NATIONAL RADIO ASTRONOMY OBSERVATORY
GREEN BANK, WEST VIRGINIA

ELECTRONICS DIVISION INTERNAL REPORT NO. 273

PCI-3000 DATA ACQUISITION SOFTWARE MODIFICATIONS

BJORN B. STEVENS*

*SUMMER STUDENT

SEPTEMBER 1987

NUMBER OF COPIES: 150

PCI-3000 DATA ACQUISITION SOFTWARE MODIFICATIONS

Bjorn B. Stevens

TABLE OF CONTENTS

APPENDICIES

PCI-3000 DATA ACQUISITION SOFTWARE MODIFICATIONS

Bjorn B. Stevens

INTRODUCTION

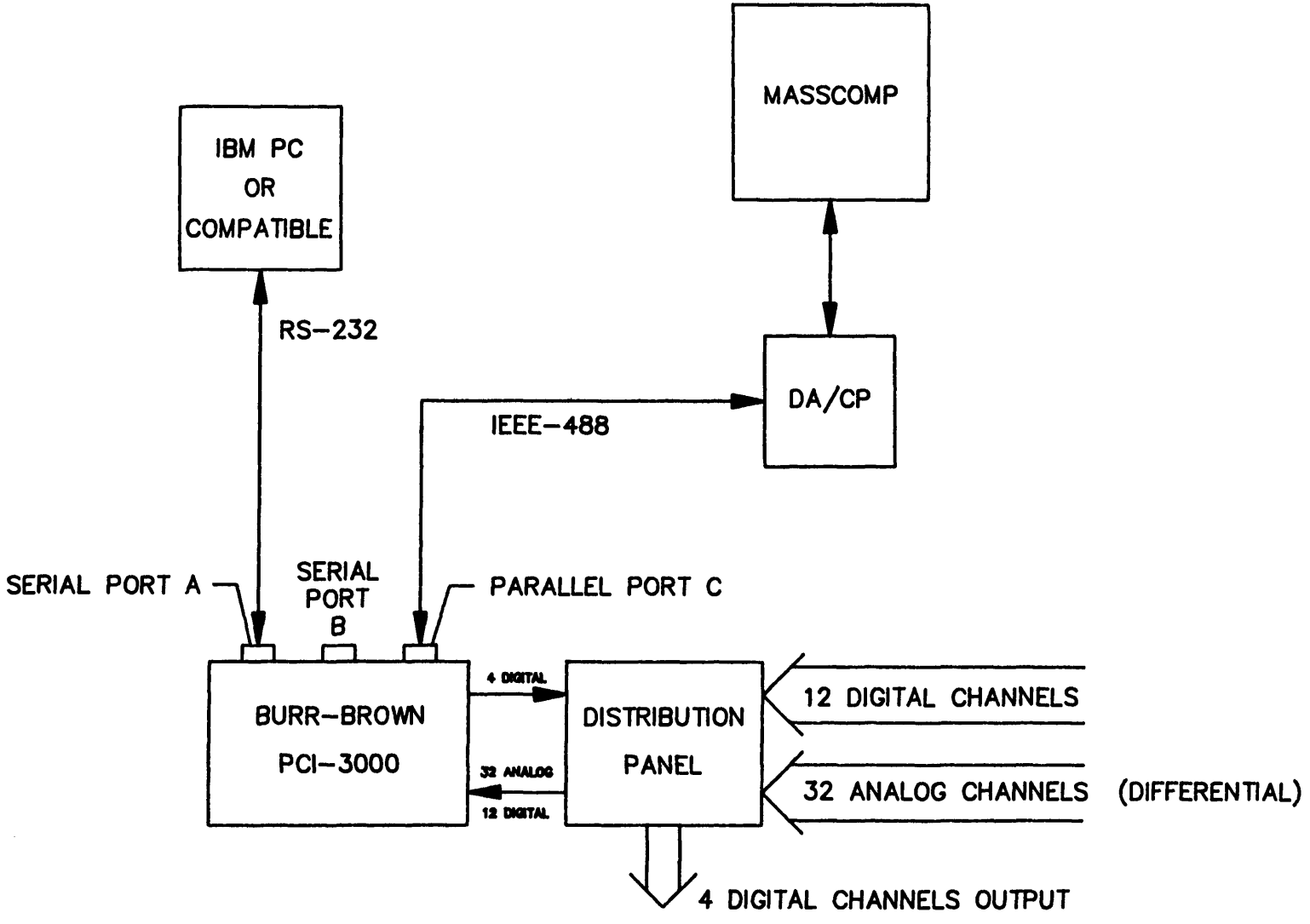
The PCI-3000 is a data acquisition device with its own Z80 processor and digital and analog interface boards. It is to be used at the 300-foot telescope as an auxiliary system to the MassComp telescope control computer for monitor and control of receiver frontends, control of signal routing, and monitor of telescope status and weather conditions. Communications between the PCI-3000 and the MassComp will be via the IEEE-488 bus.

I. PCI-3000 OPERATION:

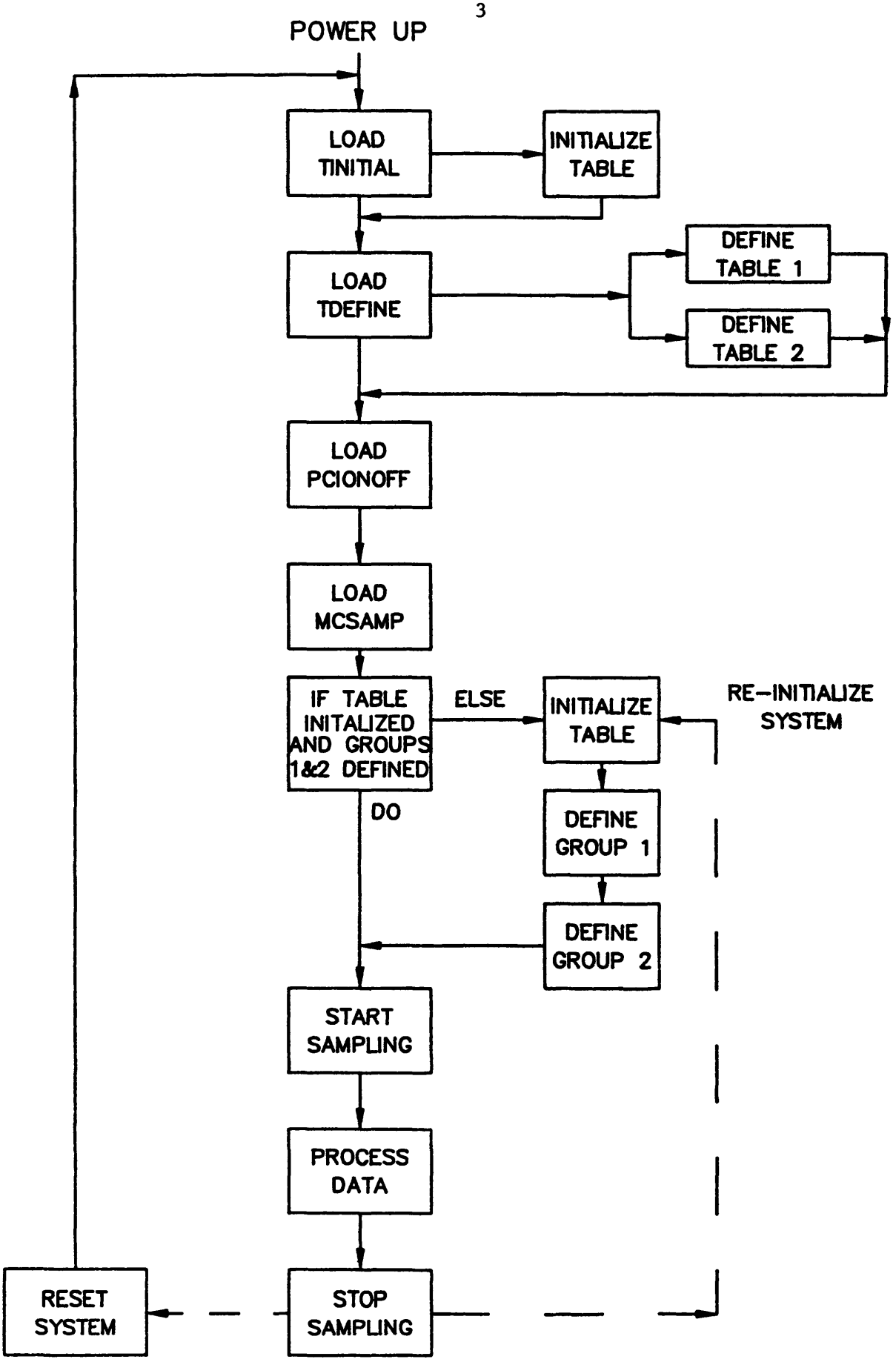
Here we will try to provide a brief outline as to the operation of the PCI-3000. The system is based on a Z80A microprocessor that runs on a 4 MHz clock. The heart of the system is the system firmware which acts much like an operating system. An excellent explanation of this system firmware is given in chapter 2 of the PCI technical reference manual. However we feel that there are several points that are important to the understanding of our manual, and these will be detailed here.

The way that the PCI keeps track of tasks that need to be done is through the use of its BSHED and BFLAG buffers. Every function that may be executed by the firmware has an associated flag in the BFLAG buffer. Corresponding to this flag number is the function starting address in the BSHED buffer. The system steps through the BFLAG buffer to see which function flags are set. When it finds a flag set it goes to the BSHED buffer to find the corresponding function starting address. Control is then transferred to this function address and the function is executed. After completion of a function its flag will be cleared in the BFLAG buffer and control will be returned to the scheduler. The system then continues this process as it steps its way to the end of the BFLAG buffer. Upon reaching the end of the BFLAG buffer the pointer is returned to the top and the process is continued. This could be called one pass through the scheduler.

An important function that is called during each pass through the scheduler is the watch dog timer. What this function does is makes sure that the system is not hung up. If it is a system warm reset will be performed. How this works is that a counter is decremented by interrupt every 16ms, if this counter is decremented to zero the system is reset. However the counter is loaded to an initial value such that it will take 0.75 seconds before the counter is decremented to zero. This counter is then reinitialized on every pass through the scheduler. Hence, only if a pass through the scheduler takes more than 0.75 seconds will the system be reset.



AUXILIARY DATA ACQUISITION SYSTEM (AUXDAS)



AUXDAS FLOW CHART

A warm reset does not seem to affect user RAM. It appears to only reinitialize the system RAM to its power up values. Hence, it will be necessary to reload our functions to the system RAM. Even though the code in user RAM is intact, the calling sequences and flag initialization in system RAM will have been wiped out. For user functions only those which will be called externally must be reinitialized (our first three functions). To do this it is only necessary to use the Load Function command to load the start addresses of each function. However, using our C program we find it easier just to reload the whole system.

II. PROTOCOL DEFINITIONS:

In our system we are primarily concerned with two types of protocol (ascii & binary) over two types of interfaces (the RS232 & the IEEE-488).

The ascii protocol consists of the '!' as a header byte followed by the microprocessor board id, which is currently configured to a one. Next come the data words which are separated by commas. The whole sequence is terminated by a carriage return.

The binary protocol consists of hexadecimal bytes. These bytes can be seperated into three parts, the preamble, the data bytes, and the checksum. The message is terminated by the data byte expiration as defined in the preamble by the 'number of bytes to follow' byte.

Both of these formats can be implemented on either interface. The RS232 interface is generally quite crude with no handshaking and serial transmission at 1200 baud. For input output interactions with the PCI the PCI.BAS program works best, especially for communications in ascii protocol. This program is capable of a 9600 baud transmission rate. However, the PCI and the basic code must be reconfigured for this to happen. The general purpose troubleshooting C program is equally helpful. It is most useful in binary protocol transmissions over the RS-232 at 1200 baud.

