) THEN
                #REPORT INVALID MESSAGE UNIT DELIMITER.
                MPERRCD = MSGDEL
                ABORTMSG
           ENDI
      ENDR

      PUTEOI #TERMINATE OUTPUT MESSAGE (IF ANY).
      EXECUTE #VERIFY AND EXECUTE ANY PENDING SETTINGS.

      IF MPERRCD <> 0 THEN
           #REPORT SETTINGS CONFLICT ERROR.
           ABORTMSG
      ENDI

      #WAIT FOR SETTINGS UPDATE TO COMPLETE
      REPEAT
           SUSPEND #AT LEAST ONCE BETWEEN MESSAGES
           UNTIL NOT( HWSETR )
      ENDR

      NEXTMSG # PREPARE FOR NEXT MESSAGE.
ENDR
```

EXECUTE

This routine verifies the validity of any **Pending Settings** that came from the GPIB and then executes those settings if the instrument does not have to wait for a < **GET** > command. It returns only one error type: settings conflict error. It holds off settings execution in the FG until the mag-latch relay timer runs to zero in order to limit the relay coil current duty-cycle.

Pseudo code for the **EXECUTE** subroutine.

```
EXECUTE

#ROUTINES CALLED: VERIFY

#CALLED BY: MSGPROC

     IF NOT ( FPCNTRL OR HWSETR ) AND NSP THEN
          # NEW SETTINGS PENDING ARE FROM GPIB
          VERIFY #PENDING SETTINGS
          IF CONFLICT = 0 THEN
               #EXECUTE THE PENDING SETTINGS IF THE PENDING DT
               #STATE IS NOT "SETTINGS" OR IF THE CURRENT DT
               #STATE IS NOT "SETTINGS"

               HWSETR = PDT <> SETTINGS OR CDT <> SETTINGS
          ELSE MPERRCD = CONFLICT
          ENDI
     ENDI

RETURN
```

Note:

Some instruments need to suspend between the time the **FPCNTRL** flag is tested and the **HWSETR** flag is set true. In this case the EXECUTE subroutine should make sure that the instrument still has GPIB settings to execute since during the time that the task is suspended the KEYPROC task could reset the Pending Settings and execute a front panel command.

For example, in the FG5010, the mag-latch relays have a limit on how often they can be cycled and the firmware must hold off the execution of settings if it could violate this limit. The delay could be put in the Hardware Settings task, but this would affect the response time to a Group Execute Trigger. Therefore it is desirable to put it in the EXECUTE subroutine, and since we only want to delay if the settings were from the GPIB, the suspend is inserted between the test of FPCNTRL and the setting of HWSETR. (Note that the EXECUTE routine is executed every pass through the Message Processor so an unconditional delay is not desireable.)

In this case, an additional test of the FPCNTRL flag is required so that the HWSETR flag is not set if the settings were not from the GPIB. The resulting code is:

```
.
.
IF CONFLICT = 0 THEN
     WHILE MLHLDOFF <> 0 DO
          SUSPEND
     ENDW
     IF NOT( FPCNTRL ) THEN #FRONT PANEL HAS NOT TAKEN CONTROL
          #SET HWSETR TRUE IF PENDING DT STATE <> CURRENT
          #DT STATE OR IF CURRENT DT STATE <> SETTINGS.
          HWSETR = .....
     ENDI
     .
     .
```

ABORTMSG

The abort message routine performs three major functions:

1. Flush the **Input Buffer** to the end of the first message.
2. Restore the **Pending Settings** buffer to the current state of the instrument. This also clears the New Settings Pending flag (**NSP**).
3. Report the error in the message to the status processor.

Pseudo code for the **ABORTMSG** routine.

```
ABORTMSG

#ROUTINES CALLED: FREESPACE, GETBYTE, COPYC2P, NEWEVENT

#CALLED BY: MSGPROC

     REPEAT #FLUSH MESSAGE
          FREESPACE
          GETBYTE
          UNTIL EOM
     ENDR
     IF NOT( FPCNTRL ) THEN
          COPYC2P #RESTORE PENDING SETTINGS
     ENDI
     EVENTCOD = MPERRCD
     NEWEVENT
     MPERRCD = 0

RETURN
```

SCANFRMT

This routine scans the **Input Buffer** for format characters <SP>, <CR>, <LF> and returns with the **GBPTR** pointing to the first non-format character. The character is returned in the variable **INCHAR.**

Pseudo code for the **SCANFRMT** routine.

```
SCANFRMT

#ROUTINES CALLED: FREESPACE, GETCHR

#CALLED BY: MSGPROC, BYTE.CMD, CHR.PROC, NUM.PROC, BIN.PROC

     REPEAT
          #FREESPACE IS CALLED BEFORE EACH CALL TO GETCHR
          #SO WE CAN BACK UP TO THE LAST CHARACTER READ.
          FREESPACE
          GETCHR
          EXITR WHEN EOM
     UNTIL ( INCHAR <> SPACE) AND ( INCHAR <> LF) AND ( INCHAR <> CR)
     THENDO
          GBPTR = BUPTR #BACK UP ONE CHARACTER.
          BYTAVAIL = TRUE
     ENDR

RETURN
```

GETCHR

The **GETCHR** subroutine can be used to get header or character type arguments from the **Input Buffer**. It calls **GETBYTE** and strips off the parity bit, converts lower to upper case, and sets the **INVALCHR** flag TRUE if the character is not legal for headers or character arguments.

Pseudo code for the **GETCHR** routine.

```
GETCHR

#ROUTINES CALLED: GETBYTE

#CALLED BY: SCANFRMT, HDRSRCH, TABLESRCH, NUM.PROC

     INVALCHR = FALSE
     GETBYTE
     INCHAR = INCHAR AND 7FH #STRIP OFF PARITY BIT
     IF INCHAR > 60H AND INCHAR <= 7AH THEN
          #CONVERT LOWER TO UPPER CASE
          INCHAR = INCHAR - 20H
     ELSE #CHECK FOR RUBOUT
          IF INCHAR = 7FH THEN
               INVALCHR = TRUE
          ENDI
     ENDI
     IF INCHAR <= QUES.MRK THEN
          IF ( INCHAR <= SPACE ) #CONTROL CHARACTER
               OR ( INCHAR = COMMA )
               OR ( INCHAR = SEMICOLN )
               OR ( INCHAR = QUES.MRK )
               OR ( INCHAR = COLON) THEN
                    INVALCHR = TRUE
          ENDI
     ENDI

RETURN
```

HDRSRCH

The header search routine scans the **Input Buffer** looking for a matching entry in the **HDRTABLE**. The **HDRTABLE** must be defined as outlined in the section " **Table Structure** ". **HDRSRCH** uses a hash table (**HASH.TBL**) to speed up the search. Since the table is organized alphabetically and all headers must begin with an alpha character, the first character is used as an index into the hash table. The hash table contains an offset from the beginning of the **HDRTABLE**, from which a search for the command starting with that character can begin.

The header search routine begins its processing by checking the first character to make sure it is an alpha and determines where in the table to begin the search. Then it calls the table search, which returns the command type and index into the dispatch table. If a header was found, it does some additional checking to determine whether it is a setting or query command and sets the **CMDTYPE** variable with the appropriate information. If the command is a query, the header search routine offsets the index so that it points into the query command handlers portion of the dispatch table. If a legal header was not found in the table search, an eventcode is returned in the **MPERRCD** variable.

Definitions used in header search:

```
SETONLY = 00H
SETQUERY = 10H
QUERYONLY = 20H
OUTPUT = 30H
OPERATIONL = 40H
```

```
HDRSRCH

#ROUTINES CALLED: TABLSRCH, GETCHR

#CALLED BY: MSGPROC

     IF ( INCHAR >= "A" ) AND ( INCHAR <= "Z" ) THEN
          #CHARACTER IS UPPER CASE ALPHA

          #SET UP VARIABLES FOR TABLE SEARCH
          TABLEPTR = HASH.TBL ( INCHAR - 40H )
          INDEX = LSR( TBLPTR.HI , 2 )
          TABLEPTR = ^ HDRTABLE + ( TABLEPTR AND 3FFH )

          TABLSRCH

          IF TYPE <> 0 THEN #HEADER WAS FOUND
               CMDTYPE = TYPE AND 70H #SELECT TYPE INFO. FROM TAG BYTE
               IF CMDTYPE = SETQUERY THEN
                    IF INCHAR = QUES.MRK THEN
                         #CLEAR REMOTE ONLY BIT IN TYPE
                         TYPE = TYPE AND 7FH
                         CMDTYPE = QUERYONLY
```

```
                    ELSE
                         CMDTYPE = SETONLY
                    ENDI
               ENDI
               IF CMDTYPE = QUERYONLY THEN
                    IF INCHAR = QUES.MRK THEN
                         GETCHR
                         INDEX = INDEX + NUM.CMDS
                    ELSE
                         MPERRCD = CMDHDR
                    ENDI
               ENDI
               #PICK UP REMOTE ONLY BIT
               CMDTYPE = CMDTYPE OR ( TYPE AND 80H )
          ELSE #INDICATE HEADER NOT FOUND
               MPERRCD = CMDHDR
          ENDI
     ELSE #INDICATE FIRST CHAR NOT ALPHA
          MPERRCD = CMDHDR
     ENDI

RETURN
```

2.4 Table Structure

The design of the table incorporates features which aid in searching the table and returning information about the command or argument to be processed. Each table entry consists of a 1 byte **Tag** and N bytes which define its name. The **Tag** byte contains the following information:

Bit	Meaning
B7	Remote Only Flag
B6,B5,B4	Type (for command tables)
	0 Setting only
	1 Setting/Query
	2 Query Only
	3 Output
	4 Operational
	5 Unused
	6 Unused
	7 Unused

(In argument tables, the type field may be encoded with any information the designer wishes, since the **TABLSRCH** routine returns this data.)

B3,B2,B1,B0
Number of bytes in the name (value of 1 thru 15).

A **Tag** byte equal to "00" hex indicates the end of the table.

The ASCII string contains the characters in the name. If M is the length of the unique substring of the name, then the first M characters in the name must have the MSB (B7) asserted. This is used by the Table Search algorithm so that the user can specify any unique substring of the name.

The table is organized alphabetically by name and a speed up technique may be used to start the table search based upon the first character in the command header or argument.

Table Search

The Table Search algorithm is based upon the table format as described in the preceding section. It expects the following inputs:

GBPTR = **BUPTR** = A pointer to the first character in the input string.

INDEX = Starting index in table minus one. (That is, to start a search from the beginning of a table the index should be set to 0.)

TBLPTR = A pointer to the tag byte for the table entry where the search is to start.

The Table Search routine generates the following outputs:

BUPTR may be changed if the string length is greater than the table entry length.

GBPTR points one character past the terminator character.

INCHAR holds the terminator character.

INDEX is the table index (1 through N) where the string was found.

TYPE holds the tag byte for the table entry where the string was found. **TYPE** = 0 implies the string was not found.

Pseudo code for the Table Search routine:

```
TABLSRCH

#ROUTINES CALLED: GETCHR, FREESPACE

#CALLED BY: HDRSRCH, CHR.PROC

     LENGTH = 0
     REPEAT #FOR EACH TABLE ENTRY
          INDEX = INDEX + 1
          TBLPTR = TBLPTR + LENGTH
          TYPE = BYTE[ TBLPTR ]
          EXITR WHEN TYPE = 0 # END OF TABLE
          LENGTH = TYPE AND 0FH

          REPEAT #FOR EACH CHARACTER
               GETCHR
               IF EOM OR INVCHAR THEN
                    TBLPTR = TBLPTR + 1
                    EXITR WHEN LENGTH <> 0 AND BYTE[ TBLPTR ] < 0 THEN
                    RETURN
               ENDI
               IF LENGTH <> 0 THEN
                    TBLPTR = TBLPTR + 1
                    EXITR WHEN (( INCHAR XOR BYTE[ TBLPTR ] ) AND 7FH)
                    <> 0
                    LENGTH = LENGTH - 1
               ELSE
                    FREESPACE
               ENDI
          ENDR
          GBPTR = BUPTR
          BYTAVAIL = TRUE
     ENDR
     RETURN
```

2.5 Rules for Writing Command Handlers

Rules for Setting Commands

Setting commands with no arguments may simply call a routine to perform the requested function (change the appropriate bits in the **Pending Setting** buffer.

Setting commands that call argument processors must perform as follows to provide adaquate syntax checking:

1. The command handler must call **HDRDELIM** (to check for a legal header delimiter) and test **MPERRCD** on its return. This must be done before calling any argument processor and if **MPERRCD** is non-zero, then the command handler should return to the dispatcher without calling any argument processors.

2. After each call to an argument processor, the **MPERRCD** variable must be checked and if it is non-zero control returned to the command dispatcher.

3. For commands with more than one argument, the **ARGDELIM** subroutine must be called to check for a valid argument delimiter. Again, the **MPERRCD** variable must be tested after each call to **ARGDELIM** and control returned to the dispatcher if it is non-zero.

Since it is possible for the command handlers to suspend the **Message Processor** (by decoding arguments, etc.) all commands which might suspend must test the **MSGREM** flag before they put the new setting into the **Pending Settings** buffer. This test is done to make sure that the **REMOTE ONLY** settings function is not performed while the instrument is in **LOCAL** state (it only can go to **LOCAL** if the **Message Processor** task is suspended).

If **MSGREM** is FALSE the **MPERRCD** variable should be assigned the eventcode which indicates that this command is not executable in **LOCAL** state and control should be returned to the command dispatcher. If the **MSGREM** flag is TRUE, it is then safe to change the **Pending Settings** buffer.

Setting Command Examples

The first example is a command which decodes a single numeric argument.

ADDR.CMD

```
#THIS COMMAND DECODES A SINGLE ARGUMENT AND STORES THE 16 BIT
#VALUE INTO THE CURRENT SETTING BUFFER.

#ROUTINES CALLED: HDRDELIM, NUM.PROC

#CALLED BY: MSGPROC IN COMMAND DISPATCH

     HDRDELIM #CHECK FOR HEADER DELIMITER
     NUM.PROC #DECODE THE NUMERIC ARGUMENT
     IF MPERRCD = 0 THEN
          CADDR = FIX( NUM.ARG )
     ENDI

RETURN
```

The second example is a routine which inputs one or optionally more numeric arguments.

```
BYTE.CMD
#THE BYTE COMMAND INPUTS ONE OR MORE DECIMAL VALUES
#(BETWEEN 0 AND 255) AND STORES THEM IN THE LOCATION
#SPECIFIED BY THE ADDRESS ENTRY IN THE CURRENT
#SETTINGS BUFFER.

#NOTE THAT THIS IMPLEMENTATION DOES NOT STRICTLY FOLLOW
#THE RULES FOR SETTING COMMANDS IN THAT IT "EXECUTES" THE
#FUNCTION (STORING THE BYTES) IMMEDIATELY.
#WE DECIDED TO ACCEPT MULTIPLE BYTES FOR EASE OF USE --
#THE COMMAND CAN BE MADE TO STRICTLY CONFORM BY RESTRICTING
#THE NUMBER OF ARGUMENTS ACCEPTED TO ONE, BUT FOR A
#DIAGNOSTIC COMMAND EASE OF USE IS PROBABLY MORE
#IMPORTANT THAN STRICT ADHERANCE TO THE RULES FOR
#SETTING COMMANDS.

#ROUTINES CALLED: HDRDELIM, NUM.PROC, SCANFRMT

#CALLED BY: MSGPROC IN COMMAND DISPATCH

     HDRDELIM #CHECK FOR A HEADER DELIMITER
     REPEAT
          NUM.PROC
          EXITR WHEN MPERRCD <> 0
          BYTE[ CADDR ] = FIX( NUM.ARG )
          CADDR = CADDR + 1
          IF INCHAR = SPACE THEN
               SCANFRMT #EXPECT EOM OR MORE ARGUMENTS
          ELSE
               EXITR WHEN INCHAR <> COMMA
          ENDI
          UNTIL EOM
     ENDR

RETURN
```

The third example is a command which has a single optional argument. This is is different than the example above because of the special case of format characters (spaces) which are allowed at the end of a message. The DM5010 has several actual commands of this form -- this example shows the general structure.

```
XXXX.CMD

     IF NOT( EOM ) AND ( INCHAR = SPACE ) THEN
          #COMMAND HAS A VALID HEADER DELIMITER
          NUM.PROC
          IF MPERRCD = MISSARG THEN
               MPERRCD = 0

               SUBROUTINE CALL TO PERFORM FUNCTION REQUIRED
               WHEN ARGUMENT IS NOT PRESENT

          ENDI

          PERFORM FUNCTION USING ARGUMENT VALUE FOUND IN NUM.ARG

     ELSE #OPTIONAL ARGUMENT NOT SENT

          SUBROUTINE CALL TO PERFORM FUNCTION REQUIRED
          WHEN ARGUMENT IS NOT PRESENT

     ENDI

RETURN
```

The final example for this section is to implement a command which decodes a character argument.

```
RQS.CMD

#ROUTINES CALLED: HDRDELIM, CHR.PROC

#CALLED BY: MSGPROC IN THE COMMAND DISPATCH

     HDRDELIM #CHECK FOR HEADER DELIMITER
     CHR.PROC #DECODE CHARACTER ARGUMENT
     IF MPERRCD = 0 THEN
          NSP = TRUE
          IF INDEX = OFF.IDX THEN
               P.RQS = FALSE
          ELSEIF INDEX = ON.IDX THEN
               P.RQS = TRUE
          ELSE
               MPERRCD = CMDARG
          ENDI
     ENDI

RETURN
```

Rules for Changing Pending Settings

The code which acutally changes the value of the Pending Settings buffer is probably a subroutine that is shared by the Message Processor and the Key Processor. There are several rules which must be followed in order to assure proper operation.

Since there is often some checking done on the parameter to be set, a method of passing error codes back to the Message Processor or Key Processor must be arranged so that each task will report only errors detected when they called the setting routine. To handle this a variable shared by both tasks (**ERR**) is cleared on entry to the subroutine and is assigned an eventcode if an error is detected. The command handler then must check this variable and report it to the GPIB Driver before it suspends (which prevents the Key Processor from calling a routine which would change its value while the Message Processor was suspended. The way the error is reported is to assign it to **MPERRCD** and return to the command dispatcher.

Once the parameters to be set in the **Pending Settings** buffer have been checked and are determined to be valid, the New Settings Pending flag (NSP) must be set before they are assigned to the **Pending Settings** buffer. This is done to assure that the system can determine whether **Pending Settings** are equal to **Current Settings** simply by testing the **NSP** flag.

Rules for Output Commands

The output commands should make sure that the data for output is saved in an area which will not be altered by another task. For queries, the settings are automatically saved by the Message Processor before the dispatch. This solves the problem of the settings changing while the query response is generated, but since it is not the form in which the data is output, the designer should assure that the variables used to store the characters for output are not shared by other tasks.

Example of an output command (query):

```
RQS.QRY

#ROUTINES CALLED: OUTP.HDR, OUTP.ARG

#CALLED BY: MSGPROC AND SET.QRY IN COMMAND DISPATCH

     #SETTINGS ARE SAVED BY THE MESSAGE PROCESSOR
     TAGPTR = ^RQS.TAG
     OUTP.HDR #OUTPUT THE HEADER
     IF Q.RQS THEN
          TAGPTR = ^ON.TAG
     ELSE
          TAGPTR = ^OFF.TAG
     ENDI
```

```
OUTP.ARG

RETURN
```

The second example given for output commands is the command handler which generates the error query response.

The error query (ERR?) provides the operator a means of obtaining additional information corresponding to the most recent status byte sent with the **RQS** message asserted. After a status byte has been sent via the serial poll response, **CRNEVENT** (the event code for that status byte) is transferred to **EQRES** (the error query response code). Then when the instrument receives the "ERR?" command is uses **EQREC** to find the 16 bit error query response in the **Status Table** and converts this to ASCII decimal for output.

The error query also allows the user to determine error conditions when the SRQ is disabled with the RQS OFF command. In this mode, the error query returns the highest priority error condition pending, and clears the condition so that it is not reported again.

Problems:

Because the **TMS 9914** does not tell the processor when it sends a status byte with **rsv** = 0, the **EQRES** is updated only for those status bytes with **RQS** asserted.

If two errors occur simultanously, the controller may poll the instrument for the second status before it issues the ERR? query. In this case it looses the information about the first error reported.

Pseudo code for the **ERRO.QRY**.

```
ERRO.QRY

#ROUTINES CALLED: OUTP.HDR, OUTP.INT, PUTBYTE

#CALLED BY: MSGPROC IN COMMAND DISPATCH

     IF NOT( C.RQS ) THEN #SRQ DISABLED
          #GET HIGHEST PRIORITY FROM PENDSTAT
          TABLEPTR = ^ PENDSTAT (1)
          WHILE BYTE[ TABLEPTR ] = 0 DO
               TABLEPTR = TABLEPTR + 1
          ENDW
          EQRES = BYTE[ TABLEPTR ]
          IF TABLEPTR <> ^ NDDSTAT THEN
               BYTE[ TABLEPTR ] = 0
          ENDI
     ENDI

     #OUTPUT THE RESPONSE
     NUM.ARG = 0.0
     IF EQRES < 80H THEN #EVENTCODE
          NUM.ARG = FLT( EQTBL ( EQRES ) )
     ENDI
```

```
    EQRES = 80H
    TAGPTR = ^ERRO.TAG
    OUTP.HDR
    OUTP.INT #OUTPUT INTEGER VALUE OF NUM.ARG
    OUTCHAR = SEMICOLN
    PUTBYTE

RETURN
```

Rules for Operational Command Handlers

The only general rule that applies to operational commands is that they must test the **MSGREM** flag if they suspend (for decoding arguments, etc.) since its value may change and all operational commands are **REMOTE ONLY** type.

2.6 Utility Routines for Command Handlers

HDRDELIM

This subroutine is used to check for the header delimiter. If the character in **INCHAR** is not a valid header delimiter, it sets MPERRCD with the header delimiter error eventcode.

```
HDRDELIM

#ROUTINES CALLED: NONE

#CALLED BY: COMMAND HANDLERS WITH ARGUMENTS (ADDR.CMD, BYTE.CMD, ETC.)

     IF INCHAR <> SPACE THEN
        MPERRCD = HDRDLM
     ENDI

RETURN
```

ARGDELIM

This subroutine is used to check for argument delimiters. If the character in **INCHAR** is not a valid argument delimiter (comma or space), then it sets **MPERRCD** with the argument delimiter error eventcode.