The real operational communications will take place over the IEEE-488 parallel port. These communications will be much more sophisticated as the PCI supports the following IEEE-488 protocol capability.

AH1	Complete acceptor handshake capability
SH1	Complete source handshake capability
T5	Basic talker, serial poll, talk only, unaddress if MLA
TE0	No extended talker
L3	Basic listener, listen only mode, unaddress if MTA
LE0	No extended listener
SR1	Complete service request capability

In ascii protocol the string termination character is a carriage return sent concurrently with an EOI signal. In binary protocol there is no termination character, just the EOI signal sent with the last byte.

An important note about the IEEE-488 bus is that the bus address for the pci on the IEEE-488 bus system is the same as the microprocessor board id. Hence if the system must be reconfigured to exist on a bus where address one is taken the system must be given a different address. This is done by reconfiguering switches 4 - 8 on SW1. See the PCI-3000 user manual page 3-2. Note this will also change the response and calling messages since the second byte is the microprocessor board address.

For more interface information see chapter 3 in the PCI-3000 manual. For more protocol information see appendix B and chapter 3 in the PCI-3000 technical manual.

III. PCI-3000 ASSEMBLY LANGUAGE FUNCTIONS:

To minimize the load on the controlling MASSCOMP computer several Z80 assembly language functions were designed to be implemented in the PCI-3000 system. As the PCI was configured on purchase it could only sample one input at a time, or several analog functions periodically. Each time a function was to be sampled it was necessary to send a controlling command to the PCI to perform the necessary task.

Hence, it was decided to take full advantage of the PCI-3000 Z80 microprocessor and design a set of functions that could sample two groups of inputs at separate sampling periods. This sampling was to be controlled intrinsically. The only time the MASSCOMP would become involved would be when the PCI had assembled a packet of sampled data and was ready to send it across the IEEE-488 bus. With this in mind the following four functions were written to perform the listed tasks:

TINITIAL.ASM:
Initializes an area of system RAM which will be used as a table, and a variable storage area.

TDEFINE.ASM:
Defines the membership of group N (N = 1,2). Defines the sampling period of group N (N = 1,2). These definitions will take place in what will be now known as the sample table.

PCIONOFF.ASM:
Set up the system firmware to either call MCSAMP periodically, or terminate the

periodic call of MCSAMP. Where the periodicity at which MCSAMP is called is determined by the group 1 sampling period.

MCSAMP.ASM:

Sample group N as determined by it's flag status in the sample table. N will always correspond to 1 except on every Xth call N will correspond to 2. X is the number entered to define the group 2 sampling period.

The foundation which governs the successful operation of this group of functions is the sample table. This table which is illustrated on the next page consists of 60H (96) bytes of RAM spanning the addresses 9000H through 905FH. The division of this table is documented below:

Location (Hex)	Purpose
9000 & 9001	Group 1 sampling period.
9002 & 9003	Group 2 sampling period only low byte used.
9004 to 9023	Table entries corresponding to 32 analog channels. Each entry contains sample flag status as well as analog gain information.
9024 to 902B	Table entries corresponding to 1st group of 8 digital channels. Each entry contains sample flag status.
902C to 902F	Table entries corresponding to 2nd group of 4 digital channels. Each entry contains sample flag status.
9030 to 904F	Additional space for table expansion.
9050 to 905F	Reserved for use by variables declared in functions.

A. TINITIAL.ASM MANUAL:

TINITIAL (table initialize) is a Z80 assembly language program that is designed to initialize the sample table that is set up in the system RAM of the PCI-3000. It will generally be called first upon implementation of this data acquisition procedure in the PCI.

The desired configuration of the system RAM in the PCI is laid out on the PCI-3000 RAM CHART which is included with this manual. The actual realization of this configuration only comes about after the loading of the four assembly language programs detailed in this manual. The purpose of this function is to initialize all the registers in the table (9000H through 905FH inclusive) to an initial value of zero. This is necessary since the default values of the registers upon power up are generally not zero. Any bit in the table must be set by the user in order to insure proper operation of the system.

The operation of the function is simple. It merely uses the IX register as a pointer to the first value in the table and then steps through the table initializing each register to zero. The B register is used for control in that it is decremented each time the pointer is incremented. Once it is decremented to zero program execution is halted.

The program also has one other minor function in that an arbitrary parameter (one byte) must be passed to it in the calling statement. This parameter, along with the 'program executed correctly byte' (4FH), should then be returned upon proper execution of the program. The passed parameter is stored in the byte just preceding the sample table in system RAM (8FFFH). This just provides a simple means to check the basic input and output mechanisms of the PCI system firmware, as well as its ability to interact with the system RAM. It also satisfies the PCI's constraint in that all functions are required to provide some output.

To call this function in ascii protocol use:

$A0,0,XX(BYTE)

where the second 0 corresponds to the microprocessor board id; 0 specifies all boards. BYTE specifies any decimal integer between 0 and 255.

In binary protocol use:

24 42 31 id 04 12 XX YY

where:
- 24 42 31 is the header;
- id is the microprocessor board id, can be 0 as in ascii;
- 04 is number of bytes to follow;
- 12 is function label (if function is entered as specified);
- AA is the arbitrary byte parameter; and
- XX YY is the modulo 16 check sum (low byte high byte format).

Note this checksum is only a checksum of the bytes following the 'number of bytes to follow' byte.

See chapter five of the PCI-3000 technical reference manual for additional infomation.

B. TDEFINE.ASM MANUAL

TDEFINE (define table) is a Z80 assembly language program that is designed to define the membership of group N, as well as the sampling period of group N, where N is presently 1 or 2. It only has the ability to add members to a group. However it can be used to change the sampling period of a group without affecting the membership of a group. In order to remove members from a group it is necessary to reinitialize the whole sample table (using TINITIAL) and redefine the whole table (using TDEFINE).

When calling TDEFINE certain parameters must be passed in the calling statement. The parameters must be passed in the order in which they are defined below.

Group number: This is either a 1 or a 2 to signify whether the data to follow is to be used in the definition of group one or two.

Sampling period: This is a two byte word that will either be the group one sampling period divided by 16ms or the group two sampling period divided by the group one sampling period. If this word is intended to represent the group two sampling period, it must have its high byte set to zero. Even though the group two sampling period must be read in as a word, other functions can only accommodate a group two sampling period of a byte in length. If the calling statement is in ascii protocol it is only necessary for:

0 < group one sampling period < 65536.
0 < group two sampling period < 256.

Number of members: This is just a number to let the PCI know how many members you wish to define in this statement. Since the definition of each member requires two bytes there should be exactly twice as many bytes as there are members passed in this statement. This parameter is also used by the TDEFINE function to tell it when there are no more members to define. In addition, this parameter can be set to zero if it is just desired to change the sampling period of a particular group.

Member's number: This is just a parameter to signify which member's sample flag you wish to set. This number is the same as each particular member's offset position within the sample table. Where the offset is with respect to the first member at position 9004H. The cross correlation between each member and the offset of their location in the table is included on the following page.

Gain: This parameter must be included for every member to be defined in to the table even though it is only relevant for those entries corresponding to analog channels. The gain numbers are as follows:

00H for an analog gain of 1.0 and
all digital definitions.
01H for an analog gain of 10.
02H for an analog gain of 100.
03H for an analog gain of 1000.

Any entry not conforming to the above specifications may cause an error in the operation of the entire system, or the execution of this particular function.

The address is the actual system RAM address of the sample status byte corresponding to the listed function.

The binary offset is equivalent to the base channel number in binary protocol. This would be the number to define this functions membership into a group when TDEFINE is called using binary protocol.

The decimal offset is equivalent to the base channel number in ascii protocol. This would be the number to define this functions membership into a group when TDEFINE is called using ascii protocol.

The connector pins are just the Elco connector pin-out of the particular function with respect to the PCI-3000 distribution panel.

The function just reflects current knowledge as to what each channel number represents.

The table entitled "Offset vs. Function Cross Reference" follows.

Special Notes on Table Notation:

SMD: STERLING MOUNT DIGITAL
SMA: STERLING MOUNT ANALOG
TFD: TRAVELING FEED DIGITAL
TFA: TRAVELING FEED ANALOG
TSD: TELESCOPE DIGITAL
TSA: TELESCOPE ANALOG
CYRO: CRYOGENICS
SPD: SPARE DIGITAL
CAL: CAL - SIG/REF
LOS: LO SELECT
IFS: IF SELECT

OFFSET VS. FUNCTION CROSS FEFERENCE

ADDRESS	BINARY OFFSET	DECIMAL OFFSET	CONNECTOR PINS	FUNCTION
9004H	00H	0	SMA: A - B	REFRIGERATOR TEMP.
9005H	01H	1	C - D	FE AMBIENT TEMP.
9006H	02H	2	E - F	DEWAR VACUUM
9007H	03H	3	H - J	LO LEVEL
9008H	04H	4	K - L	
9009H	05H	5	M - N	
900AH	06H	6	P - R	
900BH	07H	7	IFS: A - B	LO SELECT ANALOG.
900CH	08H	8	TFA: A - B	REFRIGERATOR TEMP.
900DH	09H	9	C - D	FE AMBIENT TEMP.
900EH	0AH	10	E - F	DEWAR VACUUM
900FH	0BH	11	H - J	LO LEVEL
9010H	0CH	12	K - L	
9011H	0DH	13	M - N	
9012H	0EH	14	P - R	
9013H	0FH	15	IFS: C - D	LO SELECT ANALOG.
9014H	10H	16	CYRO: A - B	PRESSURE SUPPLY S.M.
9015H	11H	17	C - D	PRESSURE RETURN S.M.
9016H	12H	18	E - F	REFRIGERATOR CURRENT S.M.
9017H	13H	19	H - J	PRESSURE SUPPLY T.F.
9018H	14H	20	K - L	PRESSURE RETURN T.F.
9019H	15H	21	M - N	REFRIG. CURRENT T.F.
901AH	16H	22	IFS: E - F	LO SELECT ANALOG.
901BH	17H	23	IFS: H - J	LO SELECT ANALOG.
901CH	18H	24	TSA: A - B	WIND SPEED
901DH	19H	25	C - D	WIND DIRECTION
901EH	1AH	26	E - F	OUTSIDE TEMPERATURE
901FH	1BH	27	H - J	BRAKE TEMPERATURE
9020H	1CH	28	A1: 1 - 2	GENERAL ANALOG 1
9021H	1DH	29	A2: 1 - 2	GENERAL ANALOG 2
9022H	1EH	30	A3: 1 - 2	GENERAL ANALOG 3
9023H	1FH	31	A4: 1 - 2	GENERAL ANALOG 4

DIGITAL BOARD ONE			DIGITAL BOARD TWO		
9024H	20H	32	902CH	28H	40
9025H	21H	33	902DH	29H	41
9026H	22H	34	902EH	2AH	42
9027H	23H	35	902FH	2BH	43
9028H	24H	36			
9029H	25H	37			
902AH	26H	38			
902BH	27H	39			

PCI-3000 RAM CHART

Memory Location	Bit 0	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7	
9000H	----- Low Byte of Group 1 Sampling Period Multiplier ------								ASAMPL
9001H	----- High Byte of Group 1 Sampling Period Multiplier -----								ASAMPH
9002H	----- Low Byte of Group 2 Sampling Period Multiplier -----								
9003H	----- High Byte of Group 2 Sampling Period Multiplier -----								

Analog (Section 1) Bank 0 = 32 Channels

9004H . . . 9023H	Reserved for Analog Gain	Reserved for possible Group 3 definition.	Group 1 Flag	Group 2 Flag	BTABLE
	Digital (Sec. 2, Bank 1) 8 Ch.				
9024H . . . 9028H					
	Digital (Sec. 3, Bank 2) 4 Ch.				
902CH . . . 902FH					
	Reserved (Sec. 4, Bank 3) 32 Channels				
9030H . . . 904FH					

	VARIABLES (ADDRESSES)		CODE		
				START	END
9050H	LIMIT				
9051H	COUNT	TINITIAL	XX 12	9290	--
9052H	CKFLAG	TDEFINE	YY 13	9210	9270
9053H	INDEX	PCIONOFF	ZZ 14	9160	9203
9054H	RELOAD LOW BYTE -> RELOAD	MCSAMPLE		9060	9154
9055H	RELOAD HIGH BYTE				
9056H	ADDR				
9057H	STATUS				
	VARIABLE PNPEL H = 90 ANNEL L = 60				

****

1. FUNCTION OPERATION:

Initially the CKFLAG register and IX pointer are configured for a group one definition. GETPAR is then used to determine which group is to be defined. If it is group two, IX and CKFLAG are reconfigured. With the correct configuration of the IX and CKFLAG registers the rest of the program has been written to be transparent to which group is actually being defined. Due to this it should be easy to expand this program to define another group.