```
ARGDELIM

#ROUTINES CALLED: NONE

#CALLED BY: COMMAND HANDLERS WITH MULTIPLE ARGUMENTS

     IF ( INCHAR <> COMMA ) AND ( INCHAR <> SPACE ) THEN
        MPERRCD = ARGDLM
     ENDI

RETURN
```

CHR.PROC

The Character Argument Processor is used to decode character arguments from the Input Buffer. If an error was detected in a previous routine, then it simply returns. Otherwise, it skips over any leading format characters and then uses the table search routine to determine the index of the argument. If the character string in the buffer does not match one of the character arguments in the table, the command error eventcode is stored in MPERRCD.

```
CHR.PROC

#ROUTINES CALLED: SCANFRMT, TABLSRCH, REM.TST

#CALLED BY: COMMAND HANDLERS WITH CHARACTER ARGUMENTS (RQS.CMD)

#INPUT: TABLEPTR - POINTER TO THE DESIRED ARGUMENT TABLE

#OUTPUT: TYPE - CONTAINS THE ARGUMENT TYPE INFORMATION FROM
                #THE TAG BYTE IN THE ARGUMENT TABLE ENTRY.  TYPE
                #CONTAINS ATHE BYTE AFTER 4 ASR OPERATIONS. IF THE
                #4 MSB'S OF THE TAG ARE 0 OR F THEN THE TYPE VARIABLE
                #RETURNED IS A 00H OR FFH WHICH CAN BE DIRECTLY
                #ASSIGNED TO ANY LOGICAL VARIBLE.  THE 4 MSB'S OF
                #THE TAG BYTE MAY ALSO CONTAIN AN ARRAY INDEX FROM
                #0 TO 7.

     IF MPERRCD = 0 THEN
          SCANFRMT
          IF NOT( EOM ) THEN
               INDEX = 0
               TABLSRCH
               IF TYPE <> 0 THEN
                    ARGTYPE = ASR( TYPE )
               ELSE #VALID ARGUMENT NOT FOUND
                    MPERRCD = CMDARG
               ENDI
               REM.TST
          ELSE
               MPERRCD = MISSARG
          ENDI
     ENDI

RETURN
```

NUM.PROC

The numeric argument processor is used to decode numeric arguments from the **Input Buffer**. It returns to the calling routine if an error had already been detected. If no error exists, it skips over any leading format characters and then prescans the **Input Buffer** to insure that the entire number string is in the buffer. This is required to prevent the floating point **REALVAL** routine from suspending during numeric string conversion, which removes the requirement that it be re-entrant. The value of the number is returned as a floating point value in **NUM.ARG**. If an error is detected in the conversion, the appropriate eventcode is returned in **MPERRCD**.

```
NUM.PROC

#ROUTINES CALLED: SCANFRMT, GETCHR, REALVAL, REM.TST

#CALLED BY: COMMAND HANDLERS WITH NUMERIC ARGUMENTS

     IF MPERRCD = 0 THEN
          SCANFRMT
          IF NOT( EOM ) THEN
               REPEAT
                    GETCHR
                    EXITR WHEN MPERRCD <> 0
                    UNTIL EOM OR INVALCHR
               THENDO
                    FPERR = 0
                    EOM = FALSE
                    NUM.ARG = REALVAL ( BYTE[ BUPTR ] )
                    IF ( FPERR <> 0 ) OR ( GBPTR <> PTRASCII ) THEN
                         MPERRCD = CMDARG
                         FPERR = 0
                    ENDI
               ENDR
          ELSE #NULL ARGUMENT
               MPERRCD = MISSARG
          ENDI
          REM.TST
     ENDI

RETURN
```

BIN.PROC

```
#THIS ROUTINE PROCESSES BINARY BLOCK ARGUMENTS

#INPUT:
     #TABLEPTR = POINTER TO THE DESTINATION AREA FOR THE BINARY DATA.
     #BINCNT = THE EXPECTED BYTECOUNT

#OUTPUT:
     #A BLOCK OF BINARY DATA AT THE SPECIFIED LOCATION.
     #INCHAR = CHARACTER AFTER THE CHECKSUM.

#ROUTINES CALLED: SCANFRMT, GETBYTE, GETBIN, FREESPACE, REM.TST

#CALLED BY: COMMAND HANDLERS THAT INPUT BINARY DATA

     IF MPERRCD = 0 THEN
          SCANFRMT
          IF NOT( EOM ) THEN
               GETBYTE
               IF INCHAR = PERCENT THEN
                    CKSUM = 0 #INITIALIZE CHECKSUM
                    GETBIN #READ 1ST BYTE AND ADD TO CKSUM
                    BCNT.HI = INCHAR #SAVE BYTE COUNT IN BNCT
                    GETBIN
                    BCNT.LO = INCHAR
                    IF BCNT = BINCNT THEN #BYTE COUNT IS CORRECT
                         BCNT = BCNT - 1
                         WHILE BCNT <> 0 DO
                              FREESPACE
                              GETBIN
                              EXITW WHEN MPERRCD <> 0
                              BYTE[ TABLEPTR ] = INCHAR
                              TABLEPTR = TABLEPTR + 1
                              BCNT = BCNT - 1
                         THENDO
                              GETBIN #READ CHECKSUM BYTE
                              IF CKSUM <> 0 THEN #CHECKSUM ERROR
                                   MPERRCD = CKSUMERR
                              ELSE #READ BYTE FOLLOWING CHECKSUM
                                   GETBYTE
                                   REM.TST #CHECK FOR REMOTE STATE
                              ENDI
                         ENDW
                    ELSE MPERRCD = BCNTERR #REPORT BYTE COUNT ERROR
                    ENDI
               ELSE #REPORT PERCENT CHARACTER NOT FOUND
               MPERRCD = CMDARG
               ENDI
          ELSE MPERRCD = MISSARG #REPORT MISSING ARGUMENT
          ENDI
     ENDI

RETURN
```

GETBIN

```
#THIS ROUTINE READS THE NEXT BYTE FROM THE INPUT BUFFER AND ADDS
#IT TO THE CKSUM VARIABLE.  IF AN EOM OCCURS IN LINE-FEED MODE,
#THEN CONTINUE BINARY INPUT IS CALLED TO START THE INPUT GOING
#AGAIN.  IF EOM IS STILL TRUE AFTER CONTBIN IS CALLED, THEN AN
#ARGUMENT ERROR IS ISSUED.  THIS ROUTINE IS USED FOR BINARY BLOCK
#DATA INPUT, SO A REAL EOM SHOULD NEVER OCCUR.

#INPUT:
     #CKSUM - THE BINARY CHECKSUM (8 BIT)

#OUTPUT:
     #CKSUM - THE BINARY CHECKSUM
     #INCHAR - THE BYTE READ FROM THE INPUT BUFFER

#ROUTINES CALLED: GETBYTE, CONTBIN

#CALLED BY: BIN.PROC


     GETBYTE
     IF EOM AND LFMODE THEN
          CONTBIN
          GETBYTE
     ENDI

     IF EOM THEN MPERRCD = CMDARG #REPORT EOM ERROR
     ELSE CKSUM = CKSUM + INCHAR
     ENDI

RETURN
```

SEND.BIN

```
#THIS ROUTINE SENDS A BINARY BLOCK ARGUMENT TO THE GPIB OUTPUT BUFFER

#INPUT:
     #QRY.BUFR(BINCNT) - BINCNT BYTES OF DATA TO BE OUTPUT

#OUTPUT: NONE

#ROUTINES CALLED: PUTBYTE

#CALLED BY: COMMAND HANDLERS THAT OUTPUT BINARY BLOCKS


     OUTCHAR = PERCENT #OUTPUT PERCENT SIGN
     PUTBYTE
     OUTCHAR = 0 #OUTPUT HIGH BYTE OF BYTE COUNT
     PUTBYTE
     OUTCHAR = BINCNT #OUTPUT LOW BYTE OF BYTE COUNT
     PUTBYTE
     TABLEPTR = ^ QRY.BUFR (1) #INITIALIZE BINARY OUTPUT POINTER
     CKSUM , BCNT.LO = BINCNT #INITIALIZE CHECKSUM AND LOOP COUNT

     REPEAT
          BCNT.LO = BCNT.LO - 1
          EXITR WHEN BCNT.LO = 0
          OUTCHAR = BYTE[ TABLEPTR ]
          CKSUM = CKSUM + OUTCHAR
          PUTBYTE
          TABLEPTR = TABLEPTR + 1
     ENDR

     OUTCHAR = -( CKSUM ) #NEGATE CHECKSUM AND OUTPUT
     PUTBYTE

RETURN
```

REM.TST

This subroutine must be called by any **REMOTE ONLY** command handler that suspends (argument processor may suspend). **REM.TST** must be called after the last suspend has been executed. The reason for this is that while the **Message Processor** is suspended, the **Key Processor** may receive an **rtl** and execute a command handler that alters the **Pending Settings**. Therefore the state of the **Pending Settings** buffer cannot be guaranteed when the **Message Processor** resumes execution. Since an **rtl** sets **MSGREM** , **MSGREM** = TRUE indicates that **Pending Settings** are still intact.

#ROUTINES CALLED: NONE

#CALLED BY: ALL REMOTE ONLY COMMAND HANDLERS

REM.TST

```
     IF REMTYPE AND NOT( MSGREM ) THEN
          MPERRCD = REMONLY
     ENDI

RETURN
```

OUTP.HDR

This routine is used to output the header for a query response. As input it requires that the variable **TAGPTR** be assigned a pointer to the tag byte of the command in the command header table and the **INDEX** of the command header which is to be output.

HDRLEN is an array containing the number of characters to be output for the query response.

#ROUTINES CALLED: OUTP.CHR, PUTBYTE

#CALLED BY: ALL QUERY COMMANDS

OUTP.HDR

```
     TAGPTR = TAGPTR + 1
     LENGTH = HDRLEN ( INDEX - NUM.CMDS )
     OUTP.CHR
     OUTCHAR = SPACE
     PUTBYTE

RETURN
```

OUTP.ARG

This subroutine is used to put a character argument and the message unit delimiter into the **Output Buffer**. It requires that the variable **TAGPTR** be assigned the address of the tag byte of the argument.

```
OUTP.ARG

     LENGTH = TBLBYTE [ TAGPTR ] AND 0FH
     TAGPTR = TAGPTR + 1
     OUTP.CHR
     OUTCHAR = SEMICOLN
     PUTBYTE

RETURN
```

OUTP.CHR

This routine outputs the number of bytes indicated by the variable **LENGTH** from the character string pointed to by **TAGPTR**.

```
OUTP.CHR

     REPEAT
          OUTCHAR = TBLBYTE [ TAGPTR ] AND 7FH
          PUTBYTE
          TAGPTR = TAGPTR + 1
          LENGTH = LENGTH - 1
          UNTIL LENGTH = 0
     ENDR

RETURN
$
```

3.1 Functional Description

The Key Processor task handles all input from the instrument front panel. It polls the hardware for the key (button) pushes and when a legal set of keys are entered, performs the function specified. The operations are performed by putting the new setting into the **Pending Setting Buffer**. Then the pending settings are checked for validity. If the new setting introduced an error, the setting remains in the display, the error indicator is lit, and the parameter which conflicts with the new entry (if any) is displayed by blinking its indicator light. When the validity check passes, the **HWSETR** flag is set and the execution of the function requested is initiated on the next pass through the **Hardware Settings** task.

Front Panel Syntax

The **ID** key is always enabled and may be used at any time without effecting the operation of the instrument. The **ID** key displays the instrument's GPIB address on the front panel display and generates an **SRQ** interrupt if that function is enabled by the controller.

When entering multiple keystroke commands, invalid keys are ignored and the entry may be continued. If an error condition is detected in an entry, it can be cleared with the **CLEAR** or **CLEAR ENTRY** keys. All other keys are ignored (except for the **ID** key). The clear entry key puts the display back to the current setting of the selected parameter. If there is no current setting to display, then a value of zero is displayed.

When a number is entered that is out of the instruments range, the error light is lit and the display retains the out-of-range value. Keyboard entries are rounded off to the nearest unit of resolution before being checked for out-of-range. A number entered that is between two valid parameter settings is rounded off to the nearest unit of resolution (this is consistent with operation from GPIB interface).

Numbers displayed on the front panel are right justified to the least significant digit of resolution. On instruments with values which require exponents, an exponent is chosen such that the mantissa is greater than or equal to 1 and the number of mantissa digits matches the resolution of the displayed parameter -- that is, a parameter with 4 digit resolution would look like "1.000E3" as opposed to "001.0E3" with suppressed leading zeros. The reason for this is that when a parameter is incremented or decremented, it is desirable to have the decimal point shift as little as possible.

As digit keys are pushed, the display fills from the left to right, starting with a digit position which allows only enough digits for the selected parameter's resolution -- the right-most digit is filled by the Nth digit of an N digit parameter. Digit keys pushed after the display is filled are ignored (except in the **DM5010**).

On instruments with exponents, digit keys pushed after the **EEX** (Enter EXponent) key fill the display from right to left. Digit keys

pushed after the exponent display is filled roll the exponent one digit left. If EEX is the first numeric key pushed, then "1.0" is displayed in the mantissa (left justified).

Non-numeric keys do not perform an implied enter function, the **ENTER** key must be explicitly pressed by the user. This is done because it is dangerous for a signal source instrument to make the assumption that the user wants the number entered, when he may have simply changed his mind about which parameter he wants to program. When a number is keyed in but not entered and a non-numeric key is pressed, (except for **CLEAR ENTRY** and **ID**), then the non-numeric key is ignored. At this point the user may push a numeric key, **ENTER** key, **CLEAR ENTRY** key or **ID**.

State Machine

The **Key Processor** is a sequential state machine which validates key sequences and dispatches to service routines. The state is determined by a combination of flags, pointers and program control flow.

KEYPROC repeats until the instrument goes to the **REMOTE** GPIB state which causes an **RLC** interrupt. The **RLC** interrupt service routine reinitializes the **KEYPROC** by resetting the **KEYPROC** task stack.

Each time around the outer repeat loop, **KEYPROC** may service one key, no keys (may ignore numeric), or a sequence of several keys. The series of IF-THEN-ENDI control blocks allows an easy way to handle key sequences without having to set many flags or save copies of previous keys or test for previous keys or having to conditionally bypass the **GETKEY** at the top of the repeat loop. Because the indivudual instruments have different requirements for the control of front panel features, a description of the DM key processor and one used for both the FG and PS are included.

3.2 DM5010 Key Processor

DM5010 Front Panel Indicators

The DM has a signed four and one-half digit seven segment numeric display, eleven backlit ennunnciators, and eleven lit push buttons. The numeric display may show five kinds of data:

1. The result of a measurement (may include math calculations).
2. The value of a programmable parameter.
3. The numeric entry for altering parameters.
4. An error message.
5. The GPIB listen address.

A variable named **DSPMODE** controls which of the first four kinds of data is displayed. **DSPMODE** is normally set by the front panel buttons, but if the processor encounters trouble it may display an error message. If the DM is remotely controlled by the GPIB, the numeric display normally shows the result of a measurement.

The DM may display numbers from one ten millionth (1. E-7 = ".0001m") to ninteen-billion nine-hundred ninty-nine million (1.9999 E 10 = "19999M") by shifting the decimal point and lighting the "m", "K", or "M" backlit ennunciators.

To indicate over range due to a measurement overflow, the display flashes "19999" and "ERROR" leaving the decimal point and m, K or M steady. If the over range is due to a calculation, the display flashes "19.E99" and the "ERROR" light.

Front Panel Keystrokes

The DM has 40 keys (buttons) which are grouped into the following types:

TYPE	KEYS
ID	INST ID
FUNCTION	DCV, OHM, DIODETEST, ACV, AC+DCV, NULL, 10HZ, AUTO, STEP, RUN, TRIG, FAST, AVERAGE, RATIO((X-B)/A), DBM, DBR, COMPARE, REAR.
RECALL	RECALL
PARAMETER	A, B, LIMIT1, LIMIT2, N,R
ENTER	ENTER
CLEAR	CLEAR
NUMERIC	CHANGE SIGN, DECIMAL POINT, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

The INST ID key always responds by displaying the GPIB listen address as set by the internal switches. This key is never dsabled so that the user may assure himself that the processor has not crashed. The ID key is the only key serviced inside the GETKEY subroutine, because it must always respond and because it is the only key that must be monitored for release. When the ID key is released, the front panel displays what was displayed before the ID key was pushed. The ID key also generates a user SRQ if the GPIB commands "RQS ON; USER ON" have been received.

All of the function keys have single stroke responses. The function keys are ignored during recall or numeric entry sequences; any other time, the display provides feedback to indicate that a function key was serviced.

The RECALL key is used to display the value of any of the programmable parameters. Pushing the RECALL key begins the RECALL sequence. The key immediately following the RECALL key must be a parameter key or the CLEAR key (function keys and numeric keys are ignored). Pressing a parameter key causes the processor to display the value of the selected parameter. Additional parameters may be displayed without additional RECALL key strokes. Numeric keys may be entered to change the value of a parameter, as described below. The RECALL sequence is terminated by any function key or the CLEAR key.

The value of a parameter may be equated to the result of a measurement or may be set with the numeric keys. The sequence to equate a parameter to the result of a measurement is to obtain the desired measurement in the display, then press a parameter key followed

by ENTER. The sequence to set the value of a parameter with the numeric eys is to select a parameter followed by the appropriate numeric keys and ending with ENTER.

The function of the CLEAR key is to cancel a sequence and to control the data to be displayed. The CLEAR key is misnamed as it does not clear the display nor does it alter any of the parameters. When pressed during a numeric entry sequence, the clear key displays the value of the selected parameter and cancels the numeric entry. At this point the processor may accept more numeric keys, or any other key. If the CLEAR key is pressed a second time, the DM displays measurements and numeric keys are ignored. Thus pressing the CLEAR key twice returns the front panel to its normal state, unless the GPIB controller has locked out the front panel.

Description of KEYPROC

The **Key Processor** is a sequential state machine which validates key sequences and dispatches to service routines. The state is determined by a combination of flags, pointers and program control flow.

FLAGS		MEANING
STATERCL	=TRUE	Set when the RECALL key is serviced to enable displaying the value of a parameter when a parameter key is pressed. Cleared by the function keys and CLEAR key.
	=FALSE	To allow the display to continue to show the result of a measurement after a parameter key is pressed so that the parameter may be equated to the measurement by pressing ENTER.
STATENUM	=TRUE	Set by parameter keys to enable the numeric keys.
	=FALSE	Cleared by function keys or CLEAR key to disable numeric entry.

POINTERS	MEANING
PTRPARM	Points to the parameter selected for display or edit. Set by parameter keys.
DISPMODE	Selects the kind of data to be shown in the numeric display (measurement, parameter, numeric, error message, GPIB listen address) set by all keys.

KEYPROC

Pseudo code for the DM Key Processor

```
KEYPROC

     DISPMODE = DISPMEAS
     STATERCL = FALSE #DISABLE RECALL
     STATENUM = FALSE #DISABLE NUMERIC ENTRY

     REPEAT #UNTIL GPIB REMOTE INTERRUPT RESETS KEYPROC
          GETKEY #RETURNS KEYCODE AND TYPE, WAITS FOR HWSETR FALSE
          IF TYPE = TYPEFUNC THEN #SERVICE FUNCTION KEYS
               STATERCL = FALSE
               STATENUM = FALSE
               DISPMODE = DISPMEAS
               CALL COMMAND HANDLER ( KEYCODE )
          ENDI

          SAVE TRIGGER MODE STATUS
               # FOR ENTERING A MEASUREMENT INTO A PARAMETER

          IF KEYCODE = KEYRECAL THEN #SERVICE RECALL KEY
               STATERCL = TRUE
               REPEAT #WAIT FOR A PARAMETER KEY
                    GETKEY
                    UNTIL TYPE = TYPEPARM OR KEYCODE = KEYNULL OR
                    KEYCODE = KEYCLEAR
               ENDR
               IF KEYCODE = KEYNULL THEN #ALLOW RECALL NULL
                    TYPE = TYPEPARM
               ENDI
          ENDI

          IF TYPE = TYPEPARM THEN #SERVICE PARAMETER KEYS
               PTRPARM =ADDRESS(PARAMETER( KEYCODE )) #PONT TO
               PARAMETER
               LOAD VALUE OF SELECTED PARAMETER INTO PARAMETER PORTION
               OF DISPLAY BUFFER
               IF STATERCL = TRUE THEN #RECALL PARAMETER
                    DSPCHNG = TRUE
                    DISPMODE = DISPPARM
               ELSE
                    DISABLE AUTOMATIC TRIGGERS #HOLD DISPLAY
                    REPEAT
                         GETKEY
                         UNTIL TYPE = TYPENUM OR KEYCODE = KEYENTER OR
                         KEYCODE = KEYRCL OR KEYCODE = KEYCLEAR
                    ENDR
                    IF KEYCODE = KEYRCL THEN
                         STATERCL = TRUE
                         DISPMODE = DISPPARM #RECALL PARAMETER
                         DSPCHNG = TRUE
```

```
            ENDI
        ENDI
        STATENUM = TRUE #ENABLE NUMERIC ENTRY
    ENDI

    IF KEYCODE = KEYENTER AND STATENUM = TRUE THEN #ENTER A
    MEASUREMENT INTO A PARAMETER
        #NOTE; THIS KEYENTER IS NOT THE ENTER FOR NUMERIC ENTRY
        PARAMETER( PTRPARM ) = VALUE IN DISPLAY
    ENDI
    IF KEYCODE = KEYCLEAR THEN
        STATENUM = STATERCL = FALSE #RETURN TO DISPLAYING
        MEASUREMENTS.
        DISPMODE = DISPMEAS #DISPLAY MEASUREMENTS
    ENDI
    IF TYPE = TYPENUM AND STATENUM = TRUE THEN #SERVICE NUMERIC
    ENTRY
        CALL NUMERIC ENTRY #CALLS GETKEY AND RETURNS ON CLEAR OR
        ENTER
    ENDI

    RESTORE TRIGGER MODE STATUS
ENDR
```

Subroutines Called

GETKEY Suspends until a key is ready. Returns **KEYCODE** and **TYPE (KEYCODE)**. Suspends if hardware settings are being updated.

Command handlers The command handlers service the functioni keys, including changing hardware settings, triggering a measurement, and setting front panel indicators. Most of these routines share common code with the **Message Processor.**

Numeric Entry Displays digits as the numeric keys are pressed, calls **GETKEY**, converts the number entered into floating point format, and stores it into the parameter selected by **PTRPARM.**

3.3 FG5010 and PS5010 Key Processor

Pseudo code for the FG key processor.