Next the group sampling period is loaded into the correct registers as pointed to by the IX register. We then read in the number-of-members parameter and check to see if this is zero in which case program execution is terminated. Otherwise it is used for program control in register B. This register will be decremented after processing each gain parameter, and once it is decremented to zero program execution is halted.

Finally we enter the loop controlled by the B register. Here the IX register points to the first entry in the table. The member's number is then added as an offset to the IX register. The IX register now points to the correct entry in the table to process and the proper flag is set. Next the gain is read and or'ed to the entry pointed to by the IX register. This process is repeated as long as register B is not zero.

To confirm correct operation of the function the hex character 4F is passed as output at the end of the program.

2. EXPANSION TO MORE GROUPS:

To expand this function for the possible definition of three groups one simply needs to do the following:

Create a location in which to store the group three sampling frequency. This could be done by increasing the system RAM defined after the group two sampling frequency bytes at 9002H & 9003H.

Create a test to see if the first parameter passed is a 3, for the definition of group three.

If the first parameter is a three, initialize CKFLAG to 20H for the group three flag bit, and point to the correct location in which to store the group three sampling period.

The only problem should be that the way MCSAMP is currently written the group identifier byte at the beginning of the output packet would identify group three as group 128. This could be remedied without too much trouble or just accepted.

To call this function in ascii protocol use:

$A0,0,YY(group number, sampling period, number of members, member's number, gain, member's number, gain ...).

In binary protocol use:

24 42 31 id bc 13 parameters XX YY

id is the microprocessor board id; bc represents bytes to follow byte = 2*(# of functions to define) + 7.

Parameters are detailed above and must be entered as hex bytes XX YY is the modulo 16 checksum.

C. PCIONOFF.ASM MANUAL:

PCIONOFF is a Z80 assembly language program that is designed to configure the buffers and reload the counter such that MCSAMP will be called at intervals of 16ms. Its other purpose is that on every other call it loads the reload counter with zeros such that the periodic calling of MCSAMP will cease.

Initially, the word representing the rate at which MCSAMP is to be called is loaded into a working area of RAM called the reload area. This is done so that when the function is in its off phase it does not have to erase the original word. This saves you having to reenter the desired period of the MCSAMP function each time you want to start it. This word which is actually the sampling period divided by 16ms will be hereafter referred to as the reload word.

To determine what phase (on/off) to enter this function defines a STATUS byte. If STATUS equals 0 the function will enter its turn on mode, set STATUS to one, and output this reset value to the terminal to signify that MCSAMP has been activated. Alternately if the initial status is 1 the function will enter its turn-off mode, reset the status, and output this reset value to the terminal to signify that MCSAMP has been terminated. Upon correct operation the program will also transmit the 'program executed correctly byte' (4F Hex) along with the reset value.

To call this function in ascii protocol use:

$A0,0,ZZ

In binary protocol use:

24 42 31 id 03 15 15 00.

Since there are no unknowns in the part of the statement used to compute the sum, XX and YY can be given.

1. THE ON PHASE, STATUS = 0, SET TO 1:

First, the BFLAG buffer is set up. This is just a buffer containing all the flags used throughout the PCI-3000 firmware. Flags 18H through 1FH are reserved for user flags. GOPSBY is used to write to buffer number one. Bit 7 in register C specifies write, while bits 0-6 specify the buffer number. Register B specifies that zero is the byte to be written and registers DE specify that flag 18H is the flag we wish to write to. Hence the user flag representing MCSAMP will be 18H.

Next the scheduler buffer is setup, this buffer contains three bytes for every flag in BFLAG. These three bytes correspond to the flag number and the low and high bytes of the starting address for the corresponding function. In our case the user flag 18H and the starting address of MCSAMP (9060H) are written into this buffer using the firmware function GOPNEB.

Finally we initialize the BCKLK buffer. This buffer is responsible for the periodic calling of functions. Each entry contains five bytes with one byte corresponding to the function flag number, two bytes corresponding to a reload value, and two bytes corresponding to a count value. Every 16ms we decrement the reload value until it reaches zero at which time the associated flag is set and the reload value is set to the value in count. At this point the flag entry is initialized to 18H while the count and reload words are both initialized to zero.

Also in the on phase the ADDR register corresponding to the analog board PCI-3000 bus address is set to 0. The INDEX register is initialized to the value for the group two sampling period byte. This byte is actually the period at which group two is sampled divided by the period at which group one is sampled.

To conclude the on phase the reload word in our variable area of RAM is copied into the reload and count location of the BCLKA buffer. This will then start the periodic call of MCSAMP.

2. THE OFF PHASE, STATUS = 1, RESET TO 0:

In the off phase it is only necessary to clear the reload and count values in the BCLKA buffer. This is done by writing zeros to the reload word in our variable area of RAM and copying these values to the reload and count locations of the BCLKA buffer.

NOTE:

For additional information see the buffer information in chapter 2 of the PCI-3000 technical manual, and the reload and setup functions in the analog read example given on pages 4-7 to 4-18 of the same manual.

D. MCSAMP.ASM MANUAL:

MCSAMP is a Z80 assembly language program that is designed to index through a table and sample various inputs as determined by their corresponding flag status. This subroutine will be called intrinsically by the reload function in the PCI-3000 firmware.
Presently it is configured to discriminate between two groups and sample the appropriate one at the appropriate interval. This function was designed for easy expansion and reconfiguration.

1. SECTION ONE:

The function itself consists of five main parts. The first part is responsible for setting up the appropriate protocol, output port, and check flag status. To change the protocol the appropriate number must be entered into the PRTCL register. 01 specifies ascii protocol while 02 specifies binary protocol. For more information consult the protocol section of this manual or the PCI-3000 technical manual pages 3-38, 3-39. To redirect the output to a different output port the PORTID register must be loaded with a value corresponding to the desired output port and the location of the PRTCL and PPARMA registers must be changed accordingly:

Serial port A:
PRTCL = 407AH, PPARMA = 406BH (PORTID) = 04H
Serial port B:
PRTCL = 4091H PPARMA = 4082H (PORTID) = 08H
Parallel port C:
PRTCL = 4106H PPARMA = 40F7H (PORTID) = 10H

Note, PORTID = 4023H regardless of protocol or output port. Only the byte corresponding to this memory location changes.

As this function is presently defined the group two sampling period is an integer multiple of the group one sampling period, where this integer is between 0 and 255. The index register initially contains this integer and is decremented every time MCSAMP is called. Group two is sampled only when this register is decremented to zero. At this time the index register is reinitialized and the CKFLAG register is reconfiguered to the status of the group two sample flags.

To expand this function to three groups the TDEFINE function must be modified to allow for the definition of the third flag. ALSO MCSAMP must have a means by which at certain intervals the CKFLAG status is configured to sample group three.

Before GETTXB is called the desired output port is examined to see if it is clear. If the output port is not clear the sampling sequence is skipped and control is transferred to the end of the program where the system is reconfigured to its

original status. The reason we do not wish to let GETTXB wait until the buffer is clear before resuming execution is that the system will hang up each time the buffer is not clear.

To see if the output port is clear we examine the byte corresponding to this buffer status. Specifically we see if the first bit is set. For the available output ports the byte location is listed below:

Port A: 421A (Hex) Port B: 421B (Hex) Port C: 421C (Hex)

Finally the IX register is pointed to the beginning of this table, the transmit buffer is initialized using GETTXB, and the number of the group to be sampled is placed into the transmit buffer.

2. SECTION TWO:

In this section analog board one is sampled. The functions on this board are represented in locations 9004H through 9023H of system RAM. The COUNT register will specify the offset of the IX register from its initial value. This in turn represents the base channel number of the function to be sampled.

The LIMIT register is initialized to the length of this section. It is used to determine when all the entries have been checked and it is time to move on to sample the next section. This occurs when LIMIT is decremented to zero.

To determine whether or not to sample a function, its corresponding entry in the table is anded with CKFLAG. If its sample flag is set it is sampled, otherwise control is transferred such that the next entry in the table will be examined.

To sample an analog channel the firmware function TIODRV must be used with the accumulator set to 04H. Further requirements dictate that register C contains the value of the analog board's PCI-3000 bus address (00H for this board). Also register B must contain the adjusted channel number of the function to be sampled. The adjusted channel number is just the base channel number offset by the number representing the analog gain multiplied by 64. After calling TIODRV the word representing the value of this function will reside in registers BC. This word is then placed in the transmit buffer using PUTPAR.

For additional information see PCI-3000 user manual appendix B-33 to B-43 and TIODRV, PUTPAR spec sheets (PCI-3000 technical manual pages 3-31 & 3-33, respectively).

3. SECTION THREE:

In this section the first digital board is sampled. This board is configured as bank 1 in the PCI. The board is set up to receive eight bytes of digital inputs at addresses 10H through

17H. These addresses correspond to locations 9024H through 902BH in the sample table set up in the system RAM.

The same procedure is used to step through the table entries as was used in section two. For this section the LIMIT register is set to 08H for the 8 channels addressed by the board. It is decremented in the same manner as before.

The sampling process is the same except that the register configuration before calling TIODRV now requires the accumulator to be set to 02H and register C to contain the corresponding bus address.
Also, for these digital reads the data returned is a single byte rather than a two byte word as was the case in the analog section.

4. SECTION FOUR:

In this section the second digital board is sampled. This board is configured as bank 2 in the PCI. The board is set up to receive four bytes of digital input at addresses 20H through 23H. These addresses correspond to locations 902CH through 902FH in the sample table set up in the system RAM. The other four addresses accessed by this board (24H through 27H) are configured as digital output registers.

The exact same procedure is used in this section as was used in section three. However, due to the implementation of only four digital input channels on the board, the LIMIT register is set to 04H in this section.

5. SECTION FIVE:

In this section the Transmit buffer is completed as per the protocol specifications. It is then sent as specified by the value of PORTID, and the choice of PPARMA. Next PORTID and the IY register are returned to their original status. In closing the user flag set by the reload function to call this subroutine is cleared.

NOTE:

If additional I/O boards are added it would be a simple matter to expand the routine such that a section after section four could be set up to sample these boards in a manner analogous to one of the previous sections depending on the type of the board. Additional space was provided in the sample table such that 32 channels could be accommodated between register locations 9030H and 904FH inclusive. Additional space could be added by reconfiguring the RAM set up. One must be aware of the restrictions imposed by the watchdog timer and transmit buffer limitations when expansions of this magnitude are being considered.

IV. FIRMWARE ERROR CHECKING:

Throughout the functions you will occasionally see a group of assembly commands like the following.

```
RLCA
JR C, ERROR
```

These commands will only be found directly after a call to a firmware function. Upon completion of a firmware function the accumulator contains the function status. If bit 7 was set this indicates that an error was incurred during execution of this function. This group of commands just checks the error status and if an error was detected (bit 7 set) control is transferred to the error section of the program which looks like this.

```
ERROR     RRCA
CALL RETERR
RET
```

What happens here is that register A is returned to its original status and the function call RETERR is called. This function merely returns the correct error information to the output stream. Function execution is then terminated.

Error checking of the firmware functions was not implemented in the MCSAMP subroutine. This was due to several reasons:

1. There is no variable external intervention in this subroutine.

2. The subroutine was paired down to optimize for speed.

3. The input output buffer manipulations were such that it was unclear as to what the error checking subroutine would accomplish, thus possibly making debugging even more difficult.

Due to these reasons and the fact that the only possible source of error that could disable the system seemed to be the entry for analog gain being incorrect, error checking was not evoked. The possibility of an incorrect analog gain affecting the system was remedied by extracting only the significant bits from the status bytes. Hence, once the system is running the only outcome of an error would seem to be incorrect output, as a result of an incorrect definition. This seemed reasonable in consideration of the above criteria.

```
;****************************************
;****************************************
;    IT(TABLE STATUS CODE)
;THIS IS A USER CREATED FUNCTION WHICH
;WILL BE USED TO INITIALIZE THE DATA
;ACQUISITION TABLE.
;****************************************

;******* PROGRAM CODE & DOCUMENTATION ****
;****************************************

;******* GET TABLE STATUS ****************
               LD C, 01H
               CALL GETPAR
               RLCA
               JR C, ERROR
;     THIS IS A PARAMETER RETRIEVING PROCE-
;DURE WHICH:  SPECIFIES THAT THE PARAMETER
;TO RETRIEVE IS A BYTE, RETRIEVES THE
;PARAMETER FROM THE BUFFER PLACING IT INTO
;REGISTER E, AND CHECKS TO SEE IF BIT 7 IN
;REGISTER A IS SET IN WHICH CASE AN ERROR
;HAS INCURRED AND CONTROL IS TRANSFERRED
;TO THE ERROR ROUTINE.

;******* PREPROCESSING SECTION ***********
               LD IX, 8FFFH
               LD (IX+0), E
               LD B, 60H
;     HERE THE POINTER IS INITIALIZED TO
;THE ADDRESS ONE BYTE BEFORE THE FIRST
;BYTE IN THE TABLE.  THIS BYTE IS THEN
;ASSIGNED THE TABLE STATUS VARIABLE PASSED
;IN THE FUNCTION CALL.  FINALLY REGISTER B
;IS INITIALIZED TO A VALUE REPRESENTING
;THE NUMBER OF BYTES IN THE TABLE.

;******* TABLE INITIALIZATION ************
LOOP           LD (IX+1), 00H
               INC IX
               DEC B
               JR NZ, LOOP
;     HERE ALL THE BYTES IN THE TABLE ARE
;INITIALIZED TO 0.

;******* RETURN CHECK EXEC. PARAMETER ****
               LD C, 01H
               LD E, 4FH
               CALL PUTPAR
               RLCA
               JR C, ERROR
```

```
;     OUTPUT 'FUNCTION EXECUTED CORRECTLY'
;BYTE (4FH).
;******* RETURN STATUS PARAMETER *********
                LD HL, 8FFFH
                LD E, (HL)
                LD C, 01H
                CALL PUTPAR
                RLCA
                JR C, ERROR

;******* FUNCTION TERMINATION ************
                RET

;******* ERROR DEFINITION*****************
ERROR           RRCA
                CALL RETERR
                RET
;     REESTABLISHES REGISTER A AND CALLS
;THE FIRMWARE ROUTINE 'RETERR' WHICH
;DIAGNOSES THE ERROR, AND OUTPUTS IT TO
;THE TERMINAL.  FUNCTION EXECUTION IS THEN
;TERMINATED.

;******* FIRMWARE ROUTINE ADDRESSES ******
GETPAR EQUAL 0055H
RETERR EQUAL 006DH
PUTPAR EQUAL 0064H
```

TINITIAL.HEX

```
:109290000E01CD5500073829DD21FF8FDD73000653
:1092A00060DD360100DD230520F70E011E4FCD6481
:1092B0000007380D21FF8F5E0E01CD6400073801D5
:0692C000C90FCD8500C9B5
:00000001FF
```

```
;*****************************************
;*****************************************
;    DT(GROUP #, SAMPLING RATE, # OF
;MEMBERS, MEMBER'S NUMBER, GAIN) NOTE IF
;THE MEMBER IS A DIGITAL FUNCTION A ZERO
;MUST BE ENTERED FOR THE GAIN.  THIS IS A
;USER CREATED FUNCTION DESIGNED TO MAKE
;A TABLE THAT WILL INDICATE WHICH GROUP(S)
;A FUNCTION BELONGS TO, THE SAMPLING FREQ
;OF THAT GROUP, AND THE GAIN OF THE
;CHANNEL FOR ANALOG GROUPS.
;*****************************************

;******* PROGRAM CODE & DOCUMENTATION ****
;*****************************************

;******* VARIABLE INITIALIZATIONS ********
                LD IX, ASAMPL
;    LOADS TABLE BASE ADDRESS INTO IX REG.

                LD HL, CKFLAG
                LD (HL), 40H
;SETS UP HL REGISTER AS A PONTER TO A
;REGISTER.  THIS REGISTER IS LOADED WITH
;THE GROUP ONE DEFINITION NUMBER.

;******* LOAD GROUP NUMBER INTO REGISTER E
                LD C, 01H
                CALL GETPAR
                RLCA
                JR C, ERROR
;     THIS IS A PARAMETER RETRIEVING PROCE-
;DURE WHICH:  SPECIFIES THAT THE PARAMETER
;TO RETRIEVE IS A BYTE, RETRIEVES THE
;PARAMETER FROM THE BUFFER PLACING IN INTO
;REGISTER E, AND CHECKS TO SEE IF BIT 7 IN
;REGISTER A IS SET IN WHICH CASE AN ERROR
;HAS INCURRED AND CONTROL I TRANSFERRED
;TO THE ERROR ROUTINE.

;******* DETERMINE GROUP NUMBER **********
                LD C, 00H
                DEC E
                JR Z, LDSAMP
;     CONFIGURES REGISTER C SO THAT NEXT
;TIME GETPAR IS CALLED A WORD WILL BE READ.
;CHECKS TO SEE IF THE GROUP NUMBER FOR
;INCOMING SET OF DATA IS A 1 IN WHICH
;CASE CONTROL IS TRANSFERED TO LDSAMP.
```

```
;******* GROUP TWO CONFIGUERATION ********
                INC IX
                INC IX
                LD A, (HL)
                SLA A
                LD (HL), A
;    ONLY EXECUTED IF NOT GROUP ONE, THIS
;SECTION CHANGES THE GROUP DEFINITION
;NUMBER SO IT CORRESPONDS TO GROUP TWO &
;INCREMENTS THE INDEX REGISTER SO IT
;CORRESPONDS TO THE BYTE REPRESENTING
;GROUP TWO'S SAMPLING FREQUENCY.  IT ALSO
;RECONFIGURES REGISTER C SO THAT A BYTE
;WILL BE READ NEXT TIME GETPAR IS CALLED.

;******* PROCESS SAMPLE FREQUENCY ********
LDSAMP              CALL GETPAR
                RLCA
                JR C, ERROR
;    PARAMTER RETRIEVING PROCEDURE WHICH
;LOADS THE SPECIFIED SAMPLING RATE INTO
;REGISTERS DE.

                LD (IX+0), E
                LD (IX+1), D
;LOADS THIS SAMPLING RATE INTO THE BYTE
;CORRESPONDING TO THE GROUP NUMBER.

;******* LOADS # OF MEMBERS **************
                LD C, 01H
                CALL GETPAR
                RLCA
                JR C, ERROR
;    PARAMETER RETRIEVING PROCEDURE WHICH
;LOADS THE NUMBER OF GROUP MEMBERS TO BE
;DEFINED INTO REGISTER E.  THIS PARAMETER
;IS MAINLY USED FOR INPUT CONTROL.

;******* TABLE DEFINITION CHECK **********
                LD B, E
                INC B
                DEC B
                JR Z, RETURN
;    REGISTER B IS GIVEN THE LOOP CONTROL
;INDEX AND THIS INDEX IS CHECKED TO SEE IF
;IT IS ZERO IN WHICH CASE THERE WOULD BE
;ZERO MEMBERS TO DEFINE AND THE PROGRAM
;CONTROL WOULD BE SENT TO RETURN AND
;TERMINATED.
```

```
;*******TABLE FILLING ********************
LOOP            LD C, 01H
                CALL GETPAR
                RLCA
                JR C, ERROR
;   PARAMETER RETRIEVING PROCEDURE WHICH
;LOADS THE FUNCTION TO BE ENTERED INTO THE
;TABLE INTO REGISTER E.

                LD IX, BTABLE
                ADD IX, DE
;LOADS THE BASE ADDRESS OF THE TABLE DATA
;SECTION INTO THE INDEX ARRAY AND ADDS THE
;INPUT LOCATION OFFSET TO THIS VALUE.

                LD A, (IX+0)
                OR (HL)
                LD (IX+0), A
;ORS THE VALUE OF THE GROUP DEFINITION
;WITH THE SPECIFIED BYTE.

                LD C, 01H
                CALL GETPAR
                RLCA
                JR C, ERROR
;PARAMETER RETRIEVING PROCEDURE WHICH
;LOADS THE ANALOG GAIN OF THE FUNCTION
;JUST ENTERED, INTO REGISTER E.

                LD A, (IX+0)
                OR E
                LD (IX+0), A
;ORS THE VALUE REPRESENTING THIS ANALOG
;GAIN WITH THE CONTENTS OF THE SPECIFIED
;REGISTER.
                DEC B
                JR NZ, LOOP
;DECREMENTS THE LOOP CONTROL INDEX IN
;REGISTER B, IF THIS IS NOT ZERO, CONTROL
;IS TRANSFERED BACK TO LOOP AND THE
;PROCESS IS REPEATED UNTIL THE LOOP INDEX
;IS 0, AT WHICH TIME CONTROL IS RESUMED
;SEQUENTIALLY AND THE PROGRAM TERMINATES.

;******* EXECUTION VERIFICATION **********
RETURN              LD C, 01H
                LD E, 4FH
                CALL PUTPAR
                RLCA
                JR C, ERROR
;    OUTPUT THE HEX CHARACTER 99 TO
;SIGNIFY COMPLETION OF PROGRAM EXECUTION.

                RET
;FUNCTION TERMINATION.
```

```
;******* ERROR DEFINITION ****************
ERROR           RRCA
                CALL RETERR
                RET
;    REESTABLISHES REGISTER A AND CALLS
; THE FIRMWARE ROUTINE 'RETERR  WHICH
;DIAGNOSES THE ERROR, AND OUTPUTS IT TO
;THE TERMINAL.  FUNCTION EXECUTION IS THEN
;TERMINATED.


;******* FIRMWARE ROUTINE ADDRESSES ******
GETPAR EQUAL 0055H
RETERR EQUAL 006DH
PUTPAR EQUAL 0064H

;******* WE EQUATE ************************
ASAMPL EQUAL 9000H
BTABLE EQUAL 9004H
CKFLAG EQUAL 9052H
```