```
FGKEYPROC

     BLINK = 0
     RTLTIME = 0
     NESTATE = 0 #RESET NUMERIC ENTRY
     RESET KEYBOARD
     RESET DISPLAY
     SUSPEND
     CLEAR INTERRUPT MASK

     REPEAT FOREVER
          GETKEY #GETKEY WAITS FOR HWSETR = FALSE

          CASE KEYTYPE OF

               [ FUNCTION ]
                         CALL COMMAND HANDLER( KEY )
                         IF NOT( ERR ) THEN
                              KPEXEC
                         ELSE #FLASH ONLY ERROR LIGHT
                              KPERROR ( 0 )
                         ENDI

               [ INCDEC ]
                         IF ACTPARAM <> STEPSIZ THEN
                              INCRDECR
                         ENDI

               [ NUMERIC ]
                         NUMENTRY
                         IF NESTATE <> 0 THEN #PERFORM FUNCTION
                              NESTATE = 0 #INDICATE TO OTHER TASKS THAT
                              THE NUMERIC ENTRY
                                        #IS COMPLETE, SO THE DISPLAY
                                        MAY NOW BE CHANGED.
                              NUM.ARG = VAL( NEBUFFER )
                              CALL COMMAND HANDLER( ACTPARAM )
                              IF NOT( ERR ) THEN
                                   KPEXEC
                              ELSE #FLASH ERROR LIGHT
                                   KPERROR ( 0 )
                              ENDI
                         ENDI

          ENDC
          IF RTLTIME <> 0 THEN RTLTIME = 1 ENDI

     ENDR
```

KPEXEC

This routine verifies the validity of any pending settings that came from the **Key Processor**, and then executes those settings. If a settings conflict is found, then the conflicting setting is passed to the key processor error handler. The **KPERROR** routine flashes the conflicting parameters lights until the user pushes CLEAR or the instrument goes to **REMOTE**.

Pseudo code for the FG Key Processor Execution Routine.

```
KPEXEC

     CONFLICT = 0 #IN CASE OF REMOTE CONTROL
     IF FPCNTRL AND NSP THEN
          VERIFY PENDING SETTINGS
          IF CONFLICT = 0 THEN
               HWSETR = TRUE
          ELSE
               KPERROR ( CONFLICT )
          ENDI
     ENDI

RETURN
```

KPERROR

Pseudo code for the FG Key Processor Error Routine.

```
KPERROR ( KPERR ) #KPERR IS PASSED PARAMETER

     TURN ON ERROR LIGHT IN DISPLAY BUFFER
     DSPCHNG = TRUE

     #BLINK CONFLICTING SETTINGS LIGHTS (IF ANY)
     IF KPERR <> 0 THEN
          BLINK = BLNKRATE
     ENDI

     REPEAT
          GETKEY
          #ONLY CLEAR KEY CANCELS AN ERROR
          UNTIL KEY = CLRKEY
     ENDR

     #RESTORE DISPLAY TO OLD VALUE(S) AND TURN OFF ERROR LIGHTS.
     BUILD DISPLAY BUFFER
     BLINK = 0

RETURN
```

Numeric Entry Variables

RESOLN Is the resolution of the active parameter in number of digits.

NECOUNT Is the number of mantissa digits entered.

NESTATE Is the state variable for the numeric entry process. **NESTATE** <> 0 indicates that numeric entry is in progress, and that the display buffer is not to be changed. However, any portion of the display buffer not used for numeric entry may be changed. **NESTATE** indicates the following numeric entry states:

0 Idle state

1 Mantissa before decimal point

2 Mantissa after decimal point

3 Mantissa full

4 After enter exponent

NEBUFFER Is a buffer which holds the ASCII equivalents of the keys pushed. This buffer is copied into the display buffer after each key is pushed, to give the user feedback as to what has been keyed in. **NEBUFFER** is also passed to the ASCII-Floating Point converter to get a numeric argument when the user pushes the ENTER key. This buffer must be filled with blanks when numeric entry begins so that unused digits are blanked and so that the ASCII-Floating Point routine terminates properly.

NEPTR Is a pointer to the next empty position in the numeric buffer.

Numeric Entry State Table

STATE >	0	1	2	3	4
DIGIT	DISPLAY DIGIT ST = 1	DISPLAY DIGIT IF FULL THEN ST=3	DISPLAY DIGIT IF FULL THEN ST=3	IGNORE	ROLL EXP DIGIT
DP	DISPLAY "0." ST = 2	DISPLAY "." ST = 2	IGNORE	IGNORE	IGNORE
EEX	DISPLAY "1.0E " ST = 4	DISPLAY "E " ST = 4	DISPLAY "E " ST = 4	DISPLAY "E " ST = 4	IGNORE
CHS	DISPLAY "-" ST = 1	IF ND<4 THEN TOGGLE SIGN	IF ND<4 THEN TOGGLE SIGN	IF ND<4 THEN TOGGLE SIGN	TOGGLE EXP SIGN
ENTER	RETURN	RETURN	RETURN	RETURN	RETURN
CLEAR	ST = 0 RESET DISPLAY RETURN	ST = 0 RESET DISPLAY RETURN	ST = 0 RESET DISPLAY RETURN	ST = 0 RESET DISPLAY RETURN	ST = 0 RESET DISPLAY RETURN

Pseudo code for Numeric Entry

```
NUMENTRY

     NESTATE = 0
     NEPTR = ^ NEBUFFER + 4 - RESOLN
     FILL NEBUFFER WITH BLANKS
     NECOUNT = 0

     WHILE KEY <> ENTER DO
          CASE KEY OF

               ["0" TO "9"]
                         IF NESTATE = 0 THEN
                              NESTATE = 1
                         ENDI
                         IF NESTATE <> 3 THEN
                              STORECHR ( KEY )
                              IF NESTATE > 3 THEN
                                   NEPTR = NEPTR - 1
                              ELSE
                                   IF NECOUNT = RESOLN THEN
                                        NESTATE = 3
                                   ENDI
                              ENDI
                         ENDI

               ["."]
                         IF NESTATE < 2 THEN
                              IF NESTATE = 0 THEN
                                   STORCHR ("0")
                              ENDI
                              NESTATE = 2
                                   IF NECOUNT = RESOLN THEN
                                        NESTATE = 3
                                   ENDI
                              STORCHR (".")
                         ENDI

               [ EXX ]
                         IF NESTATE < 4 THEN
                              IF NESTATE = 0 THEN
                                   STORCHR ("1")
                                   STORCHR (".")
                                   IF NECOUNT < RESOLN THEN
                                        STORCHR ("0")
                                   ENDI
                              ENDI
                              STORCHR ("E")
                              NEPTR = NEPTR + 1
                              NESTATE = 4
                         ENDI

               [ CHS ]
```

```
                    IF NESTATE = 0 THEN
                         NESTATE = 1
                    ENDI
                    IF NESTATE = 4 THEN
                         PTR = NEPTR - 1
                    ELSE
                         PTR = ^ NEBUFFER + 3 - RESOLN
                    ENDI
                    IF RESOLN < 4 THEN
                         IF NEBUFFER ( PTR ) = SPACE THEN
                              NEBUFFER ( PTR ) = "-"
                         ELSE
                              NEBUFFER ( PTR ) = SPACE
                         ENDI
                    ENDI

          [ CLEAR ]
                    NESTATE = 0
                    RESET DISPLAY
                    EXITW

     EXITC

     DISPLAY NUMBER IN NUMERIC BUFFER
     GETKEY

ENDW

RETURN
```

STORCHR

Pseudo code for the Store Character routine.

```
STORCHR ( INCHAR )

     NEBUFFER ( NEPTR ) = INCHAR
     NEPTR = NEPTR + 1
     IF INCHAR > = "0" AND INCHAR < = "9" THEN
          NECOUNT = NECOUNT + 1
     ENDI

RETURN
```

GETKEY

The **Key Processor** initialization routine must set RTLTIME = 0.

Pseudo code for the **GETKEY** routine:

```
GETKEY

     REPEAT
          KEY = 0
                         #MAKE SURE THAT KEY<>IDKEY SO DISPLAY
                         #UPDATE CAN CHANGE THE DISPLAY
          WHILE (KEY NOT AVAILABLE) DO
               SUSPEND
               IF RTLTIME <> 0 THEN
                    RTLTIME = RTLTIME - 1
                    IF RTLTIME = 0 THEN
                         #RE-ENABLE A TRANSITION TO REMOTE
                         STORE RTL AUXILLARY COMMAND WITH C/S = 0
                    ENDI
               ENDI
               BLNKLITE
          ENDW
          KEY = KEYCODE FROM HARDWARE
          IF KEY = IDKEY THEN
               IDPROC
          ELSE
               RTLPROC
          ENDI
     UNTIL ( KEY <> IDKEY ) AND NOT ( REMOTE )
     #TMS 9914 CHIP IGNORES RTL IN LOCKOUT STATE, SO REMOTE REMAINS
     TRUE
     RTLTIME = (5 SECONDS) / (TIME FOR ONE PASS THROUGH THE MONITOR)

     KEYMAP

     #WAIT FOR HARDWARE SETTINGS CHANGE IN PROGRESS TO COMPLETE
     #BEFORE RETURNING THE KEY TO THE CALLING ROUTINE.
     WHILE HWSETR = TRUE
          SUSPEND
     ENDW

RETURN
```

Note: **GETKEY** must call **RTLPROC** unconditionally when servicing any key. The reason for this is that if it is only called when the instrument is in remote, a transition to **LOCAL** caused by a **GTL** or REMOTE ENABLE false transitionfollowed by a key press would not set **FPCNTRL** since that is done in **RTLPROC**. Since **FPCNTRL** is clear and the **Key Processor** thinks it has control of the **Pending Setting** buffer, it is possible for both the **Message Processor** and **Key Processor** to write into the **Pending Settings** buffer at the same time. This must be prevented, and the easies way is to call **RTLPROC** for each front panel key press.

BLNKLITE

Blinking light routine

```
BLNKLITE

     IF BLINK <> 0 THEN BLINK = BLINK - 1
          IF BLINK = 0 THEN
               BLINK = BLNKRATE
               DETERMINE LIGHT TO BLINK
               TOGGLE LIGHT IN DISPLAY BUFFER
               DSPCHNG = TRUE
          ENDI
     ENDI

RETURN
$
```

The function of the **Hardware Monitor** Task is to perform those operations which support the device dependent hardware. Examples of the operations performed are:

1. Monitor the frequency actually output by the FG to increase its accuracy.

2. Determine when a measurement in the DMM is complete and update the display.

3. Monitor the regulation status of the PS and report changes (if enabled) over the GPIB interface.

In addition, the **Hardware Monitor** Task also performs device dependent hardware tests during power on initialization.

Pseudo code for the **Hardware Monitor** Task.

```
HWMONITR

     HARDWARE MONITOR INITIALIZATION

     POWER UP TESTS

     SUSPEND

     CLEAR INTERRUPT MASK

     REPEAT #FOREVER

          #DEVICE DEPENDENT FUNCTION

     ENDR
$
```

The Hardware Settings task performs the function of copying the **Pending Settings** to the **Current Settings** buffer and to the hardware registers as well as updating the front panel display. This function is activated by another task or interrupt handler in the system setting the **HWSETR** flag, which then remains set until the hardware has been updated.

The **HWSETR** flag is also used to lock out any changes to the **Pending Settings** during the Hardware Settings update processing. That is, if a routine needs to change **Pending Settings**, it must first check the HWSETR flag. If the flag is set, the task must suspend itself until the update is complete (the flag is cleared).

The **Hardware Settings** task also updates the front panel **ADDRESSED** and **REMOTE** indicators whenever necessary, and reports any floating point package errors by displaying the system error code on the front panel.

Pseudo code for the Hardware Settings task is:

```
HWSET

     HARDWARE SETTINGS TASK INITIALIZATION

     SUSPEND

     CLEAR INTERRUPT MASK
     REPEAT #FOREVER

          SUSPEND

          IF HWSETR THEN
               COPY PENDING SETTINGS TO HARDWARE
               COPY PENDING SETTINGS TO CURRENT SETTINGS

               EVENTCOD = NDDSTAT #REPORT UPDATED STATUS
               NEWEVENT #(POSSIBLE RQS CHANGE)

               HWSETR = FALSE
               BUILD DISPLAY BUFFER
                    (sets DSPCHNG if needed)
          ENDI
          REMOTE INDICATOR = REMBIT IN TMS 9914
          ADDRESSED INDICATOR = LADS OR TADS

          IF DSPCHNG AND NOT( DSPINHBT ) THEN
               DSPCHNG = FALSE
               UPDATE DISPLAY
          ENDI

          #REPORT SYSTEM ERRORS
          IF FPERR <> 0 AND SAV.SYSERR = 0 THEN
               SAV.SYSERR = FPERR
               FPERR = 0
               EVENTCOD = SYS.ERR
               NEWEVENT
```