TDEFINE.HEX

```
:10921000DD2100902152903640 0E01CD55000738D7
:1092200058 0E001D2808DD23DD237ECB2777CD5582
:1092300000073845DD7300DD72010E01CD550007D2
:1092400038374304052827 0E01CD550007382ADD9D
:1092500021 490DD19DD7E00B6DD77000E01CD55CD
:1092600000073815DD7E00B3DD77000520D90E013B
:0E9270001E4FCD6400073801C90FCD8500C91F
:00000001FF
```

```
;*****************************************
;*****************************************
;     THIS IS A RELOAD COUNT FUNCTION
;WHOSE RELOAD PARAMETER ENABLES THE CLOCK
;DRIVER PERIODICALLY TO CAUSE THE
;SCHEDULER TO CALL THE MAIN SAMPLING
;ROUTINE MCSAMP.  THIS FUNCTION WORKS ON
;AN ON-OFF BASIS IN THAT EACH TIME IT
;IS CALLED IT EITHER TURNS THE SAMPLING
;ROUTINE ON OR OFF.
;*****************************************

;******* FUNCTION CODE & DOCUMENTATION ***
;*****************************************

;******* SET UP RELOAD AREA IN RAM *******
                LD BC, 02H
                LD HL, ASAMPH
                LD DE, RELOAD
                INC DE
                LDD
                LDD
;     HERE THE WORD REPRESENTING THE
;SAMPLING PERIOD FOR GROUP ONE IS COPIED
;IN TO THE RELOAD AREA OF RAM.

                LD IX, RELOAD
;SETS POINTER TO HI BYTE OF RELOAD

;******* GET AND CHECK RELOAD STATUS *****
                LD HL, STATUS
                LD E, (HL)
                INC E
                DEC E
                JP NZ, STPSAM

;     CHECK TO SEE IF RELOAD FUNCTION IS
;TO CALL THE SETUP SUBROUTINE IF NOT SKIP
;SETUP CALLING STATEMENT.

;******* CALL SETUP FUNCTION FIRST *******
                LD (HL), 01H
;     RESET STATUS SO NEXT TIME FUNCTION IS
;CALLED IT WILL TERMINATE SAMPLING.

                LD B, 00H
                LD C, 81H
                LD DE, 18H
                CALL GOPSBY
;     SET UP BFLAG, LOADS REGISTER B (00H)
;TO BFLAG BUFFER 1 AS SPECIFIED BY
;REGISTER C AT USER FLAG NUMBER 18H AS
;SPECIFIED BY OFFSET IN REGISTERS DE.
```

```
;******* SET UP BSCHED BUFFER ************
                LD B, 18H
                LD C, 80H
                CALL GOPNEB
;     WRITE TO BSCHED BUFFER 0 AS SPECIFIED
;BY REGISTER C AT USER FLAG 18H (REG B).

                LD B, ANREDL
                CALL GOPNEB
;     LOAD LOW BYTE OF STARTING ADDRESS FOR
;SAMPLING ROUTINE INTO BSCHED BUFFFER.

                LD B, ANREDH
                CALL GOPNEB
;     LOAD HI BYTE OF STARTING ADDRESS FOR
;SAMPLING ROUTINE.  BUFFER NOW SET UP.

;******* SET UP BCLK BUFFER **************
                LD B, 18H
                LD C, 8BH
                CALL GOPNEB
;     USING USER FLAG 18 HEX (REG B), WRITE
;TO BCLK BUFFER B HEX (REG C).

                LD L, 04H
LOOP            LD B, 00H
                CALL GOPNEB
                DEC L
                JR NZ, LOOP
;     WRITE FOUR BYTES INITIALIZING BUFFER.
;THESE FOUR BYTES CORRESPOND TO THE HI AND
;LOW BYTES OF RELOAD AND COUNT, AND ARE
;ALL ZEROS.  THIS COMPLETES THE SETUP OF
;THE BCLK BUFFER.

                LD E, 00H
                LD HL, ADDR
                LD (HL), E
;     LOAD THE BUS ADDRESS FOR ANALOG BOARD
;ONE, BANK ZERO INTO THE ADDR VARIABLE.

                LD HL, 9002H
                LD C, (HL)
                LD HL, INDEX
                LD (HL), C
;     INITIALIZES INDEX REGISTER TO BYTE
;CORRESPONDING TO GROUP 2 SAMPLING RATE.

                JP BGNSAM
;     END OF SET UP.  JUMP TO BEGIN SAMPLE
;(BGNSAM).
```

```
;******* STOP SAMPLING (STPSAM)***********
STPSAM              LD (HL), 00H
;     RESET SAMPLE SO NEXT TIME FUNCTION IS
;CALLED IT WILL INITIATE SAMPLING.

                    LD (IX+0), 00H
                    LD (IX+1), 00H
;     LOADS ZERO HEX INTO RELOAD FUNCTION
;IN ORDER TO STOP SAMPLING ROUTINE.

;******* GET AND LOAD RELOAD VALUE TO BCLK
BGNSAM              LD HL, (BCLKA)
                    LD A, (HL)
                    SUB 4
;     POINT ACCUMULATOR AT BCLK OFFSET FOR
;RELOAD LOW BYTE

                    LD B, (IX+0)
                    LD C, 8BH
                    LD D, 00H
                    LD E, A
                    CALL GOPSBY
;     POINT TO RELOAD AREA IN RAM AND GET
;LO BYTE OF RELOAD VALUE.  WRITE TO BCLK
;BUFFER B HEX AT OFFSET IN REGISTERS DE.
;LO BYTE OF RELOAD NOW WRITTEN

                    LD B, (IX+1)
                    CALL AOASBY
;INCREMENTS POINTER AND LOADS HIGH BYTE OF
;RELOAD VALUE TO BCLK BUFFER.

;******* REPEAT FOR COUNT VALUE **********
                    LD B, (IX+0)
                    CALL AOASBY
;     SINCE COUNT AND RELOAD ARE TO BE
;EQUAL DECREMENT HL POINTER TO ADDRESS LO
;BYTE OF COUNT (RELOAD).  LOAD THIS VALUE
;TO BCLK BUFFER.

                    LD B, (IX+1)
                    CALL AOASBY
;REPEAT PROCESS FOR HIGH BYTE OF COUNT.

                    LD C, 01H
                    LD E, 4FH
                    CALL PUTPAR
                    RLCA
                    JR C, ERROR
;     OUTPUT 'FUNCTION EXECUTED CORRECTLY'
; BYTE (4FH)
```

```
;******* OUTPUT FUNCTION STATUS *********
                LD C, 01H
                LD HL, STATUS
                LD E, (HL)
                CALL PUTPAR
                RLCA
                JR C, ERROR
;     OUTPUT CONTENTS OF STATUS REGISTER
;INDICATING IF SAMPLING WAS INITIATED (1)
;OR TERMINATED (0).  WITH ERROR CHECKING.
;******* PROGRAM TERMINATION *************
                RET       END OF PROGRAM.

;******* ERROR DEFINITION ****************
ERROR                RRCA
                CALL RETERR
                RET

;******* FIRMWARE EQUATES ****************
AOASBY EQUAL 0043H
GOPNEB EQUAL 005BH
GOPSBY EQUAL 005EH
PUTPAR EQUAL 0064H
RETERR EQUAL 006DH

BCLKA  EQUAL 40E3H
ANREDH EQUAL 90H
ANREDL EQUAL 60H

;******* WE EQUATE ***********************
ASAMPH EQUAL 9001H
INDEX  EQUAL 9053H
RELOAD EQUAL 9054H
ADDR   EQUAL 9056H
STATUS EQUAL 9057H
```

PCIONOFF.HEX

```
:1091650001020021019011549013EDA8EDA8DD2115
:1091750054902157905E1C1DC2BF91360106000E0A
:1091850081111800CD5E0006180E80CD5B000660CB
:10919500CD5B000690CD5B0006180E8BCD5B002ED7
:1091A500040600CD5B002D20F81E0021569073218A
:1091B50002904E21539071C3C9913600DD360000EF
:1091C500DD3601002AE3407ED604DD46000E8B160F
:1091D500005FCD5E00DD4601CD4300DD4600CD4399
:1091E50000DD4601CD43000E011E4FCD640007385A
:1091F5000D0E012157905ECD6400073801C90FCDD2
:039205008500C918
:00000001FF
```

```
;*****************************************
;*****************************************
;     MCSAMP, THIS SUBROUTINE IS DESIGNED
;TO BE SET UP AND CALLED STARTING AT
;LOCATION 9060H IN SYSTEM RAM.  ITS
;PURPOSE IS TO SAMPLE CERTAIN CHANNELS AT
;CERTAIN INTERVALS AS SPECIFIED BY THE
;TABLE SET UP IN SYSTEM RAM.
;*****************************************

;******* PROGRAM CODE & DOCUMENTATION ****
;******* SECTION ONE **********************

;******* PROGRAM CONTROL ******************

               LD HL, PRTCL
               LD (HL), 02H
;   PLACES THE NUMBER CORRESPONDING TO THE
;DESIRED PROTOCOL INTO THE REGISTER
;CORRESPONDING TO THE DESIRED OUPTUT PORT.

               LD HL, CKFLAG
               LD (HL), 40H
;    PLACES THE TABLE 1 FLAG DISCRIMINATOR
;IN THE CHECK FLAG LOCATION IN MEMORY.

               PUSH IY
               LD A, (PORTID)
               PUSH AF
               LD IY, PPARMA
               LD A, 10H
               LD (PORTID), A
;    HERE THE IY AND PORTID REGISTERS ARE
;CONFIGURED SO AS TO SET UP THE OUTPUT
;PORT.  THE VALUE STORED IN PORTID CORR-
;ESPONDS TO THE PORT, AND MUST MATCH WITH
;THE CORRESPONDING CHOICE FOR PPARMA.

               LD HL, INDEX
               DEC (HL)
               JP NZ, SAMPLE
;    POINTS TO INDEX LOCATION IN MEMORY,
;DECREMENTS INDEX BY ONE AND JUMPS TO
;SAMPLE IF INDEX IS NOT EQUAL TO ZERO.
;(SAMPLES GROUP TWO AT A RATE EQUAL TO AN
;INTEGRAL NUMBER TIMES THE SAMPLING RATE
;OF GROUP ONE)  THIS OCCURS WHEN INDEX IS
;DECREMENTED TO ZERO.
```

```
                LD BC, (9002H)
                LD HL, INDEX
                LD (HL), C
                INC HL
                LD (HL), B
;     RESETS INDEX REGISTER TO BYTE COR-
;RESPONDING TO GROUP 2 SAMPLING FREQUENCY.

                LD HL, CKFLAG
                LD (HL), 80H
;     PLACES THE TABLE 2 FLAG DISCRIMINATOR
;IN THE CHECK FLAG LOCATION IN MEMORY.

SAMPLE          LD IX, BTABLE
;     THE IX REGISTER WILL BE INCREMENTED
;TO SEQUENTIALLY TEST ALL THE ENTRIES IN
;THE TABLE.  HERE IT IS INITIALIZED TO THE
;FIRST TABLE ENTRY.

                LD A, (CBFREE)
                AND 01H
                JP Z, END
;     CHECKS TO SEE IF THE TRANSMIT BUFFER
;FOR OUTPUT PORT C IS CLEAR, IF NOT THIS
;SAMPLE PASS IS TERMINATED.

                CALL GETTXB
;     THIS IS A FUNCTION CALL TO A FUNCTION
;DEFINED IN THE PCI-3000 FIRMWARE THAT
;WILL SET UP THE TRANSMIT BUFFER.

                LD A, (CKFLAG)
                RLCA
                RLCA
;     DETERMINE NUMBER OF GROUP TO BE
;SAMPLED BY CHECKING VALUE IN CKFLAG.

                LD C, 01H
                LD E, A
                CALL PUTPAR
;      PLACE GROUP NUMBER AT BEGINNING OF
;OUTPUT PACKAGE.
```

```
;******* SECTION TWO, ANALOG BOARD 1 *****
;****************************************

                LD HL, COUNT
                LD (HL), 0
;     HERE THE COUNT REGISTER IS INITIAL-
;IZED TO THE CHANNEL NUMBER CORRESPONDING
;TO THE FIRST ENTRY IN THE TABLE.

                LD HL, LIMIT
                LD (HL), 20H
;     HERE THE VALUE REPRESENTING THE
;LENGTH OF THE FIRST SECTION (ANALOG, BANK
;0) OF THE TABLE IS LOADED TO THE REGISTER
;CALLED LIMIT.

LOOPA                LD A, (CKFLAG)
                AND (IX+0)
                JP Z, NEXTA
;     CHECKS TO SEE IN THE APPROPRIATE FLAG
;IS SET,  IF SO THE APPROPRIATE CHANNEL IS
;SAMPLED.

                LD C, 00H
;     SETS UP PCI-3000 BUS ADDRESS FOR USE
;BY THE TIODRV FUNCTION.

                LD A, (IX+0)
                AND 03H
;     EXTRACTS VALUE FOR THE GAIN FROM
;TABLE ENTRY POINTED TO BY IX REGISTER.

                RRCA
                RRCA
;     MULTIPLIES THIS VALUE FOR ANALOG GAIN
;BY 64 USING TWO RIGHT ROTATIONS OF
;CONTENTS OF ACCUMULATOR.

                LD HL, COUNT
                OR (HL)
;     ORS THE VALUE STORED IN THE COUNT
;REGISTER TO THE ACCUMULATOR. THIS VALUE
;REPRESENTS THE BASE CHANNEL OFFSET FOR
;THE PARTICULAR ENTRY IN THE TABLE.

                LD A, 04H
                CALL TIODRV
;     LOADS THE NOW CONFIGURED CHANNEL
;NUMBER INTO REGISTER B AND CONFIGURES
;REGISTER A TO INDICATE AN ANALOG READ
;WITHOUT COMPENSATION IS TO BE PERFORMED
;AT A CERTAIN BUS ADDRESS (REGISTER C) AND
;CERTAIN CHANNEL NUMBER (REGISTER B), BY
;THE TIODRV FIRMWARE FUNCTION.
```

```
                PUSH BC
                POP DE
;     PLACES RETURNED WORD INTO THE DE
;REGISTER.

                LD C, 00H
                CALL PUTPAR
;     CALLS PUTPAR WHICH PLACES THE WORD IN
;REGISTERS DE INTO THE TRANSMIT BUFFER.

NEXTA                INC IX
                LD HL, COUNT
                INC (HL)
;     INCREMENT IX REGISTER TO POINT TO
;NEXT ENTRY IN THE TABLE.  INCREMENTS
;THE COUNT REGISTER TO UPDATE OFFSET.

                LD HL, LIMIT
                DEC (HL)
                JP NZ, LOOPA
;      WHILE LIMIT REGISTER NOT EQUAL TO
;ZERO CONTINUE SAMPLING SECTION ONE,
;OTHERWISE START SAMPLING SECTION THREE.
```

```
;******* SECTION THREE, DIGITAL BOARD 1 **
;****************************************
               LD HL, COUNT
               LD (HL), 10H
               LD HL, LIMIT
               LD (HL), 08H
;    LOADS THE VALUE FOR THE BASE BUS
;ADDRESS FOR SECTION TWO INTO THE COUNT
;REGISTER.  INITIALIZES THE LIMIT REGISTER
;TO REPRESENT THE LENGTH OF SECTION TWO
;(BANK 1, DIGITAL BOARD 1)

CONTDA              LD A, (CKFLAG)
               AND (IX+0)
               JP Z, NEXTB
;    CHECKS TO SEE IF THE APPROPRIATE
;GROUP FLAG IS SET, IF SO SAMPLES THAT
;INPUT, OTHERWISE JUMP TO NEXTB AND GO ON
;TO CHECK NEXT ENTRY IN TABLE.

               LD HL, COUNT
               LD C, (HL)
;    LOADS THE C REGISTER WITH THE CURRENT
;VALUE OF THE OFFSET.

               LD A, 02H
               CALL TIODRV
;    SAMPLE DIGITAL REGISTER AS SPECIFIED
;BY ADDRESS CONTAINED IN REGISTER C.

               PUSH BC
               POP DE
               LD C, 01H
               CALL PUTPAR
;    LOAD SAMPLED VALUES TO TRANSMIT
;BUFFER.

NEXTB               INC IX
               LD HL, COUNT
               INC (HL)
;    INCREMENTS POINTER TO POINT TO NEXT
;ENTRY IN TABLE.  INCREMENTS THE OFFSET
;STORED IN COUNT.

               LD HL, LIMIT
               DEC (HL)
               JP NZ, CONTDA
;    DECREMENTS LIMIT COUNTER TO REFLECT
;NUMBER OF ENTRIES LEFT TO SAMPLE.  IF NOT
;ZERO CONTINUE SAMPLING THIS SECTION,
;OTHERWISE PROCEED TO SECTION FOUR.
```

```
;******* SECTION FOUR, DIGITAL BOARD 2 ***
;****************************************
                LD HL, COUNT
                LD (HL), 20H
                LD HL, LIMIT
                LD (HL), 04H
;     LOADS THE VALUE FOR THE BASE BUS
;ADDRESS FOR SECTION FOUR INTO THE COUNT
;REGISTER.  INITIALIZES THE LIMIT REGISTER
;TO REPRESENT THE LENGTH OF SECTION FOUR
;(BANK 2, DIGITAL BOARD 2)

CONTDB              LD A, (CKFLAG)
                AND (IX+0)
                JP Z, NEXTC
;     CHECKS TO SEE IF THE APPROPRIATE
;GROUP FLAG IS SET, IF SO SAMPLES THAT
;INPUT, OTHERWISE JUMP TO NEXTC AND GO ON
;TO CHECK NEXT ENTRY IN TABLE.

                LD HL, COUNT
                LD C, (HL)
;     LOADS THE C REGISTER WITH THE CURRENT
;VALUE OF THE OFFSET.

                LD A, 02H
                CALL TIODRV
;     SAMPLE DIGITAL REGISTER AS SPECIFIED
;BY ADDRESS CONTAINED IN REGISTER C.

                PUSH BC
                POP DE
                LD C, 01H
                CALL PUTPAR
;LOAD SAMPLED VALUES TO TRANSMIT BUFFER.

NEXTC               INC IX
                LD HL, COUNT
                INC (HL)
;     INCREMENTS POINTER TO POINT TO NEXT
;ENTRY IN TABLE.  INCREMENTS THE OFFSET
;STORED IN COUNT.

                LD HL, LIMIT
                DEC (HL)
                JP NZ, CONTDB
;     DECREMENTS LIMIT COUNTER TO REFLECT
;NUMBER OF ENTRIES LEFT TO SAMPLE.  IF NOT
;ZERO CONTINUE SAMPLING THIS SECTION
;OTHERWISE POLISH AND SEND BUFFER.
```

```
;******* SECTION 5, WRAP PACKAGE AND SEND.
;****************************************

                CALL FINTXB
;READYS TRANSMIT BUFFER FOR TRANSMISSION.

                CALL TXENAB
;TRANSMIT BUFFER OVER THE PORT SPECIFIED
;BY PORTID.

END             POP AF
                LD (PORTID), A
;RESTORE ORIGIAL PORTID BYTE

                POP IY
;RESTORE IY REGISTER

                LD B, 00H
;    BYTE TO BE LOADED INTO USER FLAG TO
;CLEAR IT.

                LD C, 81H
                LD DE, 18H
                CALL GOPSBY
;    WRITE TO BFLAG BUFFER 1 USER FLAG 18
;THIS WILL CLEAR THE USER FLAG.

                RET
;END OF SUBROUTINE FINALLY.
```

```
;******* FIRMWARE EQUATES ****************

FINTXB EQUAL 004FH
GETTXB EQUAL 0058H
GOPSBY EQUAL 005EH
PUTPAR EQUAL 0064H
TIODRV EQUAL 0085H
TXENAB EQUAL 008EH

PORTID EQUAL 4023H
PPARMA EQUAL 40F7H
PRTCL  EQUAL 4106H
CBFREE EQUAL 421CH
;    CALLING ADDRESSES OF CANNED FUNCTIONS
;PROVIDED IN THE PCI-3000 ROM.

;******* WE EQUATE ************************
LIMIT  EQUAL 9050H
COUNT EQUAL 9051H
CKFLAG EQUAL 9052H
INDEX  EQUAL 9053H
BTABLE EQUAL 9004H
;     ADDRESSES ASSIGNED TO SPECIFIC
;VARIABLES.  NOTE ALL ADDRESSES BELOW
;8000H ARE ADDRESSES SPECIFYING REGISTERS
;WRITTEN TO BY THE PCI-3000 ROM.
```

MCSAMP.HEX

```
:1090600021064136022152903640FDE53A2340F573
:10907000FD21F7403E1032234021539035C28F909E
:10908000ED4B0290215390712370215290368ODD78
:1090900021049O3A1C42E601CA4891CD58003A5248
:1090A0009007070E015FCD6400215190360021 50DA
:1090B0009036203A5290DDA600CAD5900E00DD7E93
:1090C00000E6030F0F215190B63E04CD8500C5D1B7
:1090D0000E00CD6400DD232151903421509035C223
:1090E000B39021519036102150903608 3A5290DDBD
:1090F000A600CA05912151904E3E02CD8500C5D1F2
:109100000E01CD6400DD232151903421509035C2F1
:10911000EC9021519036202150903604 3A5290DD47
:10912000A600CA35912151904E3E02CD8500C5D191
:109130000E01CD6400DD232151903421509035C2C1
:109140001C91CD4F00CD8E00F1322340FDE1060091
:099150000E81111800CD5E00C96A
:00000001FF
```

V. CREATING FUNCTIONS FOR THE PCI-3000:

Here we will try to outline the creation process for PCI-3000 functions. This will include comments on all steps from encoding to downline loading.

A. CREATION OF CODE:

The PCI runs on a Z80A microprocessor with a 4 MHz system clock. We have found no instances where standard Z80 assembly code won't operate on the PCI. However, to make life easier the system firmware has a large group of functions which are designed primarily to facilitate input/output operations. These functions are very useful and are well detailed in chapter 3 of the PCI-3000 technical manual. The end result of any assembly programing for the PCI-3000 is just setting up and manipulating these functions in the desirable way. These functions, as opposed to those functions listed in chapter 4, can only be called intrinsically. Likewise, those functions in chapter 4 appear to be of no use to the programmer as they seem to only respond to external calls. Probably the most important thing to watch out for when programing the PCI is the use of the IY register. This register is used extensively by the system firmware and it is probably best to avoid it use. The only time it should be used is to set up I/O ports for the system firmware, when these ports won't be configured by external calls. This is what is done in the subroutine MCSAMP.

B. ASSEMBLING THE CODE:

Presently, all the code must be assembled using the 2500 A.D. Z80 assembler which runs on IBM compatible (MSDOS) machines. To assemble your code just insert disk #2 from the assembler packet into your resident drive and type X80. You will then be prompted for the output path, the default usually will suffice. You must then enter your file pathname; the suffix is not necessary if it is .asm. The default output file pathname is the same as that for the input, except the suffix is now .obj.

C. LINKING THE CODE:

Once you have successfully assembled your code you must link it to a form acceptable by the down line loading program. To do this call the linker, LINK (also on disk 2). Specify the pathname for your object file. Once again there is a default suffix of .obj. Next the start address of your code is prompted for. This must be given in hex notation. The user RAM area of the PCI is from locations 8000H to 9FFF with our data acquisition system taking up locations 8FFF through 9300H. Next you will encounter three more filename prompts. A carriage return for all three will suffice and give you an output pathname the same as the input with the suffix changed. Finally you are prompted for your choice of output option. Here it is very important that you pick option H. This brings the only format that is readable by our downline loading program. By picking this option your output

filename's suffix should be .hex. A documented example of this type of format (intel hex) is given on the following page. Once the above steps are completed correctly the program can be automatically downline loaded using one of our C programs.