```
            WRITE SYSTEM ERROR CODE (302) TO DISPLAYS
            DSPINHBT = TRUE
        ENDI

    ENDR
```

Note:

The **DSPINHBT** (display inhibit) flag is set by **IDPROC** or any other routine that writes directly to the displays and needs to control when it is updated.

5.1 Update Display

The Front Panel **Update Display** routine copies data from the **Display Buffer** in RAM to the display hardware. The Display Buffer is formatted to contain the information required for a direct copy to the display hardware.

A **Update Display** is initiated when the DSPCHNG flag is set and the **ID** key is not pressed. The display is not updated when the **ID** key is pressed because the **ID** key handler changes the display hardware directly and does not modify the **Display Buffer**. When the ID key is released, the display is restored by setting the DSPCHNG flag. Any changes to the **Display Buffer** which occurred while the ID key is depressed are then displayed automatically.

Any routine, including the interrupt handlers may change the display by setting up the **Display Buffer** with the new information and setting the **DSPCHNG** flag.

Note:

The **DSPCHNG** flag is reset before the display hardware is updated. This allows an interrupt handler to change the **Display Buffer** and set the **DSPCHNG** flag without it being missed because a **Update Display** had been initiated by some other routine. That is, if an interrupt handler set the **DSPCHNG** flag during the hardware update and it were cleared after the update, the new information may not get in the display because the **DSPCHNG** flag got cleared.

5.2 Display Buffer Update

There are two ways that the front panel display may be altered. The first way is by changing the **Display Buffer** and setting the DSPCHNG flag. On the next pass through the **Hardware Settings** task the **Display Buffer** is copied to the display hardware. The second way to change the display is to call a routine that builds all or part of the **Display Buffer** from the **Current Settings** and status information.

The **Display Buffer** is changed by the following:

1. The **Key Processor** when a numeric entry is in progress or when executing the **CLEAR ENTRY** function.

2. The **Key Processor** initialization, (caused by **LOCAL to REMOTE** transitions).

3. The **Hardware Monitor** task when a new result or status is to be displayed.

4. The **Hardware Settings** task when a setting change is made.

5. The **GPIB** interrupt handler when **REMOTE/LOCAL** or addressed state changes occur.

6. The **Key Processor** when error indicators must be flashed.

5.3 Display Buffer Builder

The **Display Buffer Builder** is used to construct the **Display Buffer** from the **Current Settings**. It consists of a series of calls to a set of partial buffer builders. Each partial buffer builder has its own inhibit flag which, when set, inhibits any change to that portion of the **Display Buffer** controlled by this partial builder. The reason for this structure is to allow the **Key Processor** to inhibit other tasks from changing the portion of the display that is in use during numeric entry or error display. Instruments with multiple displays have a **Display Buffer** build routine (and inhibit flag) for each display. The **Key Processor** initialization clears all inhibit flags and then builds the entire **Display Buffer**.

After the **Display Buffer Builder** builds the **Display Buffer**, it sets the **DSPCHNG** flag and then returns to the calling routine. The display hardware is changed in the next pass through the **Hardware Settings** task.

The **CLEAR ENTRY** function must reset the portion of the display being used for numeric entry, but must not clear the inhibit flag. The inhibit flag is cleared when a number is entered and accepted (no error). In the case of the Power Supply, which remains in the program mode after a number is entered, the inhibit flag is cleared when the **CLEAR ENTRY** key causes a transition out of program mode.

$

6.1 Driver Specification

This driver is a software interface between the TMS 9914 GPIB chip and the Operating System for Programmable Instruments (OSPI).

It uses approximately 1.5 K bytes of ROM (Coded in TESLA).

Requires approximately 40 bytes of RAM plus allocation for I/O Buffers. (minimum I/O buffer size is 16 bytes each)

It handles all REMOTE/LOCAL transitions -- Messages received in REMOTE are processed as if the instrument is in REMOTE upon asynchronous Remote Enable False transitions and GTL commands.

Message I/O features:

1. Always listens -- only asserts NRFD hold when the Input Buffer is full or when two messages are in the buffer and processing on the first is not complete. (This has the effect of increasing system throughput and makes the instrument more friendly to operate by allowing a new message to terminate the previous message output.)

2. Maintains a one-to-one correspondence between messages input and output messages generated.

3. Handles the generation of the Talked with nothing to say response.

4. I/O deadlock is detected and broken by terminating output.

5. Allows prescan and backup in the Input Buffer under system control to ease parsing messages.

Resets I/O buffers, status and message processing on Device Clear.

Handles both firmware and hardware controlled GET responses.

Status and Error reporting features:

1. Prioritized status reporting.

2. Synchronization of error query response to status bytes.

3. Supports reporting of error status through the error query with the instrument in the RQS OFF mode.

Written in Pseudo code and implemented in TESLA.

Handles both EOI only and <LF> character message terminators.

The GPIB Driver performs four major functions: Input, Output, Status Reporting, and handling of Interface messages and lines. The functional groups are partitioned into software modules which are listed below with a short description of what they do.

INPUT

BYTEIN An interrupt service routine which takes device dependent data bytes from the TMS 9914 chip and puts them into the **Input Buffer.**

GETBYTE A service routine called by the operating system to fetch the next available byte from the buffer.

FREESPACE This routine is called from the operating system to return bytes to the freespace area in the **Input Buffer.**

NEXTMSG Called by the **Message Processor** to start processing on the next message in the buffer.

CONTBIN Called when inputing binary data in the line feed mode to allow input to continue (it was terminated by a line feed character in the binary data).

INITINBUF This subroutine is used by the system during initialization and the processing of the DCL interface message. It sets up the pointers and flags associated with the **Input Buffer.**

OUTPUT

GPIB Task The GPIB Task is used to determine when there is data ready for output and generates the "talked with nothing to say response" when appropriate.

BYTEOUT The interrupt service routine that takes bytes from the **Output Buffer** and writes them into the TMS 9914 for transmission on the GPIB interface.

PUTBYTE Called by the command handlers or the system to put characters into the **Output Buffer.**

PUTEOI Called by the **Message Processor** to terminate the message being output. **PUTEOI** outputs the last byte of the message with the appropriate message terminator.

INITOUTBUF This subroutine is used to initialize the **Output Buffer** pointers and flags.

Status Reporting

NEWEVENT Called by the system or a command handler to report any change in status over the GPIB interface.

STORESTB	Transfers the highest priority status byte from the Pending status table into the TMS 9914 chip.
CHNGBUSY	Is called by the Message Processor to report any changes in the instruments "busy" status.
INITSTAT	This routine initializes the status reporting section of the system. It clears the Status Pending Buffer except for POWERON and Normal Device Dependent status entries.

Interrupt Handlers and Miscellaneous

GPIBDSP	This is the interrupt dispatcher which determines which GPIB related interrupts occurred and calls the appropriate service routines.
RTLPROC	The Return To Local Processor is called from the Key Processor to report the fact that a front panel button was pressed.
RLC	The Remote Local Change interrupt service routine handles the transitions from Local to Remote.
MA	The My Address interrupt handler recognizes the special case of receiving MTA (my talk address) without being untalked and resets the Output Buffer .
DCAS	The Device Clear Active State interrupt handler performs the instrument Device clear function.
HWGET.EN	This routine handles the special problems associated with enabling the Group Execute Trigger function which controls hardware directly with the output of the TMS 9914 chip.
GET.IH	The Group Execute Trigger interrupt handler performs the device dependent firmware function which is to be initiated by a GET.
INITGPIB	This routine is called during system initialization to set up the TMS 9914 chip and all of the data structures required to operate the GPIB interface in this system.

6.2 GPIB Variables

The following constants, pointers and flags are used in the definition and management of the **Input Buffer**:

*** Constant Pointers ***

INBUF This is the name used to define the address of the first physical byte in the **Input Buffer**.

INBEND The name which defines the physical end of the **Input Buffer**. It is assigned the address one after the last physical byte in the **Input Buffer**.

OUTBUF Points to the first physical byte in the **Output Buffer**

OUTBEND Points to one location after the last physical byte in the **Output Buffer**.

*** Variable Pointers ***

BIPTR Points to the first byte in free space, where the **BYTEIN** interrupt service routine stores the next byte. Initialized to **INBUF** value. BIPTR must never = 0.

GBPTR Defines the location from which **GETBYTE** fetches the next byte. Initialized to **INBUF** value.

BUPTR The Backup Pointer defines the location of the first logical byte in the **Input Buffer**, as well as the end of the freespace in the **Input Buffer**. BUPTR is initialized to **INBUF** value. The **Input Buffer** is full when BIPTR is incremented to = **BUPTR**.

EOMPTR Points to the byte following the last byte of the first message in the **Input Buffer**. If EOMPTR = 0 then the end of the first message is not in the **Input Buffer**.

EOMPTR2 Points to the byte following the last byte of the second message in the **Input Buffer**. If EOMPTR2 = 0 then the end of the second message is not in the **Input Buffer**.

BOPTR Points to the byte which is being output by the **BYTEOUT** interrupt service routine via TMS 9914 chip. **BOPTR** is incremented after the byte is placed in the TMS 9914 data out register. It is initialized to the physical beginning of the **Output Buffer**.

PBPTR Points to the location in the Output Buffer where **PUTBYTE** stores the next byte. It is initialized to the physical beginning of the **Output Buffer**.

*** Flags ***

INBUFUL	TRUE if the Input Buffer is full, FALSE otherwise. Set in the BYTEIN interrupt service routine and cleared by FREESPACE.
MSGREM	The Message Remote flag is used by the Message Processor to determine how it should process the message. If TRUE, the message is processed as though the instrument is in the Remote state. The MSGREM flag is set TRUE by BYTEIN if the first byte of the first message is received when the instrument is in REMOTE state. It is set FALSE if the first byte of the first message is received while the instrument is in LOCAL state or a transition from REMOTE to LOCAL occurred which was initiated by an rtl (front panel button push).
MSGREM2	This flag strores the REMOTE/LOCAL state of the instrument at the time the first byte of the second message is received. It is copied to MSGREM by the NEXTMSG routine.
MPBUSY	This flag is TRUE whenever the Message Processor is busy parsing a message. It is set by BYTEIN when the first byte of the first message is received. It is set FALSE in NEXTMSG when there is no message waiting for execution.
MVALID2	FALSE if the first byte of the second message has not been received. Copied to MPBUSY by NEXTMSG.
EOM	Set by GETBYTE routine when the End-of-Message condition is encountered. Cleared by GETBYTE otherwise.
BYTAVAIL	Indicates whether there are any bytes available in the first message. Is set TRUE by BYTEIN to indicate when GETBYTE can return characters from the buffer. It is set FALSE by GETBYTE when GBPTR is incremented and becomes equal to BIPTR.
DUMPOUT	This flag is used to indicate to the PUTBYTE routine when it should not put the bytes into the Output Buffer. It is set by the BYTEIN interrupt service routine when the first byte of the second message is received or when it detects that both the Input Buffer and the Output Buffer are full. DUMPOUT is tested and set in the PUTBYTE routine and cleared by NEXTMSG.
OBEMPTY	Is used to distinguish between buffer full and buffer empty. If set, the buffer is empty but there may be a hidden byte. Set TRUE by INITOUTBUF and when BYTEOUT empties the buffer. Set to FALSE when PUTBYTE stores bytes into the buffer.
SENDEOI	Set TRUE by PUTEOI to indicate to BYTEOUT that it may send the HIDDEN byte. Set FALSE by INITOUTBUF.

EOION Set by BYTEOUT to indicate that the EOI line is asserted. This is used by **BYTEOUT** to determine when to set **EOISENT**.

EOISENT This flag is set by the **BYTEOUT** interrupt service routine when the message terminator is accepted by the listening device(s). It is cleared by **INITOUTBUF** and the Talk Addressed interrupt service routine. **EOISENT** = TRUE prevents the GPIB Task from sending the talked with nothing to say message after EOI has been transmitted.

OUTBUSY Indicates whether the **Output Buffer** was used since it was initialized. It is set by the **PUTBYTE** routine when the first byte is placed in the buffer and is cleared when the instrument is talk addressed after **EOISENT** becomes true. or by **INITOUTBUF**. This prevents the GPIB **Task** from transmitting the "talked with nothing to say" response.

FLAGLOST Is set TRUE when the **BYTEOUT** interrupt flag was set and cleared but not serviced. **FLAGLOST** is cleared when a data byte is stored into the data out register in the **TMS 9914**. For further description of the problem, see the descussion of **BYTEOUT**.

*** Variable Storage ***

HIDDEN Saves bytes sent to **PUTBYTE** until it has been determined whether EOI should be asserted with this byte. After **PUTEOI** has set **SENDEOI** and **BYTEOUT** has emptied the buffer, **BYTEOUT** asserts EOI and sends the hidden byte.

INCHAR The variable in which **GETBYTE** returns the byte fetched from the **Input Buffer**.

OUTCHAR The variable used to pass the byte to be placed in the **Output Buffer**. The value must be stored in **OUTCHAR** before **PUTBYTE** is called.

GPINT The variable used to store the value of the TMS 9914 interrupt status register. Required because the reading the register clears the interrupt occurred flags.

STATQUO Status quo is a copy of the serial poll register of the **TMS 9914**, that is, the current status byte.

CRNEVENT Current Event is a copy of the **EVENTCOD** that was used to generate the **STATQUO**. It is set and used by the **STORESTB** routine.

EQRES Is a copy of the **EVENTCOD** for the most recently reported status byte in which the **RQS** message was asserted. It is used to generate the error query (ERR?) response.

C.RQS The current state of the RQS command setting. Set by the "RQS ON" command and cleared by "RQS OFF" It is used to disable service requests and force the reporting of

normal device dependent status.

EVENTCOD A temporary storage area (may be a register in the processor) used to pass event codes from one routine to another or within a routine.

PENDSTAT The **Status Pending Buffer** which is an array containing one byte of RAM for each of the priority classes identified in the **Status Table**. It saves one eventcode for each priority level and is referenced in **STORESTB** to determine which status byte to output.

NDDSTAT The entry in the **Pending Status Table** in which the Normal Device Dependent status is stored. It is given a separate name so that it is easier to reference, but it must be located at the end of the **Pending Status Table**.

6.3 GPIB Input

All input to the system from the GPIB interface is handled by the GPIB driver. The driver contains a BYTEIN interrupt service routine which puts bytes from the TMS 9914 GPIB chip into the Input Buffer whenever the chip generates a BI (Byte In) interrupt. The GETBYTE routine takes a byte out of the buffer and returns it to the calling routine. FREESPACE releases the Input Buffer data area which was scanned so that more bytes may be placed in the buffer. The NEXTMSG subroutine manages the message pointers and transfers message 2 to message 1. CONTBIN continues the input of binary data in the case of terminating prematurely on a line feed character embedded in the data stream while the instrument is in the line feed termination mode. All Input Buffer pointers and flags are initialized by the INITINBUF routine.

The Input Buffer is a circular buffer structure which may contain up to two complete messages. Further input to the instrument is prevented only when the Input Buffer becomes full or when two message terminators are marked in the buffer. The reason more than one message is allowed in the buffer is to satisfy the design goal of making the instruments always listen, which increases the system throughput and makes the instrument more friendly to operate. By limiting the number of messages to two, the overhead required to manage the messages is minimized and the benifits remain approximately the same. Pointers are used to mark the End-Of-Message position in the buffer so that the driver allows binary I/O.

Notes:

The flag MPBUSY was originally called MVALID. The reason for the change is that MPBUSY needed to have the same value, ie. MPBUSY must be set when the first byte of the first message is received in order to prevent the following situation:

> Message Processor is suspended waiting for the first byte of a message.
>
> A fast controller sends a query message and makes the instrument a talker.
>
> The GPIB Task is activated and finds the MPBUSY flag FALSE, so it starts the generation of output.
>
> The Message Processor is reactivated and also calls PUTBYTE, which is not re-entrant.

Setting MPBUSY when the first byte of the message is received solves the problem because the GPIB Task does not generate the "talked with nothing to say" response if the Message Processor is busy.

There are two RFD holdoff modes in the TMS 9914 chip. These modes may be set or cleared independently and they provide a holdoff after receiving EOI (EOI HOLDOFF) or a holdoff after every data byte (DATA HOLDOFF). Both holdoff modes are released by a release RFD holdoff command to the TMS 9914 chip.

Tests Performed on the Input Buffer

TEST	HOW DETERMINED
Is byte received the First byte of first message	MPBUSY = FALSE
Is the byte received the First byte of second message	EOMPTR = BIPTR
Instrument in line feed mode	LFMODE = TRUE
Physical end-of-buffer	PTR = INBEND
Input Buffer full	INBUFUL = TRUE
EOI received with byte	Status read from TMS 9914 chip
First Message terminator in buffer	EOMPTR <> 0
Input Buffer not full	INBUFUL = FALSE
Is the byte received the Last byte of first message	BIPTR = EOMPTR
End of first message	GBPTR = EOMPTR
Byte Available	BYTAVAIL = TRUE
Can bytes be returned to freespace	BUPTR <> GBPTR

Note to Message Processor, in scanning to End-of-Message (for format, error, etc.) the FREESPACE routine must be called between every byte. This prevents GETBYTE from suspending on a "buffer empty" condition when in reality the buffer is full, but bytes not returned to freespace.

BYTEIN

The BYTEIN interrupt service routine assumes that the TMS 9914 chip is initialized to the EOI HOLDOFF mode. If the LFMODE flag is TRUE, it additionally assumes that the chip is in the DATA HOLDOFF after every byte mode.

The function of the BYTEIN interrupt service routine is to:

1. Set the MPBUSY, MVALID2, MSGREM and MSGREM2 flags on the first byte of the first and second message.

2. Detect the deadlock condition of Input Buffer and Output Buffer being full and break the deadlock. If only the Input Buffer is full, holdoff on all data.

3. Input the data byte.

4. Indicate whether there is new data available.

5. Set the End-of-Message pointers.

6. Release the RFD holdoff if necessary.

The BYTEIN interrupt service routine is called (dispatched to) when the GPIB interrupt handler detects a BI interrupt from the TMS 9914.

The first thing the BYTEIN routine checks is whether this is the first byte of a message, which must be done before the byte is taken from the chip (while the holdoff is active). This is done by testing the MPBUSY flag, which is FALSE if there is no message being processed. Therefore, if it is FALSE, MPBUSY is set TRUE and the new status is reported, the Output Buffer is initialized, the GPIB Task is restarted, and MSGREM is set to the REMOTE/LOCAL state of the instrument.

The GPIB Task is restarted by the BYTEIN interrupt service routine when the first byte of the first message is received in order to prevent the GPIB Task from generating a response at the same time the Message Processor is. For example, if the instrument was a talker and the controller sent a DCL followed by the instrument's listen address and a message, then the instrument is talked with nothing to say during the time between the DCL and the first data byte of the message being received. Since this can be arbitrarily long, the GPIB Task may have started generating output and therefore must be reset along with the Output Buffer when the new message is received.

If MPBUSY is TRUE and BIPTR = EOMPTR, then this is the first byte of the second message, so MSGREM2 is set to the REMOTE/LOCAL state of the instrument, the MVALID2 flag is set TRUE, the Output Buffer is cleared (by calling INITOUTBUF) and the DUMPOUT flag is set true. Note that INITOUTBUF clears the DUMPOUT flag, so DUMPOUT must be set TRUE after initializing the buffer. This doesn't cause any problems because

interrupts are masked during the execution of BYTEIN.

The location into which the byte will be stored is saved and the buffer pointer is advanced. If it is at the end of the circular buffer then the pointer is set to the physical beginning of the buffer.

Next BYTEIN determines if the **Input Buffer** becomes full when the current byte is put in the buffer. If so, INBUFUL is set, **RFD holdoff** is set and the **Output Buffer** is tested to see if it is full. If full, then the **Output Buffer** is initialized and the DUMPOUT flag is set to break the deadlock of input and output buffers being full. Then the condition is reported by updating the GPIB status.

Once all pre-processing to determine if the **RFD holdoff** should be set is accomplished, the byte is read from the **TMS 9914** chip and stored in the **Input Buffer**. If this byte is part of the first message, then a new byte is being made accessable to the **GETBYTE** routines, so BYTAVAIL is set TRUE to indicate this. Then **BYTEIN** determines if an **End-of-Message** occurred with the last byte (either EOI received concurrent with the byte or the byte was a **LF** when the instrument was strapped for the LFMODE). If so, then the appropriate **End-of-Message** pointer is updated.

Finally, **BYTEIN** checks to see if it needs to release the **RFD holdoff** . This is done in **LFMODE** when the **Input Buffer** is not full and doesn't contain a second message terminator. When in **EOI** message termination mode, the **RFD holdoff** is released only if the byte stored was the last byte of the first message.

Note that releasing the holdoff when the TMS 9914 chip is not actively holding off data can cause the loss of a data byte.

Pseudo code for the **BYTEIN** interrupt service routine.

```
BYTEIN

#ROUTINES CALLED: CHNGBUSY, INITOUTBUF, NEWEVENT

#CALLED BY: GPIBDSP

        #SET THE MSGREM AND MSGREM2 FLAGS.
    IF NOT( MPBUSY ) THEN #FIRST BYTE OF FIRST MESSAGE
        MPBUSY = TRUE
        DISABLE HARDWARE TRIGGERS WHILE BUSY
        CHNGBUSY
        INITOUTBUF
        RESETGP #RESTART THE GPIB TASK
        MSGREM = REMOTE
    ELSE #CHECK FOR FIRST BYTE OF SECOND MESSAGE
        IF EOMPTR = BIPTR THEN #FIRST BYTE
            MVALID2 = TRUE
            MSGREM2 = REMOTE
            INITOUTBUF
            DUMPOUT = TRUE
        ENDI
    ENDI
    TEMPBI = BIPTR

        #ADVANCE POINTER (CIRCULAR BUFFER)
    BIPTR = BIPTR + 1
    IF BIPTR = ^ INBEND THEN #AT END OF BUFFER
        BIPTR = ^ INBUF (1)
    ENDI
        #CHECK FOR BUFFER FULL
    IF BIPTR = BUPTR THEN
        INBUFUL = TRUE
        TMSAUXCM = HDFA.SET #SET DATA HOLDOFF MODE
        IF NOT( OBEMPTY ) AND ( PBPTR = BOPTR ) THEN #OUTBUF FULL
            INITOUTBUF
            DUMPOUT = TRUE
            SAVE EVENTCOD ON STACK
            EVENTCOD = DEADLOCK #ERROR CODE = 203
            NEWEVENT
            RECOVER SAVED EVENTCOD
        ENDI
    ENDI

        #READ DATA BYTE FROM THE CHIP
    BYTE[ TEMPBI ] = TMSDATA

    IF EOMPTR = 0 THEN #PART OF FIRST MESSAGE
        BYTAVAIL = TRUE
    ENDI

        #SET END OF MESSAGE POINTERS
    IF EOIRECEIVED OR ( LFMODE AND BYTE[ TEMPBI ] = LF ) THEN
        IF EOMPTR = 0 THEN #END OF FIRST MESSAGE
            EOMPTR = BIPTR
```

```
          ELSE #END OF SECOND MESSAGE
               EOMPTR2 = BIPTR
          ENDI
     ENDI

          #DECIDE WHETHER TO RELEASE RFD HOLDOFF
     IF LFMODE THEN
          IF EOMPTR2 = 0 AND NOT( INBUFUL ) THEN
               TMSAUXCM = RHDF #RELEASE RFD HOLDOFF
          ENDI
     ELSE #EOI ONLY MODE
          IF BIPTR = EOMPTR AND NOT( INBUFUL ) THEN
          TMSAUXCM = RHDF #RELEASE RFD HOLDOFF
     ENDI

RETURN
```

GETBYTE

The GETBYTE subroutine is called by the system in the **Message Processor** to fetch the next byte from the **Input Buffer**. If the next byte of the message is not available, **GETBYTE** suspends until it is available. If it encounters the **End-of-Message**, then it returns with **EOM** set TRUE. If it detects that the buffer is full and there is no byte available it sets the **EOM** flag and error code #203. Otherwise it returns the byte in variable **INCHAR** after advancing the GBPTR and clearing the **EOM** flag.

Pseudo code for **GETBYTE**.

```
GETBYTE

#ROUTINES CALLED: LOADBYTE

#CALLED BY: SCANFRMT, GETCHR, LLSE.CMD AND
#     ANY OTHER ROUTINES THAT INPUT DATA FROM THE BUFFER.

     EOM = FALSE
     MASK INTERRUPTS
     IF BYTAVAIL THEN
          LOADBYTE
     ELSE
          IF GBPTR = EOMPTR THEN #END OF MESSAGE ENCOUNTERED.
               EOM = TRUE
          ELSE #NO BYTE AVAILABLE IN BUFFER
               IF INBUFUL THEN #ISSUE ERROR
                    MPERRCD = DEADLOCK
               ELSE #WAIT FOR NEXT BYTE
                    REPEAT
                         SUSPEND
                         UNTIL BYTAVAIL
                    ENDR
                    LOADBYTE
               ENDI
          ENDI
     ENDI
ENABLE INTERRRUPTS

RETURN
```

Note:

Interrupts must be masked during the execution of **GETBYTE** to prevent a controller from filling the **Input Buffer** between the time **BYTAVAIL** is tested and the test on **INBUFUL** is performed.

LOADBYTE

This subroutine loads a byte from the circular Input Buffer into the variable **INCHAR**. After getting the byte, it advances the **GBPTR** to the next available byte. It also indicates whether there are any more bytes available for the next call to **GETBYTE**.

```
LOADBYTE

#ROUTINES CALLED: NONE

#CALLED BY: GETBYTE

     INCHAR = BYTE[ GBPTR ]
     GBPTR = GBPTR +1
     IF GBPTR = ^ INBEND THEN #AT END OF CIRCULAR BUFFER
          GBPTR = ^ INBUF (1)
     ENDI

     #TEST TO SEE IF ANY MORE BYTES ARE AVAILABLE
     IF GBPTR = BIPTR OR GBPTR = EOMPTR THEN
          BYTAVAIL = FALSE
     ENDI

RETURN
```

Note that interrupts must be masked during the tests to determine if **BYTAVAIL** should be set FALSE because the **BYTEIN** interrupt service routine also tests and changes the state of these variables.

FREESPACE

FREESPACE is the routine called by the Message Processor to return bytes that are no longer needed to the free area in the circular Input Buffer. FREESPACE releases the bytes by setting BUPTR equal to GBPTR. If the Input Buffer is not full when FREESPACE is called, FREESPACE simply releases the space to the buffer. When the buffer is full the holdoff on all data is asserted and the instrument does not accept any more data. In this case FREESPACE must determine whether or not bytes are freed and if so enable more input by releasing the holdoff.

There are two conditions in which the holdoff must be released. One is when BUPTR <> GBPTR. That is, when the pointers are indicating that bytes are definitely released. The other condition is one in which the Input Buffer is full of the first message and it has all been read. This condition is detected when BUPTR = GBPTR and both BYTAVAIL and MVALID2 are FALSE. MVALID2 false indicates that the instrument never received any bytes in the second message and BYTAVAIL false qualifies the condition of having read all of the first message.

The following information may be helpfull in understanding the states of the Input Buffer flags when GBPTR = BUPTR.

States of the Input Buffer

1. Buffer Empty
2. Buffer Full and First message UNREAD
3. Buffer Full of Second message and First message READ
4. Buffer Full of only First message and all READ

State	1 INBUFUL 1	1 BYTAVAIL 1	1 MVALID2
1	1 FALSE 1	1 FALSE 1	1 FALSE
2	1 TRUE 1	1 TRUE 1	1 TRUE/FALSE
3	1 TRUE 1	1 FALSE 1	1 TRUE
4	1 TRUE 1	1 FALSE 1	1 FALSE

Pseudo code for **FREESPACE**.

```
FREESPACE

#ROUTINES CALLED: NONE

#CALLED BY: SCANFRMT, ABORTMSG, TABLSRCH, NEXTMSG

     #CHECK IF RELEASE OF RFD HOLDOFF IS REQUIRED
     IF INBUFUL THEN
          IF GBPTR <> BUPTR OR NOT( BYTAVAIL OR MVALID2 ) THEN
               #THESE ASSIGNMENTS MUST BE DONE BEFORE THE HOLDOFF
               #IS RELEASED BECAUSE THE HOLDOFF IS PREVENTING ANOTHER
               # BI INTERRUPT.
               INBUFUL = FALSE
               BUPTR = GBPTR
               IF EOMPTR2 = 0 THEN #SECOND MESSAGE NOT IN BUFFER, SO
                    IF NOT( LFMODE ) THEN #TURN OFF HOLDOFF MODE
                         TMSAUXCM = HDFA.CLR
                    ENDI
                    TMSAUXCM = RHDF.CLR #RELEASE RFD HOLDOFF
               ENDI
          ENDI
     ELSE #SIMPLY RELEASE SPACE TO BUFFER
          BUPTR = GBPTR
     ENDI

RETURN
```

NEXTMSG

The NEXTMSG subroutine manages the **Input Buffer** pointers and flags associated with having two messages in the **Input Buffer**. It is called by the **Message Processor** when it is ready to start processing a new message.

Pseudo code for **NEXTMSG**

```
NEXTMSG

#ROUTINES CALLED: FREESPACE, HWGET.EN, CHNGBUSY

#CALLED BY: MSGPROC

     FREESPACE
     DISABLE INTERRUPTS
     EOMPTR = EOMPTR2
     EOMPTR2 = 0
     MSGREM = MSGREM2
     #IF FIRST BYTE OF 2ND MESSAGE RECEIVED THEN
     #SET BYTAVAIL, MPBUSY TRUE, ELSE SET THEM FALSE.
          BYTAVAIL, MPBUSY = MVALID2
     HWGET.EN #CONDITIONALLY ENABLE HARDWARE TRIGGER
     CHNGBUSY
     MVALID2 = FALSE
     DUMPOUT = FALSE
     IF NOT( INBUFUL ) AND EOMPTR <> 0 THEN
          #WHEN EOMPTR = 0 THE MESSAGE TERMINATOR
          # HAS NOT BEEN RECEIVED.
          IF NOT( LFMODE ) THEN #TURN OFF HOLDOFF MODE
               TMSAUXCM = HDFA.CLR
          ENDI
          TMSAUXCM = RHDF.CLR #RELEASE RFD HOLDOFF
     ENDI
     ENABLE INTERRUPTS

RETURN
```

Note:

Interrupts must be disabled to prevent **BYTEIN** interrupts while the pointers and flags are being updated.

CONTBIN

This routine is used by command handlers that input binary data while the instrument is set to delimit on an ASCII character (like <LF>).

Note that commands which input binary data may not be in the same message as output commands since **BYTEIN** clears the **Output Buffer** when it receives the first byte after a terminator.

Pseudo code for the Continue Binary Input Routine.