D. EXPLANATION OF INTEL HEX FORMAT USING TINITIAL.HEX

```
:109290000E01CD5500073829DD21FF8FDD73000653
:1092A00060DD360100DD230520F70E011E4FCD6481
:1092B0000007380D21FF8F5E0E01CD64000738 01D5
:0692C000C90FCD8500C9B5
:00000001FF
```

FIELD:	SYMBOLS:	EXPLANATION:
1	:	Preamble.
2	10	Number of data bytes in field.
3	9290	Start address for first byte in data field.
4	00	Status byte signifies a field to process.
5	0E ...06	Data field 16 (10 hex) bytes long.
6	53	One-byte checksum, not used by our downline loading programs since the PCI requires a two-byte checksum.

NOTES:

The above explanation is given using symbols from the first line. You can see that the fourth line only has 06 hex bytes following it, and the status byte of the fifth line is 01 signifying absence of data on this line. Also note all the numbers are in hex format, as would be expected since this is called the intel hex format. Further information may be found in the 2500 A.D. manual for our Z80 assembler.

E. DOWNLINE LOADING:

This can be accomplished using one of two specially written C programs. The first program called \turboc\chagal.c comes with a help file and many options. This program is very user friendly and should be self-explanatory. The primary use of this program is for troubleshooting interfaces and assembled code. It has 9 options which are outlined below:

1. Load function (interactively).
   This is good for loading a function for the first time to see that the data is being correctly loaded and to observe the response from the PCI. Data is only sent upon approval.

2. Load function (automatically).
   A streamlined version of option 1. It performs error checking of response code and should notify you of a loading error. Useful for loading finished error free programs over a correctly operating interface.

3. Load block (interactively).
   Same as option 1 but uses a different downline load function, here the code is not given a label. Good for loading code of subroutines such as MCSAMP.

4. Load block (automatically).
   To option 3, what option 2 is to option 1.

5. Binary communications from keyboard.
   This option lets you communicate with the PCI using binary protocol. It is fairly automated and requires you only to input the number of bytes to follow in the message, the function label, and the function parameters. The header and check sum are configured for you. There is also an option to save the message created which is useful for option 6.

6. Binary communications from text file.
   This option is the same as option 5 except your command must be preconfigured and exist in a text file accessible by the system.

7. TOD communications.
   This option allows you to perform TOD communications which are detailed in chapter 4 of the PCI technical regerence manual. This type of communication is valuable for examining the PCI-3000 RAM and trouble-shooting programs. However, this option is also available in the basic program \basic\pci.bas. The option that resides in the basic program is quicker and easier to use.

8. Monitor communications port.
   This option was mainly for code debugging, but it is useful for observing the communications interface for five minute intervals. It provides a constant output of the device status followed by a colon followed by the character read. To interpret the device status convert it to hex and examine the bioscom function in the turboc reference manual.

9. Help option.
   Read this yourself.

0. Quit.

The second C program called \turboc\cezanne.c is used to load a collection of files and commands to the PCI. It was designed to be implemented in a reset mode. Once you have an operable system it might be desirable to load it all in one shot. This program will do that. It is, therefore, an excellent tool to set up your system to an operational level with the use of one command at power up. This program is not too friendly and as such must be explained here.

After calling the program you will be prompted for your source file path name. This will be a file similar to the one shown on the next page. This file must contain, in properly formatted fields, the following information.

option number<cr>
source pathname<cr>
first letter<cr>
second letter<cr>
option number<cr>
source pathname<cr>
first letter<cr>
second letter<cr>
.
.
.
0<cr>

The first and second letter fields are only applicable if the option number specified is a 2 (load function option). Otherwise the option number for the next pathname will directly precede the previous pathname. The only option numbers supported by this program are 2, 4, and 6. These are the load function, load block, and load binary protocol command options. The original source file must be terminated by an option number of zero. The path numbers specified within the source file would be the same as those you would use if you were using the friendly C program \turboc\chagall.c. Note the names chosen for the C programs are the names of artists I admire and have no relevance. Thus, it may be desirable to rename them. However, I am sick of trying to think of representative names, and I leave that option up to you.

Examining the source code on the next page we see that it is designed to:

1. Load tinitial.hex as a function, calling label XX.

2. Load tdefine.hex as a function, calling label YY.

3. Load pcionoff.hex as a function, calling label ZZ.

4. Load mcsamp.hex as a block (hence no calling label).

5. Load tabini.dat as a binary command. (This file contains the code necessary to call tinitial and intialize the sample table.)

6. Load defgrp1.dat as a binary command. (This file contains the code necessary to call tdefine and define the membership and sampling period of group two.)

7. Load defgrp2.dat as a binary command. (This file contains the code nexessary to call tdefine and define the membership and sampling period of group two.)

8. Load startsam.cat as a binary command. (This file contains the code necessary to call pcionoff and initialize the system to start calling mcsamp periodically.)

9. Quit.

```
2
a:\assem\tinitial.hex
X
X
2
A:\ASSEM\TDEFINE.HEX
Y
Y
2
A:\ASSEM\PCIONOFF.HEX
Z
Z
4
A:\ASSEM\MCSAMP.HEX
6
A:\BINCOM\TABINI.DAT
6
A:\BINCOM\DEFGRP1.DAT
6
A:\BINCOM\DEFGRP2.DAT
6
A:\BINCOM\STARTSAM.DAT
0
```

EXAMPLES OF BINARY COMMAND FILES:

TABINI.DAT:
24 42 31 01 04 12 99 AB 00

DEFGRP1.DAT:
24 42 31 01 1F 13 01 02 02 0C 00 00 07 00 08 00 0F 00 10 00 17 00
18 00 1F 00 20 00 27 00 28 00 2B 00 3A 01

DEFGRP2.DAT:
24 42 31 01 0B 13 02 05 00 02 00 00 22 00 3E 00

STARTSAM.DAT:
24 42 31 01 03 14 14 00

IV. IMPORTANT NOTES:

The PCI-3000 has several limitations. One of these is the default values given to functions with respect to the binary protocol. The binary protocol calling label is determined by the order in which the function is loaded into the system. At power up or reset the functions loaded into the system will have the following calling labels:

1. 12H for first function loaded into system.
2. 13H for second function loaded into system.
3. 14H for third function loaded into system.
4. 15H for fourth function loaded into system.
5. 16H for fifth function loaded into system.

It is important to remember that these calling labels are independent of the user defined ascii protocol calling label and only depend on the order in which the functions were loaded into system RAM. Hence, we suggest that upon reset these functions always be loaded into the system RAM in the same order. That order being as follows:

1. TINITIAL.HEX
2. TDEFINE.HEX
3. PCIONOFF.HEX

Hence, whenever calling sequences are given for these functions in binary protocol it is assuming that they were loaded in the above order. It is also important to note that the system cannot maintain more than five externally callable functions at one time.

VII. MORE ON LOADING FUNCTIONS:

When a function is loaded to the PCI by either of our C programs, the following takes place:

1. The PCI function Load Function is used to load the first 16 byte packet of code to the PCI.

2. The PCI function Load Block is used to load the remaining packets of code to the PCI.

When a block of code is loaded to the PCI, as is our MCSAMP subroutine, all of the code is loaded using the Load Block command.

Obviously, the only difference between loading code as a function and as a block is in how the first packet of code is transmitted to the PCI. What the load function command does is the following:

1. Set up the function, with its defined calling label and start address in the appropriate buffers in system RAM.*

2. Load the packet of data to the appropriate locations in system ram.

The Load Block command only performs the second of the operations performed by the Load Function command. What that means is that in order to call a function loaded entirely by the Load Block command its calling label and start address must be loaded by an entirely different means. This is why we have created PCIONOFF. PCIONOFF will, on alternate calls, setup the system RAM to call MCSAMP periodically or initialize the system RAM to stop the periodic call of PCIONOFF.

---

* This is somewhat equivalent to placing a function header in an interrupt table. However, I don't believe that the PCI user defined functions are called on interrupt. Rather, the function is called when the system sees that this function flag is set during a pass through the scheduler. For example, if by calling a function, flag number 17 is set, and at the same time the BFLAG pointer is at position 6, the BFLAG pointer won't jump on interrupt to flag 17 to call the desired function. Instead, it seems as if it continues through the BFLAG buffer sequentially and will deal with the flag at position 17 in due course.

VIII. ASCII VS BINARY PROTOCOL AND BUFFER LIMITATIONS:

In the PCI-3000 system we have the choice of two protocols, ascii and binary, each of which have there own advantages.

Ascii protocol is friendlier and easy to use, especially using the basic communication program PCI.BAS. Its primary use would be in calling functions, especially those using a small number of parameters. However, due to the fact that it uses a larger number of bytes to convey the same amount of information it is inherently less efficient.

Binary protocol will generally be the protocol of choice for the following reasons:

1. More information can be transmitted with the same I/O buffer limitations than for ascii protocol.

2. It is faster.

3. It is easier to use when down-loading code or commands from a disk file.

The buffer limitations play a large role in the choice of protocols. Since both the PCI-3000 input and output buffers are only 128 bytes long, efficiency of space is a concern. For instance, when a complete complement of data is being transmitted this corresponds to the following number of bytes:

Header:	3 bytes
# of bytes to follow:	1 byte
group number:	1 byte
32 analog channels 1 word each:	64 bytes
12 digital channels 1 byte each:	12 bytes
1 word checksum:	2 bytes
Total number of bytes:	83 bytes

Formatting these in ascii protocol would exceed the buffer limitations. Also, using ascii protocol several calls to TDEFINE may be necessary to define one group. Otherwise, buffer limitations may be exceeded.

It is possible to expand the length of the input and output buffers so exceeding the buffer length would not be a problem with either protocol. However, it was felt that the less we changed the default values of the system the less chance we had of running into unforeseeable errors. Chapter 2 of the PCI technical reference manual provides a good explanation of the system buffers and can be used as a further reference.

IX. PROBLEMS ENCOUNTERED (& SOLUTIONS?):

When loading data using options one through four, sometimes the last field will give a "data loaded incorrectly to PCI" error, or enter an infinite loop. If the last data field only has one byte, the C function cruncher might not recognize this as the last field. Check the cruncher.hex file, and, if this is the case, figure out why. (I ran out of time. It shouldn't be too hard.) Otherwise, add a meaningless command to your assembly file so the last field contains more than one byte and try again.

When sampling data the host MassComp computer gives an internal fatal error and times out. This happened randomly, and it was suspected that the IEEE-488 board had a bug. We tried a new board and this did not happen, although we did not do enough tests to be sure it was the old board. I suggest that you reload the dacp and start processing the transmitted data again. This type of crash should not affect the operation of the PCI.

The MassComp enters the SRQ routine continuously with a status byte of FFFFFFC0. This happened for awhile, but only when the bus analyzer was not hooked up. There are several possibilities, my favorite being that the IEEE-488 board was at fault. It has not happened since we started reloading the dacp after crashes, and I changed my code so that it would not wait on the output buffer. Now, if the output buffer is not clear, the function will terminate.

On option five the response code does not seem to agree with protocol. In particular, the function executed properly, but byte 4F hex is not present, and the response looks something like 21 00 01 00. I do not really know what this is. I think maybe my get_response_rs232 function is messing up. The same code saved to a text file and used in option 6 executes perfectly. With this response the function seems to still execute fine, but the response code seems out of order. I suggest you do not worry about it if it works. If you need a correct response code use option 6 with a file created in option 5.

Two status lights on PCI light up and stay lit, and the travelling train of pulses stops. This happened frequently when the PCI used to wait on a buffer and the MASSCOMP had crashed. It has not happened since I changed the code to not wait on the buffer but rather terminate if the buffer is not clear. This seems to indicate that this happens only when a function in the PCI hits the timer to stop counting and waits on a buffer. If it happens again make sure the PCI is being serviced, and, if this does not help, try control C'ing your program. As a last resort physically reset the system. I think this problem is gone now that the code has been changed, but what has happened before can always find a way to happen again. Note, when the system used to hang up like this before, it was impossible to talk to it.