```
CONTBIN

#ROUTINES CALLED: FREESPACE

#CALLED BY: BIN.PROC

     FREESPACE
     DISABLE INTERRUPTS
     EOMPTR = EOMPTR2
     EOMPTR2 = 0
     MVALID2 = FALSE
     IF NOT( INBUFUL ) AND ( EOMPTR <> 0 ) THEN
          TMSAUXCM = RHDF.CLR #RELEASE RFD HOLDOFF
     ENDI
     ENABLE INTERRUPTS

RETURN
```

Input Buffer Initialization

This subroutine is used to initialize the Input Buffer pointers and flags to their power-on state.

INTERRUPTS MUST BE MASKED BY THE CALLING ROUTINE.

Pseudo code for **INITINBUF.**

```
INITINBUF

#ROUTINES CALLED: CHNGBUSY

#CALLED BY: DCAS (DEVICE CLEAR)

     INBUFUL = FALSE
     BYTAVAIL = FALSE
     DUMPOUT = FALSE
     MPBUSY = FALSE
     MVALID2 = FALSE
     BIPTR = ^ INBUF (1)
     GBPTR = ^ INBUF (1)
     BUPTR = ^ INBUF (1)
     EOMPTR = 0
     EOMPTR2 = 0
     CHNGBUSY
     IF NOT( LFMODE ) THEN
          TAKE OUT OF HOLDOFF ON ALL DATA MODE
          #IN CASE THE BUFFER WAS FULL WHEN DCL OCCURRED.
     ENDI
     RELEASE RFD HOLDOFF

RETURN
```

6.4 GPIB Output

All messages output to the GPIB interface from the operating system are handled by the GPIB Driver. Because the Message Processor task doesn't know whether EOI is to be transmitted at the time it puts bytes into the Output Buffer, the PUTBYTE routine hides this byte from the BYTEOUT interrupt service routine and exposes the previously hidden byte. PUTEOI then exposes the "hidden byte" and in doing so indicates that the EOI is to be transmitted with the last byte.

The BYTEOUT interrupt service routine takes bytes from the Output Buffer and puts them into the TMS 9914 GPIB interface chip. The BO interrupts are always enabled. When a BO interrupt occurs and there is no data ready for output, the FLAGLOST variable is set TRUE to indicate that the interrupt occurred. When data becomes available, the interrupt is serviced by faking a BO interrupt (by calling the BYTEOUT interrupt service routine). In a normal software system the control for the BO interrupt would be achieved by disabling the interrupt while the Output Buffer was empty. In the TMS 9914 driver however, this technique does not work because the TMS 9914 chip clears the interrupt occurred bit when the interrupt status register is read (whether or not the interrupt was masked). Therefore, all interrupts must be serviced when the register is read and the interrupt lost information must be maintained by the firmware.

The Output Buffer is a circular buffer which at any time contains at most one message for output. The GPIB Output Buffer management routines include:

GPIB Task	The GPIB Task is used to determine when there is data ready for output and generates the "talked with nothing to say response" when appropriate.
BYTEOUT	an interrupt service routine which takes bytes from the Output Buffer and puts them into the TMS 9914 GPIB interface chip.
PUTBYTE	The subroutine which is called by the operating system and stores a byte in the Output Buffer.
PUTEOI	terminates the output message.
INITOUTBUF	initializes the Output Buffer pointers and flags.

GPIB Task

The GPIB Task is called in a round robin fashion with the other tasks in the system. It performs two major functions:

1. Determines when the instrument is "talked with nothing to say" and outputs the appropriate response. (For acquisition instruments this is the measurement -- other instruments send the hex byte "FF" along with the appropriate message terminator.

2. Fakes a **BYTEOUT** interrupt when the **BO** interrupt occurred flag is lost in the chip. See the **BYTEOUT** interrupt service routine for more information.

Pseudo code for the **GPIB Task:**

```
GPIBTASK

#ROUTINES CALLED: BYTEOUT

#CALLED BY: SYSTEM MONITOR

     SUSPEND

     CLEAR INTERRUPT MASK

     REPEAT #FOREVER

          SUSPEND

          IF (Talk Addressed) THEN
               IF NOT( OUTBUSY OR MPBUSY ) THEN
                    #THE OUTPUT BUFFER WAS NOT USED SINCE LAST TALKED
                    #AND MSGPROC IS NOT GENERATING ANY OUTPUT.
                    GENERATE "TALKED WITH NOTHING TO SAY" RESPONSE
               ENDI
               IF FLAGLOST AND (NOT( OBEMPTY ) OR SENDEOI ) THEN
                    #RESTART OUTPUT BY FAKING BYTEOUT INTERRUPT
                    MASK INTERRUPTS
                         BYTEOUT
                    ENABLE INTERRUPTS
               ENDI
          ENDI

     ENDR

RETURN
```

Note: Testing for **Message Processor** not busy insures that it is not in the process of generating output or dumping output.

Talked With Nothing To Say

The intent of this feature is to prevent a controller from getting hung in an input statement when it addresses an instrument as a talker without instructing it to do anything. In an attempt to make our acquisition instruments "easier to use", we send the measurement result in this condition. For example, with the DM or Counter it is nice to have a system in which the controller need only input the measurement result.

There is a conflict between the features described above in practical applications. The intent of sending the measurement result is to give an accurate value -- but suppose the controller has already read the result. Surely it is not interrested in getting an "old" result (one that it had read previously) but what should the instrument do if no new result is ready? In the case of the counters this could be a long time, particularly if the instrument is not being triggered. Clearly simply waiting for a new result would violate the intent of the "talked with nothing to say" response.

The solution for instruments that have this conflict is to:

1. Send the latest measurement if not previously transmitted.

2. If no result is ready, wait one "typical" measurement cycle (ie. in the DC approximately 1/3 to 1/2 second).

3. If still waiting for the result after the time has expired, send the FF

EOI.

BYTEOUT

The **BYTEOUT** interrupt service routine's function is to select the next byte to output and put it into the **TMS** 9914 Data Out Register.

If there are no bytes in the **Output Buffer**, **BYTEOUT** checks **SENDEOI** and **EOION** and **EOISENT** to determine if it should send the **HIDDEN** byte. Otherwise it disables the **BO** interrupt and if the previous bye was sent with EOI, it also sets the **EOISENT** flag.

The **BYTEOUT** interrupt service routine is dispatched to when the GPIB interrupt handler detects a **BO** interrupt. The **BO** interrupt is enabled by the **GPIB Task** when a byte is available for output.

The **BO** interrupt occured flag in the TMS 9914 chip is set if the source handshake sends a byte, or by MTA if there is no byte available for output in the data out register. The **BO** interrupt occurred flag is cleared by a read of the interrupt flag register, or by a write to the data out register.

In an early version of the TMS 9914 chip, it was possible for a controller to lose a byte in the transfer of data from the instrument due to the fact that the chip did not clear the **BO** interrupt occurred flag when data was written to the Data Out register. The following example illustrates the problem:

1. The controller sends **MTA** which sets the **BO** and **MA** flags (or perhaps a device dependent interrupt occurs).

2. The processor vectors to the interrupt dispatch routine due to the **MA** interrupt, but since reading the interrupt register clears the **BO** flag, and since there is no data yet generated, the processor sets a flag to indicate that a **BO** interrupt has come and gone, unserviced.

3. If the controller times out while waiting for the instrument to talk or decides to service an SRQ interrupt, or for any other reason sends an **UNT** and later **MTA**, the **BO** interrupt occurred flag is set again.

4. Eventually the processor has data ready to send and sees that the **BO** interrupt has come and gone unserviced, so it fakes a **BO** interrupt.

5. Note that there is a small window of time between the call to the **BYTEOUT** interrupt service routine and when the data byte is stored in the data out register. During this time, a **MTA** message from the controller sets the **BO** interrupt flag and it is not cleared by the store to the data out register. This could result in a second store to the data out register before the listener reads the first byte and thus a byte is lost.

*** The above sequence is no problem now because the new mask of the TMS 9914 chip clears the interrupt occurred flag when a data byte is stored into the Data Out register.

Description of BYTEOUT operation.

The **BYTEOUT** interrupt service routine begins by clearing the **FLAGLOST** variable in anticipation of storing a new byte into the chip.

If there are no bytes in the **Output Buffer**, **BYTEOUT** checks **SENDEOI** to determine if it should send the **HIDDEN** byte. Otherwise it tests **EOION** to determine if the previous byte was sent with EOI. If **EOION** is set, then it sets **EOISENT** TRUE to indicate that the transfer is complete. If it was not set, then it sets **FLAGLOST** to indicate that it did not store a byte in the chip and that no more **BO** interrupts will occur until an interrupt is faked by the **GPIB Task**.

If the **Output Buffer** is not empty, it stores the next byte into the chip, advances the pointer to the next available byte and if there are no more bytes available it sets the **OBEMPTY** flag TRUE.

Pseudo code for the **BYTEOUT** interrupt service routine.

```
BYTEOUT

#ROUTINES CALLED: NONE

#CALLED BY: GPIBTASK, GPIBDSP

     FLAGLOST = FALSE
     IF OBEMPTY THEN
          IF SENDEOI THEN #TRANSMIT THE LAST BYTE
               TMSAUXCM = FEOI #FORCE EOI WITH NEXT BYTE
               EOION = TRUE #SHOW EOI ASSERTED
               SENDEOI = FALSE
               TMSDATA = HIDDEN
          ELSE #WAIT FOR MORE OUTPUT
               IF EOION THEN
                    EOISENT = TRUE #SHOW EOI BYTE ACCEPTED
               ELSE
                    FLAGLOST = TRUE #INTERRUPT LOST (THANKS TI)
               ENDI
          ENDI
     ELSE #SEND A BYTE
          TMSDATA = BYTE [ BOPTR ]
          BOPTR = BOPTR + 1
          IF BOPTR = ^ OUTBEND THEN
               BOPTR = ^ OUTBUF (1)
          ENDI
          IF BOPTR = PBPTR THEN #BUFFER EMPTY
               OBEMPTY = TRUE
          ENDI
     ENDI

RETURN
```

PUTBYTE

This is the subroutine that is used to store a single byte into the **Output Buffer**. It is called by the **Message Processor** and the **GPIB Task** (when the GPIB Task determines that the instrument is talked with nothing to say).

Because the **Message Processor** doesn't know whether **EOI** is to be transmitted with bytes it puts into the **Output Buffer** until it scans the entire input message, the **PUTBYTE** routine cleverly hides one byte from the **BYTEOUT** interrupt service routine until **PUTEOI** is called.

In normal operation, PUTBYTE stores the hidden byte into the **Output Buffer**, transfers the byte passed in the variable OUTCHAR into the **HIDDEN** byte and returns. If the **Output Buffer** is full, then **PUTBYTE** checks to see if the **Input Buffer** is also full. If the **Input Buffer** is not full, then **PUTBYTE** suspends itself, waiting for available space in the **Output Buffer**. If both the **Input Buffer** and the **Output Buffer** are full, then **PUTBYTE** clears the **Output Buffer** sets the **DUMPOUT** flag to break the deadlock detected and also sets an error code to report the error condition to the controller.

Note: The **MPBUSY** flag prevents the **GPIB Task** from calling PUTBYTE when the **Message Processor** is using it, and the **GPIB Task** is restarted whenever **MPBUSY** is set. Therefore **PUTBYTE** does not need to be re-entrant, but interrupts must be masked to prevent **BYTEIN** from changing the **Input Buffer** variables (and **DUMPOUT**) during the execution of **PUTBYTE**.

Pseudo code for the **PUTBYTE** subroutine.

```
PUTBYTE

#ROUTINES CALLED: NEWEVENT, INITOUTBUF

#CALLED BY: OUTPUT ROUTINES IN MSGPROC AND GPIBTASK

     MASK INTERRUPTS #SO THAT BYTEIN CAN'T CHANGE VARIABLES

     IF PBPTR = BOPTR AND NOT( OBEMPTY OR DUMPOUT ) THEN #OUTBUF FULL
          IF INBUFUL THEN #BOTH BUFFERS FULL
               EVENTCOD = DEADLOCK
               NEWEVENT
               INITOUTBUF
               DUMPOUT = TRUE
          ELSE #ONLY OUPUT BUFFER FULL
               WHILE #OUTPUT BUFFER FULL
                    PBPTR = BOPTR AND NOT( OBEMPTY OR DUMPOUT ) DO
                         SUSPEND
               ENDW
          ENDI
     ENDI
```

```
     IF NOT( DUMPOUT ) THEN
          IF OUTBUSY THEN
               BYTE[ PBPTR ] = HIDDEN
               PBPTR = PBPTR + 1
               IF PBPTR = ^ OUTBEND THEN
                    PBPTR = ^ OUTBUF (1)
               ENDI
               OBEMPTY = FALSE
          ELSE #FIRST BYTE OF MESSAGE
               OUTBUSY = TRUE
          ENDI
          HIDDEN = OUTCHAR
     ENDI

     ENABLE INTERRUPTS

RETURN
```

PUTEOI

The **PUTEOI** routine is called by the **Message Processor** and the **GPIB Task** to terminate the message in the **Output Buffer**. Its function is to add the alternate message delimiter (if strapped for the **LF** mode) and set the **SENDEOI** flag to allow **BYTEOUT** to send the **HIDDEN** byte with EOI.

Pseudo code for the **PUTEOI** routine.

```
PUTEOI

#ROUTINES CALLED: PUTBYTE

#CALLED BY: MSGPROC, GPIBTASK

     IF OUTBUSY THEN
          IF LFMODE THEN
               OUTCHAR = CR
               PUTBYTE
               OUTCHAR = LF
               PUTBYTE
          ENDI

          #EXPOSE HIDDEN BYTE

          MASK INTERRUPTS
          SENDEOI = OUTBUSY
          ENABLE INTERRUPTS
     ENDI

RETURN
```

Note:

The interrupt masking is done to prevent the value of the **OUTBUSY** flag from changing while being copied to **SENDEOI**. If the processor can perform a memory-to-memory data transfer without being interrupted, then the interrupt mask is not necessary. The reason this must be protected is that a DCL interrupt changes the state of OUTBUSY and if the interrupt occurred during the assignment, **SENDEOI** could end up in an invalid state.

INITOUTBUF

The INITOUTBUF routine resets all of the Output Buffer pointers and flags to their power on state.

Pseudo code for the INITOUTBUF routine.

```
INITOUTBUF

#ROUTINES CALLED: NONE

#CALLED BY: DCAS, BYTEIN, PUTBYTE, MA

     OUTBUSY = FALSE
     EOISENT = FALSE
     SENDEOI = FALSE
     EOION = FALSE
     DUMPOUT = FALSE
     OBEMPTY = TRUE
     #FORCE TMS 9914 SOURCE HANDSHAKE IDLE WITH
     #NEW BYTE AVAILABLE FALSE
     TMSAUXCM = NBAF
     PBPTR = ^ OUTBUF (1)
     BOPTR = ^ OUTBUF (1)

RETURN
$
```

6.5 Status and Error Reporting

In general, Status and Error reporting are among the most difficult cummunications that need to be accomplished in a programmable instrument system. The reasons for this are:

1. Status/Errors that need to be reported typically occur asynchronous to the "normal" communication between an instrument and a controller.

2. As a result of #1 above, the number of events that can occur between transmissions is impossible to predict and therefore the management of these events within the instrument has a direct impact upon the information that it can report to the controller.

3. Within the definition of the IEEE-488 (GPIB) there is an interface function (SRQ) which provides the instrument with the capability to alert the controller that it needs "service". A method of reporting a minimmal amount of information (7 bits) concerning the reason for the ReQuest for Service (RQS) is also included in the standard as the Serial Poll response byte.

A set of protocols which govern how status and errors are reported have been established. These protocols are intended to provide a framework for the communication of status and errors through the use of the SRQ. Since the status byte returned in the Serial Poll response is difficult to decode and doesn't provide all of the information about the condition an alternate approach to the SRQ and Serial Poll was developed using a device dependent message -- the **ERROR?** command.

Use of Status Bytes

The **Codes and Formats** Standard specifies the coding for the status byte when used to report various commmonly occurring instrument conditions and also indicates codes which are available to a device designer for reporting device dependent status.

The following is a summary of the status bytes and the conditions they represent:

```
                                  ^------> Device Dependent
                                  1 ^----> RQS
                                  1 1 ^--> Error
                                  1 1 1 ^> Busy Bit
                                  1 1 1 1
SYSTEM EVENTS
     SRQ Query Request            0 1 0 X  0 0 0 0
     Power On                     0 1 0 X  0 0 0 1
     Operation Complete           0 1 0 X  0 0 1 0
     User Request                 0 1 0 X  0 0 1 1


SYSTEM ERRORS
     Command Error                0 1 1 X  0 0 0 1
     Execution Error              0 1 1 X  0 0 1 0
     Internal Error               0 1 1 X  0 0 1 1
     Execution Warning            0 1 1 X  0 1 0 1
     Internal Warning             0 1 1 X  0 1 1 0


DEVICE DEPENDENT EVENTS  (Examples)
     Channel A Overflow           1 1 0 X  0 0 0 1
     Channel B Overflow           1 1 0 X  0 0 1 0


DEVICE DEPENDENT STATUS  (Examples)
     Status Byte                  1 0 0 X  Y Z 0 0
        Where  Y = 1  if waiting for trigger
               Z = 1  if reading available
```

See the Codes and Formats Standard for descriptions of the errors.

The coding scheme provided by the **Codes and Formats** Standard implies that only one condition may be reported in any status byte. Since more than one condition may exist in the instrument at any given time, provision must be made for a means of determining which status is reported when a Serial Poll is performed. This specification is not included in the **Codes and Formats** Standard, therefore the following rules are used in the TM5000 programmable instruments to report status:

1. The first condition to occur which is to be reported with the **RQS** message asserted is written into the **TMS 9914** chip.

2. When a status byte is read by the controller, the status byte is updated with the highest priority status condition which had not been sent.

3. All status conditions which occurred but have not been reported (except power on) can be cleared by a device clear (**DCL**).

4. The Busy Bit represents the status of the **Message Processor**.

Use of ERROR? Command

Not all applications of programmable instruments using the GPIB need the capability provided by the SRQ function and the Serial Poll sequence. Very often the complexity of an interrupt service routine for the SRQ is more than the application demands. For this reason the RQS (Request for Service) command is implemented in all of the instruments to allow the controller to shut off the ability of the instrument to assert SRQ. In this mode of operation, another mechanism is provided to transmit the error conditions detected. The **ERROR?** command performs this function.

The scope of the error query was also expanded to provide information about some conditions which are not specifically errors, but are a status condition or event that the instrument needs to report. As a result of this broader definition for the response to the **ERROR?** command, the controller programmer has a simple way to acquire information about both error and status conditions.

In addition, the **ERROR?** command provides more information about the cause of the error or event than can be encoded in the Serial Poll Status Byte. For this reason, the error query can be useful in both the RQS ON and RQS OFF modes.

The **ERROR?** command always returns a single code which defines the event -- the codes are defined in the table below. In the RQS OFF mode, the **ERROR?** command returns the highest priority event waiting to be transmitted and clears the event from the pending event table. With RQS ON, the **ERROR?** command returns the code corresponding to the last event reported in the Serial Poll Status Byte.

In both modes, all pending events (except **POWERON**) can be cleared by a DCL or SDC.

Status Table

EVENTCODE	PRIORITY	STATUS BYTE	ERROR CODE	DESCRIPTION
Command Errors				
CMDHDR	2	$61	101	Command Header Error
HDRDLM	2	$61	102	Header Delimiter Error
CMDARG	2	$61	103	Command Argument Error
ARGDLM	2	$61	104	Argument Delimiter Error
NONNUM	2	$61	105	Non-numeric Arg. (numeric expected)
MISSARG	2	$61	106	Missing Argument
MSGDLM	2	$61	107	Invalid Message Unit Delimiter
CKSUMERR	2	$61	108	Checksum Error
BCNTERR	2	$61	109	Bytecount Error
Execution Errors				
REMONLY	3	$62	201	Command Not Executable in LOCAL
NSPLOST	3	$62	202	Settings lost due to rtl
DEADLOCK	3	$62	203	I/O Buffers Full, output dumped
SETCONFL	3	$62	204	Settings Conflicts
ARGRANGE	3	$62	205	Argument out of range
GETLOST	3	$62	206	Group Execute Trigger Ignored
CAL.MODE	3	$62	231	Not in Calibrate Mode
CANT.CAL	3	$62	232	Beyond Calibration Capability
SYM.FRQ	3	$62	251	Symmetry/Frequency Conflict
AMP.OFF	3	$62	252	Amplitude/Offset Conflict
AMP.AM	3	$62	253	Amplitude/AM Conflict
HLD.PHS	3	$62	254	Hold/Phase lock Conflict
HLD.FRQ	3	$62	255	Hold/Frequency Conflict
PHS.FM	3	$62	256	Phase lock/FM Conflict
PHS.VCF	3	$62	257	Phase lock/VCF Conflict
GATE.MOD	3	$62	258	Gate/Mode Conflict
Internal Errors				
INTFAULT	4	$63	301	Interrupt Fault
SYS.ERR	4	$63	302	System Error
MATH.ERR	4	$63	303	Math Pack Error
MEAS.ERR	4	$63	311	Timeout (measurement not completed)
OVRFL.ERR	4	$63	312	Measurement Overflow
SR.ERR	4	$63	313	Serial I/O Fault
MLSTRB.ERR	4	$63	314	Mag-latch relay strobe too long
STAT.ERR	4	$63	315	Phase lock Range Error

AFCRANGE	4	$63	316	Frequency correction range exceeded
FRONT.TO	4	$63	317	Front Panel Time out
BAD.CAL	4	$63	318	Bad Calibration Constant
	4	$63	320	
	4	$63	.	Device Dependent Errors
	4	$63	339	
*	*	*	340	System RAM Error
*	*	*	341	System RAM Error (low nibble)
*	*	*	342	
*	*	*	.	Reserved for Additional RAM Errors
*	*	*	349	
*	*	*	350	CPU RAM Error
*	*	*	351	Calibration RAM Checksum Error
*	*	*	360	0000 ROM Placement Error
*	*	*	361	1000 ROM Placement Error
*	*	*	362	2000 ROM Placement Error
*	*	*	363	3000 ROM Placement Error
*	*	*	364	4000 ROM Placement Error
*	*	*	365	5000 ROM Placement Error
*	*	*	366	6000 ROM Placement Error
*	*	*	367	7000 ROM Placement Error
*	*	*	368	8000 ROM Placement Error
*	*	*	369	9000 ROM Placement Error
*	*	*	370	A000 ROM Placement Error
*	*	*	371	B000 ROM Placement Error
*	*	*	372	C000 ROM Placement Error
*	*	*	373	D000 ROM Placement Error
*	*	*	374	E000 ROM Placement Error
*	*	*	375	F000 ROM Placement Error
*	*	*	380	0000 ROM Checksum Error
*	*	*	381	1000 ROM Checksum Error
*	*	*	382	2000 ROM Checksum Error
*	*	*	383	3000 ROM Checksum Error
*	*	*	384	4000 ROM Checksum Error
*	*	*	385	5000 ROM Checksum Error
*	*	*	386	6000 ROM Checksum Error
*	*	*	387	7000 ROM Checksum Error
*	*	*	388	8000 ROM Checksum Error
*	*	*	389	9000 ROM Checksum Error
*	*	*	390	A000 ROM Checksum Error
*	*	*	391	B000 ROM Checksum Error
*	*	*	392	C000 ROM Checksum Error
*	*	*	393	D000 ROM Checksum Error
*	*	*	394	E000 ROM Checksum Error
*	*	*	395	F000 ROM Checksum Error

System Events

POWERON	1	$41	401	Power On
OPCOM	#	$42	402	Operation Complete

IDREQ	#	$43	403	User Request

Execution Warning

*	*	*	521	Displayed During Signature Analysis

Internal Warning

OVERRANGE	#	$66	601	Overrange
CHAPROT	#	$66	602	Channel A Protect
CHBPROT	#	$66	603	Channel B Protect
NOPRESCL	#	$66	604	No Prescaler

Device Status

BELOWLIM	#	$C1	701	Below Limits
ABOVELIM	#	$C3	703	Above Limits
CHAOVF	#	$C1	711	Channel A Overflow
CHBOVF	#	$C2	712	Channel B Overflow
NEGVCHNG	#	$C5	721	Neg. Supply Change to Voltage Reg.
NEGICHNG	#	$C6	722	Neg. Supply Change to Current Reg.
NEGUCHNG	#	$C7	723	Neg. Supply Change to Unregulated
POSVCHNG	#	$D9	724	Pos. Supply Change to Voltage Reg.
POSICHNG	#	$DA	725	Pos. Supply Change to Current Reg.
POSUCHNG	#	$DB	726	Pos. Supply Change to Unregulated
LOGVCHNG	#	$DD	727	Log. Supply Change to Voltage Reg.
LOGICHNG	#	$DE	728	Log. Supply Change to Current Reg.
LOGUCHNG	#	$DF	729	Log. Supply Change to Unregulated
INLOC	#	$CC	731	In lock
OUTLOC	#	$C8	732	Not locked

* Not reported over GPIB so no entry in Status Table Required.
(Error Code displayed on Front Panel)

$ Hex Data

Priority dependent upon device requirements.

Note that the BUSY Bit is shown false in all of the status bytes in the table. The instruments, however, set it depending upon their state at the time the status bye is stored into the TMS 9914.

Problems imposed by TI 9914 Chip

Because of limitations imposed by the TMS 9914 chip design, any status byte written into the chip with the rsv bit set (the device's request for service message) cannot be changed until after it is read by the controller.

If the firmware does attempt to change this status, it is possible that the high priority status could be lost and the chip would report the low priority status twice. To demonstrate how this could occur, suppose:

1. An instrument puts some status with rsv = 1 into the chip.
2. The instrument has some new, higher priority status to report.
3. Just prior to storing the new status code in the chip, the controller starts a serial poll.
4. The controller reads the low priority status byte.
5. The processor finishes storing the byte in the chip.
6. The instrument firmware services the SPAS interrupt and has no way of knowing which status was read -- it must assume that the controller got the higher priority status. Therefore, it puts the low priority status (which it thinks the controller didn't get) into the chip.
7. The controller reads the low priority status a second time (with RQS asserted) and never gets the higher priority status.

As a result, any new higher priority status cannot be reported until the first status byte is read by the controller. However, the busy bit can be changed at any time since it does not affect the status condition (event) being reported. To summarize: the first occurance of a status condition for which the RQS message is asserted is reported. If other conditions occur before the status byte is read, they are held until the byte is read, and then the highest priority condition which is waiting to be reported is put into the chip.

Another problem in the TMS 9914 was discovered. The symptom of the problem is that events waiting to be reported with RQS asserted are lost when the controller performs part of the poll sequence very slowly or reads the status byte more than once. The reason this occurs is that the firmware in the instrument gets an SPAS interupt as soon as the status byte with RQS asserted is read. If the firmware writes a new status byte into the chip before the controller asserts ATN or if the controller reads the status byte more than once, a new SPAS interrupt occurs each time the status byte is read or ATN is asserted. Since the firmware cannot tell whether the controller has read the previous byte or the new byte, or is just toggling the ATN line, it

continues to put new status bytes into the master serial poll register in the chip. They are lost because the chip never transfers them to the slave register and thus they never get to the controller. There is no way to fix or even help this problem in the instrument firmware -- it must be fixed in the chip.

If the operator sends the instrument an RQS OFF command, the instrument only reports its normal device dependent status. This is due to the fact that there is no way to tell when a status byte with the **rsv** bit clear is read from the **TMS 9914** chip. When the instrument is in the RQS OFF mode, the status conditions which require the **SRQ** line asserted are saved and can be reported when the SRQ function is re-enabled if the events were not cleared by the device clear (**DCL**) message. When the instrument receives the RQS OFF command, it turns off the **SRQ** line by reporting normal device dependent status.

The instruments may also have other individual controls over whether a condition should be reported with **SRQ**. These may be turned ON or OFF at any time, however, the RQS OFF command has priority over the other ON conditions. That is, if the instrument is in the RQS OFF mode, no conditions with **SRQ** asserted are reported even though that conditon may be turned ON. One other constraint which applies to these functions is that if a status byte had been written into the chip to report a condition before the condition was disabled, the pending status is not modified, and only new occurances of the condition are not reported. This is because the **TMS 9914** chip has too many possibilities for problems when replacing a status byte with the **rsv** bit set.

Implementation Overview

For each condition which is reported in a status byte, there is an associated EVENTCOD. These codes are 8 bit integer values which start at 1 and indicate the priority of the condition, with 1 being highest priority. A **Status Table** in ROM is used to look up status bytes, status priority class and error query responses, using the EVENTCOD as an index. EVENTCODs which are greater than 128 are reserved for reporting normal device dependent status. There are no entries in the **Status Table** for these EVENTCODs. This is due to the fact that, for normal device dependent status, the EVENTCOD is equal to the status byte, and the priority class and error query response usually associated with the EVENTCOD are not required. An EVENTCOD equal to zero indicates that there is no status to report.

To report a change in status, an EVENTCOD is passed to the **NEWEVENT** routine, which enters that code into the **Status Pending Buffer.** If this event can be reported at this time, then **NEWEVENT** calls the **STORESTB** routine, which puts a status byte into the serial poll register of the TMS 9914 GPIB interface chip. Otherwise, the event is reported after the controller performs enough serial polls and it becomes the highest priority status not reported.

Each time the controller polls and receives a status byte with the RQS message asserted, **STORESTB** is called by the interrupt dispatcher to remove the EVENTCOD associated with that service request from the **Status Pending Buffer**, store the EVENTCOD for the error query response, and store a new status byte into the TMS 9914 chip.

NEWEVENT

The **NEWEVENT** routine is called from any routine that has status changes to report to the GPIB interface. The calling routine passes an EVENTCOD and **NEWEVENT** stores the **EVENTCOD** in the appropriate slot in the **Status Pending Buffer** If the current status in the TMS 9914 has the RQS message unasserted, NEWEVENT calls **STORESTB**.

To change the state of the **BUSYBIT** being reported, call the **NEWEVENT** routine with the current normal device dependent EVENTCOD which can be found in the **Status Pending Buffer**.

To report a change in one of the bits of the normal device dependent status, load the current normal device dependent EVENTCOD from the **Status Pending Buffer**, change the desired bit and call **NEWEVENT** with this new EVENTCOD.

Pseudo code for **NEWEVENT**.

```
NEWEVENT

#ROUTINES CALLED: STORESTB

#CALLED BY: MSGPROC, KEYPROC, OR ANY ROUTINE WITH STATUS TO REPORT

     SAVE AND MASK INTERRUPTS

     IF EVENTCOD >= 80H THEN #NORMAL DEVICE DEPENDENT STATUS
          #CLEAR RQS AND BUSY BIT
          NDDSTAT = EVENTCOD AND NOTRQSBUSY
     ELSE #EVENTCOD IS AN INDEX INTO PENDING STATUS
          IF EVENTCOD > NUMEVENTS THEN
               EVENTCOD = SYS.ERR
          ENDI
          PSPTR = ^ PENDSTAT ( PRIORITY ( EVENTCOD ) )
          IF BYTE[ PSPTR ] = 0 THEN #PRIORITY LEVEL IS EMPTY
               BYTE[ PSPTR ] = EVENTCOD
          ENDI
     ENDI
     IF ( STATQUO AND RQSBIT ) = 0 OR NOT( CRQS ) THEN
          STORESTB
     ENDI
     RESTORE INTERRUPT MASK

RETURN
```

STORESTB

The STORESTB routine finds the highest priority status in the **Status Pending Buffer**, looks up the associated status byte in the **Status Table** , adds the **Message Processor** Busy Bit (**MPBUSY**) and stores it into the **TMS** 9914 serial poll register. If the instrument is in the "RQS OFF" mode, the **STORESTB** routine reports normal device dependent status, regardless of other pending status conditions.

Called from **NEWEVENT** and **INITSTAT.**

Pseudo code for **STORESTB**

```
STORESTB

#ROUTINES CALLED: NONE

#CALLED BY: NEWEVENT, GPIBDSP

     SAVE AND MASK SPAS INTERRUPTS
     IF CRQS THEN #SRQ SHOULD BE REPORTED
          IF ( STATQUO AND RQSBIT ) <> 0 THEN
               #CURRENT STATUS BYTE HAS RQS ASSERTED
               #CLEAR RQSBIT IN CHIP AND INDICATE STATUS WAS REPORTED
               TMSSPOLL = BUSYBIT AND MPBUSY
               PENDSTAT ( PRIORITY ( CRNEVENT ) ) = 0
               EQRES = CRNEVENT
          ENDI

          #SEARCH TABLE FOR HIGHEST PRIORITY STATUS
          #NOTE THAT NDDSTAT <> 0 SO SEARCH DOESN'T FAIL
          PSPTR = ^ PENDSTAT (1)
          WHILE BYTE[ PSPTR ] = 0 DO
               PSPTR = PSPTR + 1
          ENDW
     ELSE #DON'T REPORT SRQ'S
          PSPTR = ^ NDDSTAT
     ENDI
     CRNEVENT = BYTE[ PSPTR ]
     IF CRNEVENT < 80H THEN
          #LOOK UP NEW STATUS BYTE IN Status Table
          STATQUO = STATTBL ( CRNEVENT )
     ELSE #NORMAL DEVICE DEPENDENT STATUS
          STATQUO = CRNEVENT
     ENDI
     TMSSPOLL = STATQUO OR ( MPBUSY AND BUSYBIT )
     RESTORE INTERRUPT MASK

RETURN
```

CHNGBUSY

The Change Busy routine is used to update the status of the BUSY bit reported in the GPIB Status Byte. The current state of the MPBUSY flag is stored into the TMS 9914 chip as the new status for the BUSY bit in the Status Byte.

Pseudo code for the CHNGBUSY routine.

```
CHNGBUSY

#ROUTINES CALLED: NONE

#CALLED BY: BYTEIN, INITINBUF, NEXTMSG

     SAVE THE INTERRUPT MASK
     MASK ALL INTERRUPTS
     TMSSPOLL = STATQUO OR ( MPBUSY AND BUSYBIT )
     RESTORE THE INTERRUPT MASK

RETURN
```

Note that the interrupt mask must be saved because this routine is called by interrupt handlers and active tasks.

INITSTAT

This routine initializes the status reporting section of the system. It clears the **Status Pending Buffer** except for power on and normal device dependent status.

It is called by the Device Clear interrupt service routine (**DCAS**).

```
INITSTAT

#ROUTINES CALLED: STORESTB

#CALLED BY: DCAS (DEVICE CLEAR)

     #CLEAR STATUS PENDING EXCEPT FOR POWER ON AND NDDSTAT.
     PSPTR = ^ PENDSTAT (2)
     REPEAT
          BYTE[ PSPTR ] = 0
          PSPTR = PSPTR + 1
          UNTIL PSPTR = ^ NDDSTAT
     ENDR

     IF CRNEVENT <> POWERON THEN
          STORESTB #SERVICE HANDLER TO PUT STATUS BYTE IN CHIP
     ENDI
     EQRES = 80H #NO NORMAL DEVICE STATUS

RETURN
```

Note that **NDDSTAT** is not initialized here. It could be set to 80H in this code, but it is probably more appropriate to set it to some device dependent status during **HWMONITR** initialization.

6.6 Interrupt Handlers and Miscellaneous

GPIB Interrupt Dispatch

The GPIB Interrupt Dispatcher reads the TMS 9914 interrupt register 0 and saves it, since reading the flags clears the interrupts. It then drops into a repeat loop which continues to service GPIB interrupts until there are no more pending.

The dispatch routine is optimized to provide optimal service time for BYTEIN interrupts. This is done on the assumption that the controller is almost always faster than the instruments and we can increase the system throughput most by getting the message into the buffer.

Note that all of the interrupts currently enabled in interrupt register 1 holdoff on NDAC. For this reason, once the condition causing the interrupt is determined, no other tests must be performed for the other conditions in register 1.

Pseudo code for GPIB dispatch.

```
GPIBDSP

#ROUTINES CALLED: BYTEIN, BYTEOUT, STORESTB, RLC, MA, DCAS, GET

#CALLED BY: IRQ.DSP

     GPINT = TMSINT0
     REPEAT #AS LONG AS IRQ IS ASSERTED BY TMS 9914
          IF ( GPINT AND BIBIT ) <> 0 THEN BYTEIN ENDI
          #BYTEIN TAKES CARE OF END INTERRUPTS AND MUST BE
          #CALLED BEFORE DCAS (TO RELEASE THE HOLDOFF)
          IF ( GPINT AND NOT.BI ) <> 0 THEN
               IF ( GPINT AND BOBIT ) <> 0 THEN BYTEOUT ENDI
               IF ( GPINT AND SPASBIT ) <> 0 THEN STORESTB ENDI
               IF ( GPINT AND RLCBIT ) <> 0 THEN RLC ENDI
          ENDI
          IF ( GPINT AND INT1BIT ) <> 0 THEN #LOOK IN REGISTER 1
               GPINT = TMSINT1
               IF ( GPINT AND MABIT ) <> 0 THEN MA
               ELSEIF ( GPINT AND DCASBIT ) <> 0 THEN DCAS
               ELSEIF ( GPINT AND GETBIT ) <> 0 THEN GET
               ENDI
               TMSAUXCM = DACR.CLR
          ENDI
          GPINT = TMSINT0
          UNTIL ( GPINT AND (INT0BIT + INT1BIT)) = 0
     ENDR

RETURN #TO IRQ HANDLER
```

REMOTE/LOCAL Processing

LOCAL to REMOTE

The instruments power up in the **LOCAL** state. A transition is made from the **LOCAL** to REMOTE state if the **Remote Enable** line of the GPIB interface is asserted when **MLA** (my listen address) is received. Whenever a transition from **LOCAL** to **REMOTE** occurs, the **Key Processor** Task is reset to its power on state. Therefore, any partially entered front panel commands are aborted and front panel buttons pressed at the time of the transition are ignored. Front panel entries are considered complete when the **Key Processor** Task.

On a transition from **LOCAL** to **REMOTE** state the **Key Processor** task is reset, possibly leaving the **Pending Settings** buffer in an invalid state. However, the **Pending Settings** are valid if they are a result of the **Message Processor's** normal sequence of command execution. Therefore, information must be provided to the **REMOTE/LOCAL** interrupt handler to indicate where the **Pending Settings** came from. A flag called **FPCNTRL** is used to indicate that the Front Panel user has control over the instrument settings. The **REMOTE/LOCAL** status cannot provide this function because the instrument may be in **LOCAL** state as a result of a **Remote Enable** false transition, in which case a succeeding transition from **LOCAL** to **REMOTE** should not cause a copy of **Current Settings** to **Pending Settings.**

The **FPCNTRL** flag is set by the **rtl** processor when any key (except **ID**) is pushed. **FPCNTRL** is reset by the **Message Processor** before any command that changes **Pending Settings** is processed. If **FPCNTRL** is set, then the **LOCAL** to **REMOTE** transition handler (**RLC**) calls the **COPYC2P** routine.

To make it easier for front panel operators to input multiple keystroke commands while a controller program is accessing instrument settings (with queries), the **rtl** which occurs when a user presses a front panel button is treated as a level which goes false only after a time out of approximately five seconds. This prevents the controller from taking the instrument to **REMOTE** state if a front panel operator is actively using the front panel. If the controller wants absolute control, then it can use the **LLO** command.

REMOTE to LOCAL

The REMOTE to LOCAL transition is initiated by either the **Remote Enable** line of the GPIB becoming unasserted, the controller issuing a GTL (go to local) interface message, or an operator pressing any of the front panel buttons except the **ID** button when the instrument is not in a **LOCKOUT** state. In each case the instrument immediately goes to its **LOCAL** state.

The **Message Processor** function is complicated by the asychronous occurance of **REMOTE/LOCAL** transitions in that some commmands (operational or setting commands) cannot be processed when the instrument is in **LOCAL** state, and it is possible (although unlikely) for several **REMOTE** to **LOCAL, LOCAL** to **REMOTE** cycles to occur during the processing of a single message. To simplify the situation for the operator and the **Message Processor** function, a variable is defined and associated with each message input. This variable is assigned the **REMOTE/LOCAL** state of the instrument at the time the first byte of the message is received. Since this variable is only modified by the **rtl** process, any subsequent transitions from **LOCAL** to **REMOTE** do not affect the processing of that message.

When an **rtl** occurs, (operator pressing a front panel button) the variable associated with each message in the **Input Buffer** (**MSGREM** and **MSGREM2**) is set to **LOCAL**. When the **Message Processor** encounters a **REMOTE ONLY** type command and **MSGREM** is set to **LOCAL**, it issues an execution error (number 201) to indicate that a command was processed which is not executable in **LOCAL** state.

Upon detecting either a **Remote Enable** false transition or a GTL interface message in the GPIB driver, the instrument state is set to **LOCAL**, but the variables associated with the messages in the input buffer are NOT changed. This is done to accomodate controllers like the 4051 which unassert the **Remote Enable** line when they are not busy. In this case the instrument processes the entire message as though it were in **REMOTE**. **Notes:**

1. A distinction between keys that change **Pending Settings** and keys that do not change **Pending Settings** cannot be made. The reason is that all keys (except ID) must force an **rtl** to occur and set the **Message Processor** status to **LOCAL**. Any **Pending Settings** set up before the **rtl** occurred must not be executed at the end of message processing because the instrument is now in **LOCAL** state, (under Front Panel control).

2. If any **Pending Settings** exist which have not been copied to **Current Settings** (indicated by **NSP** "New Settings Pending" flag) when an **rtl** occurs, then error #202 is issued and **Current Settings** are copied to **Pending Settings**. The **NSP** flag is set by any routine that alters the **Pending Settings**. It is cleared by: copying **Pending Settings** to **Current Settings** , copying **Current Settings** to **Pending Settings** , a **DCL**.

RTLPROC

This is pseudo code for the **rtl** processor, which performs the front panel return to **LOCAL** function. It handles transitions from **REMOTE** to **LOCAL** initiated by front panel button presses and is called from the **GETKEY** routine. This function is not included in the **RLC** interrupt service routine because the processor cannot tell **rtl** transition from the **GTL** or **REN** false, and we don't want to do this for these other transitions.

```
RTLPROC

#THIS ROUTINE PERFORMS THE FRONT PANEL RETURN TO LOCAL FUNCTION

#ROUTINES CALLED: NEWEVENT, COPYC2P

#CALLED BY: GETKEY (IN KEYPROC)

     MASK INTERRUPTS

     #FORCE AN RTL TO INHIBIT FURTHER TRANSITIONS TO REMOTE
     FORCE RTL AUX COMMAND WITH C/S = 1
     #NOW CHECK IF IT WENT TO LOCAL.  IF NOT, THEN INSTRUMENT IS
     #IN THE REMOTE WITH LOCKOUT STATE, SO THE FUNCTION
     #OF RTLPROC IS IGNORED.
     IF LOCAL THEN #MESSAGES ARE TO BE PROCESSED IN LOCAL
          MSGREM = FALSE
          MSGREM2 = FALSE
          IF NSP AND NOT( HWSETR) OR FPCNTRL THEN #SETTINGS FROM GPIB
          LOST
               EVENTCOD = NSPLOST #ERRORCODE = 202
               NEWEVENT
          ENDI
          COPYC2P
          FPCNTRL = TRUE
     ELSE #IN RWLS SO SET RTL PULSE
          TMSAUXCM = RTL.CLR
     ENDI
     UNMASK INTERRUPTS

RETURN
```

Note:

The routine **COPYC2P** "Copy **Current Settings** to **Pending Settings**" never suspends.

RLC Service Routine

The RLC interrupt is set by a LOCAL to REMOTE transition (REN AND MLA AND NOT rtl) or by a REMOTE to LOCAL transition (((NOT REN) OR GTL OR (rtl AND (NOT LLO))) while REMOTE). The MLA, LLO, GTL, rtl or REN FALSE messages do not cause a RLC interrupt unless they cause a transition from REMOTE to LOCAL or LOCAL to REMOTE. The REMOTE to LOCAL transition is not handled in the interrupt service routine because the processor cannot tell the cause of the transition. The rtl device message is handled by the GETKEY routine. GTL and REN FALSE are handled by the TMS 9914.

The operation of an instrument with the Group Execute Trigger directly (in hardware) controlling the device dependent function imposes an additional constraint upon the RLC routine. This is a result of the fact that the hardware trigger is disabled when the instrument is in the LOCAL state and is conditionally enabled on a transition to REMOTE state. There is a problem with this in that a controller may send MLA along with Remote Enable true, followed by a GET command. In this case, the TMS 9914 trigger pulse may occur before the instrument has enabled the hardware trigger. This will cause loss of synchronization with the other instruments.

To solve this problem, the My Address interrupt in the TMS 9914 must always be enabled. This causes an ACDS holdoff on MLA, preventing the controller from sending GET. Then the interrupt handler must service the RLC interrupt (if any) before releasing the ACDS holdoff on MLA. This allow the Processor to enable the hardware trigger before the controller sends GET.

Whenever a RLC interrupt occurs, the processor must check if the TMS 9914 chip is in the Remote With Lockout State (RWLS). If it is, then the rtl pulse command must be written into the TMS 9914 auxiliary command register. This prevents the instrument from getting locked in LOCAL state if the instrument is taken to REMOTE immediately after a front panel button push (which sets the rtl level mode). The reason it gets locked in the LOCAL state is that the transition to REMOTE resets the Key Processor, preventing it from timing out -- the usual way out of the rtl level mode, -- and any Remote enable false transition puts the instrument back into LOCAL with the rtl level active so MA and Remote Enable asserted does not take the chip to REMOTE.

Pseudo code for the RLC interrupt service routine.

```
RLC

#ROUTINES CALLED: COPYC2P, HWGET.EN, RESETKP

#CALLED BY: GPIBDSP

     IF RWLS THEN
          TMSAUXCM = RTL.CLR
     ENDI
     IF REMOTE THEN
          IF FPCNTRL THEN
               COPYC2P
               #MESSAGE PROCESSOR SETS FPCNTRL FALSE
          ENDI
          HWGET.EN #CONDITIONALLY ENABLE HARDWARE TRIGGER
          RESETKP #RESET KEY PROCESSOR
     ELSE #NOW IN LOCAL STATE
          DISABLE HARDWARE TRIGGER
     ENDI

RETURN
```

My Address Service Routine

The only function this service routine has to perform is to initialize the Output Buffer if the end of message terminator had been transmitted so that a new response will be generated.

Pseudo code for the MA Interrupt Service Routine

```
MA

#ROUTINES CALLED: INITOUTBUF

#CALLED BY: GPIBDSP

     #THIS ALLOWS A NEW MEASUREMENT TO BE SENT FOR EACH MTA RECEIVED

     IF TALKADDR AND EOISENT THEN
          INITOUTBUF
     ENDI

     TURN ON "ADDRESSED" LED

RETURN
```

Note:

The TMS 9914 chip doesn't generate a **BO** interrupt when talk addressed if the previous byte has not been sent.

Device Clear & Selected Device Clear

The purpose of the Device Clear (DCL) or Selective Device Clear (SDC) functions is to restart communication with the instrument. It does not cause the instrument to change any of its current settings, (these can be queried by the user when communication is re-established).

The TMS 9914 Chip is initialized to generate an interrupt on DCL and also to perform a **ACDS** holdoff on the DCL or SDC message. All device clear processing is performed before the holdoff is released.

When the **DCL** or **SDC** interface message is received, the following actions are taken:

1. All command processing is terminated. This is accomplished by restarting the **Message Processor**.

2. Clear the **Input Buffer** and the **Output Buffer**. Reset the **Message Processor** and the GPIB Task. Input is disabled by setting the **RFD holdoff** and output is terminated by disabling the BO interrupt in the **TMS 9914** and aborting the **Source Handshake** with **nbaf** (new byte available false) auxillary command.

3. Reset the instrument status (clear **SRQ** and the status byte which caused it to be asserted -- except **POWERON**).

4. If the front panel is idle (**FPCNTRL** = FALSE) then Call the **COPYC2P** routine to make sure that the **Pending Setting Buffer** and the hardware pending setting registers are in the proper state to begin processing a new message.

Notes:

Restarting the **Message Processor** and GPIB Task is accomplished by re-initializing their stacks.

Clearing the SRQ requires writing the new status byte without the **rsv** bit set into the TMS 9914 chip.

The DCL interrupt service routine must prevent the **BI** and **BO** interrupts which were pending from executing their service routines. This assures that the instrument doesn't transmit a spurious byte or think that it got a byte after the DCL occurred when the byte really arrived before the DCL.

$

Pseudo code for the DCAS interrupt service routine.

```
DCAS

#ROUTINES CALLED: RESETMP, RESETGP, INITINBUF, INITOUTBUF,
                    COPYC2P, HWGET.EN, INITSTAT

#CALLED BY: GPIBDSP, INITGPIB

     RESETMP #RESET MESSAGE PROCESSOR STACK

     RESETGP #RESET GPIB TASK
     INITINBUF
     INITOUTBUF
     #CLEARING OF SOURCE HANDSHAKE AND ACCEPTOR HANDSHAKE HAS ALREADY
     #BEEN DONE SINCE THE INTERRUPT DISPATCHER CALLS THE BO AND BI
     #SERVICE ROUTINES BEFORE DCAS.
     IF NOT( FPCNTRL ) THEN
          COPYC2P
     ENDI
     HWGET.EN #CONDITIONALLY ENABLE HARDWARE TRIGGER
     INITSTAT

RETURN
```

Device Trigger Function (GET)

Overview of GET Operation

The **Device Trigger** function is implemented in this operating system and the feature is enabled or disabled with the DT command. In the device trigger active state, the device dependent function is not executed until the Group Execute Trigger (**GET**) interface message is received. The DT OFF command is used to disable the Device Trigger function.

Since there may be more than one device function in an instrument which may be initiated by the **GET** message, the DT command accepts arguments which define the event or events to be initiated when the **GET** message is received. This approach also resolves problems in communicating to the instrument operator the fact that some events cannot be synchronized. This is a result of the fact that some of the events are initiated by firmware and others are initiated by the hardware signal generated by the TMS 9914 chip when a **GET** message is received.

There are two basic classifications of functions which are often controlled by the GET. These are **TRIGGER** events, which for example determine when a measurement is made, or **SETTINGS**, which specifies a non-standard mode of executing a previously defined group of settings.

The implementation of the GPIB driver routine to handle the **DT TRIGGER** mode is device dependent and the firmware performs the appropriate action to implement the trigger. In **DT TRIGGER** mode, the **GET** is ignored if the instrument is in the **LOCAL** state.

The GPIB driver responds to **GET** when in the **SETTINGS** mode by testing to see if the instrument is busy processing **REMOTE** message, in **LOCAL** state, or **NSP** is false. If any of these conditions is true, then the **GET** is ignored since it has not completed building the pending settings or in **LOCAL** control, in which case the operator does not expect the settings to change. If neither condition is true, the interrupt handler then simply sets the **HWSETR** flag and the execution of the settings is initiated upon the next pass through the Hardware Settings task.

Using the TMS 9914 GET Pulse

The hardware **GET** pulse from the TMS 9914 chip is enabled only if all of the following conditions are true:

1. The instrument is in the **REMOTE** state.
2. The **Message Processor** is not busy.
3. A hardware setting change is not in progress.
4. The device trigger mode is not "DT OFF"
5. The front panel user does not have control of the settings. This condition only applies in the DT SETTING mode.

The hardware trigger must be disabled when any of the following events occur.

1. The **BYTEIN** interrupt service routine sets **MPBUSY** TRUE.
2. The **rtl** processor forces a transition to LOCAL (and therefore sets **FPCNTRL** TRUE).
3. The **RLC** interrupt service routine detects a transition to **LOCAL** state.
4. The **GET** interrupt service routine sets **HWSETR** TRUE.

The hardware trigger is conditionally enabled when any of the following events occur.

1. The **RLC** interrupt service routine detects a transition to **REMOTE** state.
2. The **NEXTMSG** routine sets the **MPBUSY** flag FALSE.
3. The **Hardware Settings** task sets the **HWSETR** flag FALSE.
4. The **DCL** interrupt service routine sets the **MPBUSY** flag FALSE.

Hardware Considerations

In order to use the **GET** pulse provided by the TMS 9914, additional hardware must be included to provide the following features:

1. The ability to prevent the TMS 9914 **GET** pulse from triggering the instrument.
2. The ability to determine whether a **GET** pulse occurred before or after the trigger was enabled or disabled. That is, when a **GET** interrupt occurs, the micro-processor must determine if the instrument really received this trigger.
3. The ability to force a trigger from the micro-processor.

Interaction of GET with other tasks

A. REMOTE/LOCAL processor.

1. The hardware trigger is disabled in the **LOCAL** state and conditionally enabled on a transition to **REMOTE** state. When the instrument receives a **MLA** and **REN** (and goes to **REMOTE**) followed by a **GET**, the TMS 9914 trigger pulse may occur before the instrument has enabled the hardware trigger. This causes loss of synchronization with other instruments. To solve this problem, the **MA** interrupt in the TMS 9914 must always be enabled. This causes an **ACDS** holdoff on **MLA**, thus preventing the controller from sending the **GET**. Then the interrupt handler services the **REMOTE/LOCAL** change interrupt (if any) before releasing the **ACDS** holdoff on **MLA**. This allows the processor to enable the hardware trigger before the controller sends **GET**.

2. The **rtl** processor disables the hardware trigger to prevent triggers from occurring in **LOCAL** state.

3. The **RLC** interrupt service routine, upon detecting a transition to **REMOTE**, performs the call to **COPYC2P** before calling **HWGET.EN**. This insures that the **NSP** flag is cleared if the Key Processor had control of the settings buffers.

B. Hardware Settings task.

1. The new settings pending flag (**NSP**) is cleared before the **HWGET.EN** is called to enable the hardware trigger. This prevents the trigger from being enabled in the case where no **New Settings Pending** execution exist.

C. Message Processor

1. The hardware trigger enable routine (**HWGET.EN**) is called when the **End-of-Message** is processed. The **MPBUSY** flag must be cleared to enable the trigger, but the busy bit in the status byte must not be changed until the hardware trigger has been conditionally enabled. The busy status may be cleared upon return from the **HWGET.EN** routine.

2. The **Message Processor** must wait for **HWSETR** to go FALSE before calling **HWGET.EN**. This insures that the busy bit remains asserted until the hardware trigger has been enabled.

3. The device trigger mode can only be changed in the Message Processor while MPBUSY is TRUE. Device trigger mode must be set before **HWGET.EN** is called (this simplifies the Hardware Settings task).

D. **GET** Interrupt Service Routine

1. The **GET** interrupt must be enabled at all times. This is done for two reasons; 1) It becomes easier to determine if an **ACDS** holdoff must be released, and 2) a **GET** may occur before the hardware trigger is disabled, but not serviced until after the hardware trigger is disabled. (see section on hardware considerations).

E. DCL Interrupt Service Routine (**DCAS**).

1. The **DCL** Interrupt service routine must set **MPBUSY** FALSE, then call **HWGET.EN** and then clear the Busy bit in the status byte. This conditionally enables the hardware trigger before the controller sees the busy go false.

HWGET.EN

This routine is used to conditionally enable the hardware trigger to be pulsed by the TMS 9914 **GET** output.

Pseudo code for **HWGET.EN.**

NOTE:

This routine is for the general case of an instrument with a hardware trigger for everything. It is modified slightly for the PS5010 and the FG5010. The FG uses a mix of hardware and firmware triggers.

HWGET.EN

```
     SAVE THE INTERRUPT MASK
     MASK ALL INTERRUPTS

     #INTERRUPTS ARE MASKED TO PREVENT A TRANSITION TO LOCAL
     # OR DCL INTERRUPT BETWEEN THE TIME REMOTE AND MPBUSY
     # ARE TESTED AND THE TRIGGER IS ENABLED.

     IF DT <> OFF THEN
          IF REMOTE AND NOT ( MPBUSY OR HWSETR ) THEN
               IF DT <> SET OR ( NSP AND NOT ( FPCNTRL ) ) THEN
                    GET.EN = 1
               ENDI
          ENDI
     ENDI
     RESTORE INTERRUPT MASK

RETURN
```

Pseudo code for the FG5010 **HWGET.EN** routine

FGHWGET.EN

```
     SAVE INTERRUPT MASK
     MASK ALL INTERRUPTS
     IF REMOTE AND NOT ( MPBUSY OR HWSETR ) THEN

          CASE (SFLGS.CS AND DT.STAT) OF

               [ DT.TRIG ]
                    IF (MODE.CS = TRIG.MODE) OR (MODE.CS = BRS.MODE)
                    THEN
                         TRIG.CR = TRIG.CR OR GET.ENBL
                    ENDI

               [ DT.GATE ]
                    IF MODE.CS = GAT.MODE THEN
                         IF GAT.CS THEN
                              TRIG.CR = (TRIG.CR AND 0EFH) OR GET.ENBL
                         ELSE
```

```
                                TRIG.CR = TRIG.CR OR GATE.LVL LOR
                                GET.ENBL
                            ENDI
                        ENDI

            ENDC
        ENDI
        RESTORE INTERRUPT MASK

RETURN
```

GET Interrupt Service Routine

The **GET** Interrupt Service routine is implemented differently in each instrument, depending upon the individual hardware features. Examples of the code for the interrupt service routines for several applications follow.

For the DM:

```
DMGET.IH
     IF DT = TRIG AND REMOTE THEN
          GOTGET = TRUE #SET FLAG FOR THE HARDWARE MONITOR
     ENDI

RETURN
```

For the PS5010

The **GET** interrupt should be enabled at all times, so we know that the **ACDS** holdoff must be released unconditionally.

```
PSGET.IH

     IF ( REMOTE AND NSP AND CDT )
          AND NOT( MPBUSY OR HWSETR OR FPCNTRL ) THEN
          HWSETR = TRUE #THIS UPDATES CURRENT SETTINGS BUFFER
     ELSE
          SAVE EVENTCOD
          EVENTCOD = GETLOST
          RECOVER EVENTCOD
     ENDI

RETURN
```

For the FG5010:

The **GET** interrupt should be enabled at all times, so we know that the **ACDS** holdoff can be released unconditionally.

```
FGGET.IH

     IF NOT( HWSETR ) THEN #IGNORE GET'S WHILE UPDATING HARDWARE

          CASE (SFLGS.CS AND DT.STAT) OF

               [ DT.GATE ]
                    IF (MODE.CS = GAT.MODE) AND ( NSP = 0) THEN
                         IF (STROBE AND GAT.STAT)<>0 THEN
                              #THE CURRNET GATE IS HI
                              GATE.PS,GATE.CS=TRUE
                              #SET PENDING GATE LOW
                              TRIG.CR=TRIG.CR AND 0EFH
                         ELSE #THE CURRENT GATE IS LOW
                              GATE.PS,GATE.CS=FALSE
                              #SET PENDING GATE HIGH
                              TRIG.CR=TRIG.CR OR GATE.LVL
                         ENDI
                    ELSE GET.LOST #REPORT TRIGGER IGNORED
                    ENDI

               [ DT.SET ]
                    IF REMOTE AND ( NSP <> 0) AND
                         NOT( MPBUSY OR FPCNTRL ) THEN
                         HWSETR = TRUE #BEGIN SETTINGS EXECUTION NOW
                         TRIG.CR=TRIG.CR AND GET.DSBL #DISABLE HW
                         TRIGGERS
                         #WHILE THE SETTING CHANGE IS IN PROGRESS
                    ELSE GET.LOST
                    ENDI

               [ DT.TRIG ]
                    #PULSE THE TRIGGERED LIGHT OF A TRIGGER OCCURRED
                    IF (TRIG.CR AND GET.ENBL) <> 0 THEN PULS.LIT=1AH
                    ELSE GET.LOST
                    ENDI

                    [ ELSE ] #DT IS OFF
                         GET.LOST #REPORT TRIGGER IGNORED

          ENDC
     ELSE GET.LOST
     ENDI

RETURN
```

Interface Clear

The **Interface Clear** function (IFC) takes the instrument to an unaddressed state. No other action is taken in the instrument to reset any other status. The **IFC** is handled by the **TMS 9914.**

Hopefully this includes releasing the RFD holdoff and the new byte available false (see MAC or MA for recovery of a lost byte). Is it possible to loose bytes??

No code is necessary for the **Interface Clear** function, since the **ADDRESSED** light change is handled by the **Hardware Settings** task.

Parallel Poll

The following changes would be required to implement the Parallel Poll function for the TM500 instruments:

Hardware

PE must be tied low on the 75160 buffers to allow a wired "OR" capability during a Parallel Poll.

4051 A Parallel Poll routine in a rompack.

Instrument Firmware

The GPIB Interrupt Dispatch routine must recognize the PPC, PPE, PPD, and PPU interface messages by means of an unrecognized command interrupt from the TMS 9914 chip.

The PPC routine stores a "pass next secondary" and a "release ACDS holdoff" auxillary command into the TMS 9914 chip.

The PPE routine saves the interface message so that the processor knows which line to program when there is a reason to request service. The chip doesn't set up the Parallel Poll register, so the processor must use the PPE message and the CRNEVENT status to build a byte to store into the Parallel Poll register.

The PPU and PPD routines set a flag to disable reporting information via the Parallel Poll and clear the Parallel Poll register. The TMS 9914 chip does not do this automatically.

The status reporting routine must update the Parallel Poll register as necessary.

Note:

There is no automatic link in the TMS 9914 chip between the Serial Poll and the Parallel Poll.

A good use for the Parallel Poll function is to report Busy Status.

Conclusions:

Implementing the Parallel Poll function would slow down every GPIB interrupt since the dispatcher has to check for the Unrecognized Command interrupt.

The status reporting routines would have extra work if they must drive both the Serial Poll and the Parallel Poll functions.

Four extra interface messages (single bytes) must be decoded and handled by the processor.

GPIB Driver Initialization

Pseudo code for power on initialization of the GPIB Driver.

```
INITGPIB

#ROUTINES CALLED: DCAS

#CALLED BY: POWERUP

     TMSAUXCM = SWRST.SET #GIVE THE CHIP A HARDWARE RESET
     FLAGLOST = FALSE
     IF ( SWITCHES AND 1FH ) <> 31 THEN
          TMSADREG = SWITCHES AND 1FH
          TMSAUXCM = RTL.CLR #LOCAL STATE WITH PULSE MODE
          TMSAUXC = HDFE.SET #HOLDOFF ON EOI
          TMSPOLL , STATQUO = STATTBL ( POWERON )
          CRNEVENT, PENDSTAT (1) = POWERON
          DCAS

          IF LFMODE THEN #SET HOLDOFF ON EVERY BYTE
               TMSAUXCM = HDFA.SET
          ENDI

          #ENABLE RLC,SPAS, BI, BO, MA, DCAS, AND GET INTERRUPTS
          TMSMASK0 = BIBIT + BOBIT + SPASBIT + RLCBIT
          TMSMASK1 = MABIT + DCASBIT + GETBIT

          TMSAUXCM = SWRST.CLR #GIVE TMS9914 CHIP A SOFTWARE RESET
     ENDI

RETURN
$
```

There are two types of "firmware diagnostics" included in the TM500 instrument operating system:

1. Firmware which directly performs some diagnostic function.

2. Firmware features within the instrument which aid in the diagnosis of problems by performing a more general function which in itself is not a "diagnostic".

The following is a list of the diagnostic features. The implementation of items in the list is of course subject to the availability of adaquate ROM space to perform these functions, and each instrument implements as much as its space allows.

1. Firmware provision for extended diagnostics with "MISTY" box. This is done at power on by testing the contents of address 7FFE hex. If the result has the high order bit clear, the firmware jumps to the address specified in the location 7FFE (ie. it uses the vector provided by the ROM in the MISTY box - reserved address space is 4000 to 7FFF).

2. At power on, if the MISTY box is not attached, a RAM test is executed (for details of the test, see section titled RAM Test). If the test fails an error code is displayed on the front panel and the instrument will not operate.

3. Following the RAM test which is performed by executing code only from the ROM containing the power up vectors, the other ROMs in the instrument are checked for correct placement. If they are not installed correctly, an error code is displayed in the front panel and the instrument will not operate.

4. After the Placement tests all of the ROMs are checksummed, and again if an error is detected an error code is displayed and the instrument will not operate.

5. Next any tests that can be performed on the device dependent hardware are executed and the appropriate error codes are displayed if a problem is found. Whether the instrument will operate is dependent upon the nature of the failure. These tests may also be available by execution of the TEST command or initiated from the front panel.

6. Finally the settings of switches 7 and 8 of the GPIB address switch group are examined and the appropriate function (outlined below) is performed.

SWITCH		FUNCTION
7	8	
0	0	Normal Operation
0	1	Signature Analysis
1	0	Calibration Mode
1	1	* Extra test, calibration or excercise program

* This is an extra function which is dependent upon the instrument. For example, a special set of power on settings which place the PS5010 in a maximum power disapation mode so that the cycle rack tests are easier to perform.

The above decoding is recommended as a minimum provision. If additional capabilities are needed, they can be added by using the "*" switch position to activate a program which uses the front panel for input to select the test or operation to be performed.

7.1 Diagnostic Commands

The following "diagnostic" commands are included in all instruments in the system.

ADDRESS	<NR1>	The ADDRESS command sets the address for the BYTE command to operate upon. The argument must be a decimal value between 0 and 65535 decimal.
ADDRESS?		Returns the address to be used in the next BYTE command executed.
BYTE	<NR1>	Deposits the data specified by the argument into the address defined by the ADDRESS command. The argument must be a decimal number between 0 and 255. The address is automatically incremented by one after the data is stored.
BYTE?		Returns the data read from the location specified by the ADDRESS command. The number returned is a decimal number between 0 and 255. The address is automatically incremented by one after the data has been read.
JSR	<NR1>	The JSR command allows a way to start diagnostic routines (signature analysis, etc.) included in the instrument ROM from the GPIB interface. It performs a Jump to SubRoutine at the address specified by the argument. The argument must be a decimal number between 0 and 65535.

Note:

The ADDRESS command may be abbreviated to ADDR. The other commands must be entered with exactly the characters shown ("BYTE" and "JSR").

7.2 Diagnostic Task

Another function included in some of the instruments (those with enough free RAM space) is the ability to add a "diagnostic" task to the tasks currently executing in the instrument. In this way diagnostic routines can be written to monitor certain operations, etc. on an on-going basis and share the processor resource with the other tasks in the instrument.

This is done by downloading the code for the task and the definition of the new task stack, initializing a pointer to the task stack in the reserved area of the task stack table (ie. set BSTKTBL = ^STACKTBL(1)) and activating the task by modifying the top-of-task-stack-table variable (this is all done using the **ADDRESS** and **BYTE** commands.

Clearly the ability to do this requires an intimate knowledge of the instrument firmware and is not recommended that this be attemped by a customer.

8.1 Copy Current To Pending

This subroutine is called from the **Message Processor** during initialization and error processing, as well as from the **Key Processor** during **rtl** processing. The **GPIB REMOTE/LOCAL** interrupt service routine also calls the **Copy Current To Pending** subroutine when it makes a transition from **LOCAL** to **REMOTE** and the **FPCNTRL** flag is set.

Pseudo code for the Copy **Current Settings** to **Pending Settings** routine.

```
COPYC2P
     SAVE AND MASK INTERRUPTS
     IF NOT( HWSETR ) THEN
          MOVE Current Settings TO Pending Settings
          MOVE Current Settings TO HARDWARE REGISTERS
     NSP = FALSE
     ENDI
     RESTORE INTERRUPT MASK

RETURN
$
```

APPENDIX A

A.1 Control Structure

The counters do not have enough RAM to implement OSPI directly. Instead OSPI is emulated in the archtecture of the counters. OSPI uses a stack for each independent task, the SWI instruction to suspend tasks and the RTI to re-activate them. The operating system used in the counters (COS) maintains a single stack for the whole system and can suspend in only one task. All other tasks in the COS are polled -- in other words, a monitor loop calls each task in a predetermined order and starts execution from the same position in the task each time.

The task chosen to suspend is the **Message Processor** because it requires waiting for indefinite amounts of time for certain events to occur. Examples are: waiting for a byte from the GPIB interface, waiting for space in the output buffer, waiting for a measurement to complete, etc. By suspending in the **Message Processor** task we are able to make the COS look like OSPI when viewed by the user and therfore maintain compatibility across the TM500 programmable line. The only waiting allowed in other tasks is for events that will appear within a definite period of time or can be tested at the highest level in the module called by the monitor. For example, the relays have a maximum duty cycle and a wait loop is used to assure that the duty cycle is below the maximum. In this case, the waiting time is constant and SHORT enough to be unnoticed by the user.

The mechanism used to cause a suspend is the JSR instruction and the re-activation is accomplished with an RTS. The subroutine called SUSPEND pushes the X and Y registers on the stack and performs a JSR to the beginning of the monitor. The monitor polls the other tasks and as the last action performs an RTS. This returns control to the routine called SUSPEND, which restores registers X and Y and returns control to the point in the **Message Processor** just after the call to SUSPEND.

Notice that the **Message Processor** shares the same processor status as the rest of the system, unlike OSPI, which allows each task to have its own processor status. Care must be taken to assure that interrupts are not disabled when a SUSPEND is called because this could easily "hang" the instrument in a state waiting for **BYTEIN** interrupts, and the front panel buttons would not be active because interrupts are used to initiate the front panel button scans.

System Monitor Initialization

The COS has no **Stack Table** or **Active Task Pointer** since there is only one stack shared between the tasks. At power up the stack is initiated with the address of the beginning of the **Message Processor** so that the execution of the RTS at the end of the monitor loop transfers control to the **Message Processor** task.

Interrupt Handling

Most interrupts are serviced and control is returned to the point of the interrupt. There are two exceptions to this normal interrupt handling, however, for the **DCL** and the **GET** interrupts from the TMS 9914. These interrupts require certain actions to be taken that must be synchronized with the operating system.

The DCL interrupt is supposed to restart the **Message Processor**, but this can only be done by resetting the return address pushed on the stack by the SUSPEND subroutine to the beginning of the **Message Processor** task. Because the stack is shared by the other tasks, it cannot be modified until just prior to returning control to the **Message Processor**.

The **GET** needs to call routines used in the hardware request processor. These routines are not re-entrant and therefore the execution of them must be synchronized with the hardware request processor.

The synchronization for both the **DCL** and the **GET** is handled in the GPIB task. The **GPIB Task** is informed by flags that a DCL or **GET** occurred and it calls the appropriate routines to reset the **Message Processor** or to START, STOP, or RESET the measurement in progress. Once the function is performed, the GPIB task also releases the data acceptor holdoff which was automatically set by the TMS 9914.

A.2 Message Processor

COS sets the New Settings Pending flag at a higher level than done in OSPI. COS sets the New Settings Pending flag when the Message Processor determines that the command was of "settings" type. OSPI doesn't set the New Settings Pending flag until the new setting is about to be stored in the buffer. The main reason for this deviation from OSPI is to decrease the code size. The only side effect is that the flag gets set even if there is an error in the message and no new settings are put in the buffer.

COS processes arguments in a different fashion also. In COS the arguments are decoded in the Message Processor before the command handler is called, while OSPI calls the argument decoders from the command handlers (a more general approach to allow multiple arguements and binary block arguments). This saves code since the arguments are decoded in one place instead of in the many command handlers. To enable argument decoding by the Message Processor, a table of argument types required for each command is used. A limitation on the COS approach is that it does not handle multiple arguments, although this could be added by providing more information about the number and type of arguments to the argument processor.

Information is passed from the argument processor to the command handle using **ARGKEY** and **NBUF**. **ARGKEY** contains a zero if there was no argument or the argument table index of the character argument found. **NBUF** is three bytes and hold the numeric argument in a floating point format

The use of this method of communication is easy and allows the argument in the tables to be arranged so that the conversion of the argument to the hardware format required is simplified. For example, in the pending and current settings buffers, the ON and OFF status are stored with zero representing OFF and one representing ON. The argument table has the OFF argument first. Then **ARGKEY** contains a one when the argument is OFF and a two when the argument is ON, which only needs to be decremented to get the correct bit state. This implies of course that the argument tables cannot be re-arranged indiscriminately, but any change to the table must be offset by a change to the command handlers that expect to see that arguments key in **ARGKEY**.

Because the same GPIB board is used in the DC509P and DC5010 the **Message Processor** must determine if the command found in the command table is an executable command for that particular instrument. There are commands in the DC509P that are not in the DC5010 and visa versa. This is accomplished by maintaining two tables, one table contains the indexes of the DC509P commands that are not executable in the DC501 and the other table contains the indexes of the DC5010 commands that are not executable in the DC509P. When the table search finds a command in the command table, it looks to see which instrument type it is and looks for that index in the other instruments "only" command index table. If it is found in the other instruments table, **MPERRCD** is set to show an invalid command header.

ARGPROC

```
ARGPROC

      SAVEINDEX = INDEX
      IF ARGTYPTBL ( INDEX ) = 0 THEN
           ARGKEY = 0 #NO ARGUMENT
      ELSEIF ARGTYPTBL ( INDEX ) =0FFH THEN
           HDRDELIM
           IF MPERRCD = 0 THEN
                SCANFRMT
                IF EOM THEN #NO OPTIONAL NUMERIC ARGUMENTS
                     MPERRCD = MISSARG
                ELSE
                     GETCHR
                          NUMPROC
                ENDI
           ENDI
      ELSE
           IF INCHAR <> SPACE THEN
                ARGKEY = 0 #OPTIONAL CHARACTER ARGUMENT
           ELSE
                SCANFRMT
                IF EOM THEN #OPTIONAL CHARACTER ARGUMENT
                     TBLPTR = ARGTYPTBL ( INDEX ) #CHARACTER ARGUMENT
                     INDEX = 0
                     TBLSRCH
                     IF TYPE = 0 THEN #INVALID ARGUMENT
                          MPERRCD = CMDARG
                     ELSE
                          ARGKEY = INDEX
                     ENDI
                ENDI
           ENDI
      ENDI
      INDEX = SAVEINDEX

RETURN
```

A.3 REMOTE/LOCAL Considerations

Because the DC509P has front panel memory elements, ie. push buttons that are bi-stable push-push type, the transition from REMOTE to LOCAL is handled differently than in OSPI. A special provision is in COS to prevent a 4051 controller from taking the instrument to its front panel settings when it is not busy (the 4051 drops REN when not busy). Basically the counters do not go to front panel settings until a button is pushed.

The REMOTE led indicates the state that is controlling the settings and not necessarily the state of the instrument. For example, if the instrument is in REMOTE and receives a GTL command from the bus, it will go to the LOCAL state but the settings remain at their REMOTE states and the REMOTE led remains lighted. Over the GPIB interface the INIT command is used to set the instrument to the front panel state (note that INIT also resets system flags to their power on state).

At this point you may be wondering why a GTL interface command does not send the instrument (the DC509P) to its front panel settings since a GTL is a definite attempt by the user to got to local settings. The reason is simply because the TMS 9914 does not provide a way to distinguish between REN false and a GTL. Since we do not want to got to local settings on a REN false transition (due to the 4051 problem) we are forced to handle the GTL in the same manner.

A.4 GPIB Task

As mentioned previously, the GPIB Task is used as the synchronizing routine for the device clear and group execute trigger. It also performs the talked-with-nothing-to-say processing as described in OSPI.

OSPI, however, resets the **GPIBTASK** when the first byte of the first message is received. This prevents a part of the talked with nothing to say message from being generated if the first byte of the first message comes in while the instrument is outputing the talked with nothing to say message.

COS cannot reset the **GPIBTASK** because the task does not have a separate stack. Instead, COS sets a flag **DUMPNTS** (dump nothing to say) when the first byte of the first message comes in. The flag is tested in **PUTBYTE** to determine if the byte should be put in the buffer. The flag is unconditionally set false at the end of the **GPIBTASK** to prevent the flag from dumping a message other than the talked with nothing to say message.

$

APPENDIX B

B.1 Storage Format

The floating point format for TESLA's REAL*32 variables is similar to the single precision format for the proposed IEEE standard (see January 1980 COMPUTER magazine, pp. 68-79, published by IEEE). The most significant bit of the first byte of the floating point number is the sign of the mantissa, where "0" is positive and "1" is negative. The rest of the first byte is the 7 most significant bits of the exponent. The most significant bit of the second byte is the least significant bit of the exponent. The exponent is a power of 2 biased by 127 (for examples, see the abbreviated table of exponents).

The remaining 23 bits are the lower bits of the mantissa, where the most significant bit of the mantisa is assumed to be a "1" as the numbers are normalized. A binary point is implied between this implied most significant bit and the lower 23 bits of the mantissa, so that 1 <= mantissa < 2.

Examples of floating point numbers (in hex).

```
7F8XXXXX = Positive Infinity
FF8XXXXX = Negative Infinity
7F7FFFFF = 3.4 E+38 (largest value)
4BFFFFFF = 2^24 - 1 (largest exact integer, approximately
           1.678 E+7)
49742400 = 10^6
42C80000 = 100
41200000 = 10
40800000 = 4
40000000 = 2
3F800000 = 1
3F000000 = .5
3DCCCCCD = 0.1 (not exact)
00800000 = 2^-126 (smallest non-zero number, approximately
           1.18 E-38)
000XXXXX = +0
```

Exponent Table (Abbreviated)

2^X	Storage	FPACC
8	43 8	87
7	43 0	86
6	42 8	85
5	42 0	84
4	41 8	83
3	41 0	82
2	40 8	81
1	40 0	80
0	3F 8	7F
-1	3F 0	7E
-2	3E 8	7D
-3	3E 0	7C
-4	3D 8	7B
-5	3D 0	7A
-6	3C 8	79

B.2 Calling Sequence Examples

This section describes how to call the REAL*32 floating point functions from a TESLA program. The following is an example program which uses all of the functions.

```
PROCEDURE/EXAMPLE/


     EXT
          REALSTR #EXTERNAL PROCEDURE
          REAL*32:  REALINT, LOG2, REALVAL #FUNCTIONS THAT RETURN
                                                  # REAL*32 VALUES
     ENDE

     VAR
          #ALL REAL CONSTANTS MUST HAVE "."
          REAL*32: A,B,C/2., 4.2E+6, 60000./
          CHAR*8: STRING(15)/"1.2345"/
          BINARY*16: X

     ORIGIN 0BH
          #FORMAT FOR REALSTR
          BINARY*8: SIGDIG, EXPFIX, DECPNT, DIGITOUT, OPTIONS

     ORIGIN 1AH
          BINARY*8: FPERR #FLOATING POINT ERROR
     ENDV

     FPERR = 0
     A=A+B*C/(5.-A) #USES ADD,SUB,MUL,DIV,LOD,STO
     X= FIX (C) #CONVERT REAL*32 TO BINARY*16
     C= FLT (X) #CONVERT BINARY*16 TO REAL*32
     A= REALVAL (STRING(1)) #CONVERT ASCII STRING TO REAL*32
     A= REALINT (A+0.5) #ROUND TO NEAREST INTEGER
     IF ABS (A-5.) > 2. THEN #USES ABS AND CMP
          A=0.30103* LOG2 (A) #FIND COMMON LOG USING LOG2
     ENDI
     SIGDIG=4H #4 SIGNIFICANT PLACES
     OPTIONS=63H #ENGINEERING FORMAT
     REALSTR (A, STRING(1)) #CONVERT REAL*32 TO ASCII
     IF FPERR <> 0 THEN #CHECK FOR ERRORS
          .......
     ENDI

END.
```

B.3 OSPI vs TESLA Implementation of REAL*32

1. REALSTR for OSPI has formatted output. Standard TESLA assumes 6 digits of scientific output.

2. FIX and FLT assume unsigned binary for OSPI rather than two's complement integers for TESLA.

3. Error codes are different and are stored in FPERR in OSPI. Also, there is no need for ZZABORT.

4. REALVAL uses a circular buffer defined by INBUF and INBEND, and checks for end-of-message pointed to by EOMPTR. After REALVAL is called, ASCIIPTR (at address 14H) points to the character following the character that caused REALVAL to terminate. This information will be lost after calls to DIVIDE, LOG2 or REALSTR.

5. Code size has been reduced and speed enhanced by using direct memory (locations 0 - 1AH) for REAL*32's variables.

6. The object code for the OSPI version of REAL*32 must be appended to the users object code generated by the TESLA compiler before linking to the TESLA library.

7. Fuzzy compare says that numbers within 4 counts of each other are equal.

Source for the OSPI version of REAL*32 Math may be found on file:

DMMATH/UN=LOVECAH

and object code assembled for some fixed address between C000 and FFF8 is in file:

OBJMATH/UN=LOVECAH

A bug list with fixes for REAL*32 Math pack is available from:

Bob Bretl, ext 1118 or
Carl Hovey, ext 1104.

B.4 Structure Chart and Stack Depth

ROUTINE	STACK DEPTH	CALLS	CALLED FROM	ROM SIZE (BYTES)	APPROXIMATE MAX TIME (CYCLES)
ZZRSSABS	2		COMPILER	4	20
ZZRSSADD	6	LOD,ASR,NRM	COMPILER	219	1282
ZZRSSASR	2		ADD,FIX,INT,STR	25	777
ZZRSSCMP	4		STR	190	155
ZZRSSDIV	6	NRM	COMPILER	189	2404
ZZRSSFIX	4	ASR	COMPILER	55	614
ZZRSSFLT	4	NRM	COMPILER	25	690
REALINT	5	LOD,ASR,NRM	USER	40	1905
ZZRSSLOD	3		ADD,INT, COMPILER	40	82
LOG2	5	LOD,NRM	USER	349	11000
ZZRSSMUL	6	NRM	COMPILER	164	2160
ZZRSSNEG	3		COMPILER	13	38
ZZRSSNRM	2		ADD,DIV,FLT, INT,MUL	99	996
ZZRSSSTO	4		COMPILER	27	73
REALSTR	8	ASR,CMP,MUL, T10,LOD,NRM	USER	603	8714
ZZRSSSUB	8	ADD,NEG	COMPILER	9	1320
ZZRSST10	2		STR,VAL	62	118
ZZRSSTST	3		COMPILER	30	55
REALVAL	8	MUL,NRM,T10	USER	312	6469
MAX	8		TOTAL	2448	

B.5 VARIABLES FOR REAL*32 MATH

ABSOLUTE LOCATION	NAME	HOW USED
00-05	**FPACC**	6 bytes are used for the floating point accumulator (it is left in unpacked format to simplify programs).
00	**SIGN**	Most significant bit is "1" for negative, "0" for positive. Other bits change randomly.
01	**EXPONENT**	8 bits of exponent, offset by 127 FF = Infinity FE = 2^127 (approximately 1.7E+39) 80 = 2^1 = 2 7F = 2^0 = 1 7E = 2^-1 = 0.5 01 = 2^-126 (approximately 1.17E-38) 00 = ZERO (regardless of mantissa)
02,03,04	**MANTISSA**	24 bits of mantissa, implicit bit explicitly set and binary point assumed after most significant bit so that 1 <= mantissa < 2 when normalized.
05	**GUARD**	An 8 bit extention of the mantissa which is used for rounding and extra precision during intermediate operations. The most significant bit is the guard bit, the next bit is round bit, and the least significant bit is STICKY bit.
06,07,08	**REGISTER**	Used by ADD, CMP, DIV, LOG2, MUL, STR, VAL
09,0A	**POINTER**	Used by ADD, DIV, LOG2, MUL
0B	**SIGDIG**	Number of significant digits (see formatted REALSTR)
0C	**EXPFIX**	Exponent to fix to (see formatted REALSTR)
0D	**DECPNT**	Position of Decimal Point (see formatted REALSTR)
0E	**DIGOUT**	Number of columns of output for digits (see formatted REALSTR)
0F	**OPTIONS**	Packed flags for formatted REALSTR B0 - Delete "0" in exponent B1 - Delete "E+00" B2 - Delete trailing "." B3 - Right justify B4 - Delete Trailing "0"s B5 - Delete all trailing "0"s except ".0" B6,B7 0 0 — Scientific Notation 0 1 — Engineering Notation 1 0 — Floating Point 1 1 — Fixed Point

10	ZZSHARE	Ten bytes of shared memory, which may be used for temps by routines other than the mathpack. Use in the mathpack is as follows:
10		DIV, LOG2, STR, VAL
11		DIV, LOG2, VAL
12	QUOTIENT	DIV, LOG2, STR
13		DIV, STR
14	ASCIIPTR	DIV, LOG2, STR, VAL
15		DIV, LOG2, STR, VAL
16		LOG2, STR, VAL
17		LOG2, STR, VAL
18		LOG2, STR
19		LOG2, STR
1A	FPERR	Error Code indicating type of error and subroutine that detected the error. Subsequent errors will overwrite previous errors. The mathpack will never store a value 0 which is reserved for "NO ERROR". If used, this variable must be cleared by the routine calling the mathpack (see table of errorcodes).

The 6800 processor accumulators (A and B) are both clobbered by: ASR, FLT, NRM, FIX, STR, VAL, T10, LOG2.

The processor index register is clobbered by: INT, LOG2, STR, VAL.

The **FPACC** is modified by all routines except: LOD, STO, CMP, TST. It is clobbered by LOG2, STR, and VAL.

B.6 Formatting for REALSTR

A new version of REALSTR allows one to specify the format, number of significant digits, delete trailing zeros and right justify. There are 5 bytes used to specify the format: SIGDIG, EXPFIX, DECPNT, DIGITOUT, and OPTIONS.

SIGDIG, the number of significant digits to generate, must be specified for all formats. SIGDIG should be >= 1 and as a practical limit <= 7, since this is the limit of precision with the REAL*32 variables. If SIGDIG is not >= 1 and <= 127, then an error is generated and "1E+99" is output.

EXPFIX, the exponent to fix to. EXPFIX is a binary integer that specifies the power of 10 to appear in the output. For example, if the output should be expressed in milivolts then EXPFIX=$FD (-3) is used. Normally EXPFIX is equal to zero.

DECPNT, the position of the decimal point, must be specified for FIXED format only, however, the other formats automatically change DECPNT. DECPNT indicates the column of hte output after which the decimal point appears. For example, if DECPNT = 1 then the decimal point is after the first column (the sign appears in column 0). DECPNT must be >= 1 and <= SIGDIG, which means that the decimal point cannot appear before the first column.

DIGITOUT, the number of columns for digits, is used only with the right justify option, which may be used with any format. DIGITOUT must be >= SIGDIG and <= 127. Normally DIGITOUT = SIGDIG. Note that the sign, decimal point, and exponent are not included in the column count for DIGITOUT.

OPTIONS contains several fields defined as follows:

(00XX XXXX)	SCIENTIFIC	Has the decimal point after the first digit.
(01XX XXXX)	ENGINEERING	Has an exponent of an integer multiple of 3 with a decimal point after the first, second or third digit. Generally SIGDIG should be >= 3.
(10XX XXXX)	FLOATING POINT	Automatically places the decimal point. EXPFIX must be specified. SIGDIG must be large enough so that the decimal point may appear in the field of digits. For example, if the number to be output is one million and EXPFIX=3, then SIGDIG must be >= 3.
(11XX XXXX)	FIXED POINT	Allows one to specify the position of the decimal point and the value of exponent, making REALSTR shift the digits so that the decimal point appears in the desired column. This feature is used in the digital voltmeter as the position of the

	decimal point and indicates the range in use. Both DECPNT and EXPFIX must be specified. For example: for a 4.5 digit meter on the 200 milivolt range, SIGDIG=5, DECPNT=3, and EXPFIX=-3, giving "XXX.XXE-3".
(XX0X XXXX)	Output Trailing zeros, for example: "1.0000".
(XX11 XXXX)	Delete all trailing zeros, for example: "1.".
(XX10 XXXX)	Delete trailing zeros except for the one after the decimal point, for example: "1.0".
(XXXX 1XXX)	Right justify to DIGITOUT columns (not counting the columns for the sign and the decimal point) by inserting DIGITOUT - SIGDIG spaces after the sign.
(XXXX X1XX)	Delete trailing decimal point, for example: "1".
(XXXX XX1X)	Delete "E+00"
(XXXX XXX1)	Delete leading zero in exponent, for example: "E+1".

Format Examples for REALSTR

SIGDIG	EXPFIX	DECPNT	DIGITOUT	OPTIONS	OUTPUT RESULT
1 to 7	0	-	-	43H	Scientific (ala 4051)
1 to 7	0	-	-	00H	Scientific with all digits shown
1 to 7	0	-	-	63H	Engineering
1 to 7	0	-	>=SIGDIG	7FH	Engineering right justified
1 to 7	0	-	-	A3H	Floating point
1 to 7	0	=SIGDIG	-	F7H	Fixed point, integers
1 to 7	0	=SIGDIG-2	>=SIGDIG	CBH	Dollars and cents, right justified
5	-3	4	5	CBH	1 to 2000.0 milivolts
5	0	2	5	CBH	1 to 20.000 volts
5	+3	3	5	CBH	1 to 200.00 K ohms

B.7 Error Handling For REAL*32 Mathpack

When any of the REAL*32 floating point routines detects an error, it stores an errorcode into location **FPERR**, generates a default output, and returns to the calling routine. If more errors occur during subsequent calls to the mathpack, the first errorcode is overwritten and thus lost. **FPERR** = 0 is reserved to mean that no error has occurred. This must be cleared before calling the mathpack if it is to be useful upon return.

Note that there are no calls to **ZZABORT**, and also that the error codes are different than for the standard TESLA REAL*32 floating point routines.

Error codes are formed by adding a code to identify the routine issuing the error to another code which indicates the class of error. Only two classes of errors are reported: Invalid inputs and Overflow during processing. Underflow is converted to true zero but is not reported as an error.

Error Codes issued by REAL*32

FPERR	ISSUED BY	DEFAULT OUTPUT	DESCRIPTION OF ERROR
0	NONE		NO ERROR
1	ADD	+-INFINITY	INFINITY+X OR X+INFINITY
2	COMPARE	V (OVERFLOW BIT) SET IN CONDITION CODE	INPUT IS INFINITY
3	DIVIDE	+-INFINITY	INFINITY/X OR X/INFINITY
4	FIX	FFFFH	ABS(INPUT)>=2^16
5	LOG2	+-INFINITY	INPUT<=0 OR INPUT =+INFINITY
6	MULTIPLY	+-INFINITY	INFINITY*X OR X*INFINITY (BUT NOT 0*INFINITY)
7	REALSTR	"1E+99^@"	INPUT = INFINITY
8	TST	V (OVERFLOW BIT) SET IN CONDITION CODE	INPUT = INFINITY
9	REALVAL	+-INFINITY	NO DIGITS FOUND IN INPUT STRING. (REALVAL DOESN'T SCAN OVER ANYTHING BUT SPACES)
10	NORMALIZE	+-INFINITY	OVERFLOW WHILE ROUNDING
11	ADD	+-INFINITY	OVERFLOW
12	not used		
13	DIVIDE	+-INFINITY	OVERFLOW OR DIVIDE BY 0
14	not used		
15	not used		
16	MULTIPLY	+-INFINITY	OVERFLOW
17	REALSTR	"1E+99^@"	REQUESTED FORMAT IS INCOMPATABLE WITH FLOATING POINT VALUE (PROBABLY SIGDIG IS TO SMALL).
18	not used		
19	REALVAL	+-INFINITY	EXPONENT TOO LARGE OR TOO MANY DIGITS.

$

$