Status lights on PCI blink incoherently, kind of faint and random blinks. This used to happen if I would interact with the PCI a lot over the RS232, especially if it was running MCSAMP periodically. However, this never seemed to affect the operation of the system. The only effect it seemed to have was on the aesthetic quality of the display. If it happens, and it bothers you, reset the system.

A function hangs up when calling TDEFINE or MCSAMP in ascii.

You may have exceeded a buffer limitation. Try again in binary protocol, and make sure that the message is less than 128 bytes. The message should never exceed 128 bytes unless things have been expanded since this manual was written.

When loading code you consistently get an error.

This may happen if you try to define more functions than the system firmware can accommodate or if you are redefining functions that already exist. I think the firmware can only accommodate 5 or fewer functions. Reset the system and try again.

The MassComp is terminating early on some streams in binary protocol.

Make sure that there is no termination character set for binary protocol and that the termination character for ascii protocol is a carriage return as opposed to the default value of a line feed.

The PCI samples group two at the incorrect interval.

Make sure that your definition for the group two sampling rate is the low byte in a two byte binary word < 256 in ascii. Also make sure you are using low byte, high byte format in group two sampling period statement.

You get an error 130 decimal when you call a function. This means that your function is no longer defined in the system firmware. The system has probably timed out and reset. Reload your code. For a further definition of error codes returned by the firmware see Appendix A-1 of the PCI-3000 Technical reference manual.

X. STATS AND STUFF:

It takes the PCI between 80 & 96 ms to sample a full complement of data (32 analog channels and 12 digital channels).

The period of the group one sampling rate can be defined as low as 16 ms. However the actual period at which the sampling rate will occur is limited by the controlling computer on the IEEE-488 bus and its ability to service data out of the PCI's buffer. Also the PCI has had a tendency to time out when the MassComp crashes if its fastest sampling rate (group 1) is less than the time it takes the function to execute. Hence, it seems reasonable to place an absolute limit of 200 ms (about 2 * max run time of MCSAMP) on the group 1 sampling rate. A more practical limit is 0.5 s.

The PCI was tested for over 40 hours sampling a full complement of data with a sampling period of 0.5 seconds. It did not crash once during this time and was running as smooth as could be when the test was terminated. The MassComp did have a few crashes however and we attribute these to "that bad IEEE-488 board".

All analog channels seem to be reading the input data perfectly for all values of the gain (1, 10, 100, 1000).

If the group two sampling period is set to 0, group two will be the only group sampled. The group two sampling period will then be equivalent to the sampling period defined for group one. This means that group one will be shut off and group two will be sampled at the group one rate.

To shut off the group two sampling just define group two with the same membership as group one, at any sampling period. In actuality group two will be still sampled and the group identifier byte will be at the head of the data at the defined intervals, but that will be the only difference.

Sampling group one at a sampling period of zero will succeed in doing nothing since no sampling will take place.

APPENDIX A:

The \turboc\chagall.c program, and function documentation.

EXPLANATION OF C PROGRAM SUBROUTINES:

Here we will try to give a brief outline of the functions written for use with our C programs:

int talk_to_partner (fname, letters, type_data)

```
int letters[2];
char fname[25];
char type_data;
```

This is a function which, when called, must have the type_data variable specified and pointers to fname and letters.

The purpose of this function is to configure the system for either a load block option or a load function option as specified by type_data. For each call it will prompt the user for a filename and return this to the calling statement. If type_data[0] = 'y' the load function options are specified and it will prompt the user for an ascii calling label. This two letter calling label will be converted to Hex representation by ascii_to_hex and returned to the calling functions by letters[2].

The function inquires to see if you wish to reenter the data or return to the top of the program. The function returns an integer corresponding to one of the following three options:

1. Reenter data. Return = 1
2. Quit this option go back to start. Return = 2
3. Everything is ok proceed as normal. Return = 0

int cruncher(source_file)

char source_file[25];

This is the file passed by talk_to_partner. It must contain the intel hex representation of your assembly code.

The purpose of this function is to index through your source file and extract the significant bits of code. It then rewrites this code into a temporary file called cruncher.hex. This file format is displayed on the next page. The actual file is that created from loading our TINITIAL function.

Examining the file we see that it has four blocks of data consisting of five similar fields. Each block of data should correspond to one line in the original intel hex file. The five fields in each block have the following meaning:

Field 1. This just consists of a colon and a number which is the hex representation of the number of bytes in field 5. The colon is very important because it is used by the file pointer as a yard stick.

Field 2. This is just the desired (hex) address to which we wish to load the first byte in field number five.

Field 3. This is just a status byte telling you about field five. A 01 specifies that field five contains an end of file, while a 00 specifies that there is data to be read in field five.

Field 4. This is just a decimal number computed by converting the number in field one to decimal and adding one. This number will be used for program control.

Field 5. This contains up to 16 bytes of data to be loaded as code to the PCI-3000 and a one byte check sum.

```
:10
9290
00
17
0E,01,CD,55,00,07,38,29,DD,21,FF,8F,DD,73,00,06,53
:10
92A0
00
17
60,DD,36,01,00,DD,23,05,20,F7,0E,01,1E,4F,CD,64,81
:10
92B0
00
17
00,07,38,0D,21,FF,8F,5E,0E,01,CD,64,00,07,38,01,D5
:06
92C0
00
 7
C9,0F,CD,85,00,C9,B5
```

NOTE:

The last block in the file will always have either a status byte of 01 or less than 16 bytes of data.

If there is an error in running the program this function is usually the one at fault. This file provides a good means from which one can begin to troubleshoot.

The return value is the number of data fields in this file. This is an important program control variable and if computed incorrectly could cause your program to infinitely loop. This will be most noticable when using an interactive option and displaying the last field.

```
int display_data(x, nb, type_data)

int x[128];     /* data field to display */
int nb;         /* size of x: x[nb]      */
char type_data[3];  /* specifies a load block or function */
```

This function has passed to it the data array containing the parameters to be loaded as code to the PCI. It merely creates a nice display of these parameters which the user can view before deciding whether or not he or she wishes to send the data to the PCI. This function is only called if the user chooses options one or three. Also, it also has not been configured to return a value or alter any passed parameters.

```
int com_control(function_codes, type_load)

int function_codes[128];
char type_load[3];
```

This function is used much like a traffic light in that it controls the calling and response messages of the communications functions, send_function_rs232 (function_codes) and get_response_rs232 (response_pci). Due to the crude nature of the Turbo C communication function, BIOSCOM, it is necessary to carefully control the sequence in which the communication programs are called. This is done in order to insure the response is not missed. This function can be configured for both the interactive and automatic features as specified by type_load. In the automatic mode it will check the response code in order to insure proper transmission.

```
int ascii_to_hex(character)

char character;
```

This function returns the upper case hexadecimal representation of **character**, however it does not alter character.

int high_byte(word)

int word;

This function returns the high byte of a two byte word to the calling statement.

int low_byte(word)

int word;

This function returns the low byte of a two byte word to the calling statement.

int assemble_data(field_to_process, letterss, data_array, type_data)

```
int field_to_process;     /* specifies field in cruncher to
process
int letterss[2];          /* hex representation of ascii calling
code
int data_array[128];      /* array for assembled data
int type_load;            /* specifies a load block or load
function
```

This function assembles a block of data located in the file cruncher.hex. The field it reads is determined by **field_to_process**. If the type_data function specifies a load function type option, it uses the letters contained in **letterss** in the assembled data field. In assembling the function it computes the correct header for calling either load block or load function in binary protocol. It then computes the ending address for the packet of data. This address is computed using the starting address specified in cruncher.hex. The "number of bytes to follow" byte and the two byte check sum are also computed and arranged in the proper format (low byte, high byte). All of these bytes are arranged in the array **data_array**. The order in which the bytes are placed in **data_array** corresponds exactly to the order in which the bytes will eventually be transmitted. Also, there are no extraneous bytes in **data_array**. The calling statement to this function should have a pointer to an array such that the loaded

data_array will be passed back through the call statement to the array pointed to by the pointer. Upon proper execution this function will return the length of **data_array.**

int pack_package(box)

int box[128]; /* array into which data bytes are to be loaded */

This function operates much in the same way as assemble data in that it passes an array containing a complete set of bytes suitable for transmission. However, this array does little assembling since the source of the bytes in the array is a properly formatted text file. The data in the disk file must be completely arranged as per binary protocol specifications, and all this function is responsible for is:

1. Prompting for the file name of the source file.

2. Making a simple check to see if the file is formatted properly (sees if the first byte is correct).

3. Reading the data directly into the array **box.**

4. Returning an integer specifying that the function executed either correctly or incorrectly.

int send_function_rs232(package)

int package[128]; /* array of bytes to be transmitted */

This function just uses the Turbo C BIOSCOM function to configure serial port com 1 for transmission at 1200 baud with 8 data bits. It then will transmit byte by byte the elements in **package** through this port over the RS232 cable. The function returns, as an integer, the initial status of the port.

int get_response_rs232(response_pci)

int response_pci[20]; /* PCI response code array */

This function is designed to receive a properly formatted response code from the PCI. It assumes a

properly formatted communications port and will wait for a short time for the header byte of the response code. Upon reception of this one-byte preamble the function exits its wait loop and gets the rest of the response message. If the function does not receive the header byte after a short period of time, it will return the error code of 999, otherwise it will return an integer specifying the length of the response array.

It is important to remember that the com port does not accept buffered input and that this function cannot be called on interrupt. Hence, if this function is not called in time to receive the header byte, it will return an error condition. Alternately, if the accepted code is not formatted correctly, it has the wrong header byte, or the third byte is not the 'number of bytes left to read' byte, the program will give an error code or possibly loop for the incorrect amount of time when it is reading in the rest of the response code.

int key_com()

This function is designed to facilitate binary protocol keyboard communications with the PCI. It is responsible for internally configuring the message array. It will place the preamble at the head of the array and query the user for the 'number of bytes to follow' byte as well as the remaining bytes of code. Hence, the user must input the following:

1. The "number of bytes to follow" byte.

2. Binary protocol calling label.

3. Parameter bytes as needed by the particular function.

NOTE:

The "number of bytes to follow" byte is equal to the number of bytes in 2 and 3 above plus two for the two-byte checksum.

After creating a function this routine will print it to the terminal as it would be sent to the PCI. It will then ask you if you want to send it as well as giving you the option of saving it in a text file for use with option 6.

int tod_communications()

This function is used to facilitate option 7 in that it will send the necessary control signals to the PCI to facilitate terminal on-line debugging. However, due to the lack of sophistication of the C communication functions at our disposal, one is better off using the program \BAS\PCI.BAS. TOD communications from this program are easier and quicker. For a good explanation of TOD communications see chapter 4 in the PCI-3000 technical reference manual.

int monitor_com_port()

This is a fairly barbaric function in that it goes and looks at the communications port at about five minute intervals. It can look in either of two modes.

Active: It will continuously output port status, followed by a colon, followed by the character read for the five minute interval.

Other: It will only output the port status, and character read when data is coming into the port.

NOTE:

Information received is output as ascii characters and may have to be converted to hex or decimal for proper interpretation. Also, once the port is being observed there is no way to exit the function until the five minute interval is up, unless of course you wish to be so destructive as to crash this program.

int get_help()

This function just gets the help file, help_file.txt, and outputs it page by page to the screen. Note that help file must reside in the same directory as does the calling program.

APPENDIX B

This appendix is dedicated to the detailing of the Z80 assembly code function TIMEOUT.HEX. This functions use is optional but it may be helpful.

MORE ON THE WATCH DOG TIMER & AUXILIARY TIMEOUT FUNCTION:

The reason that the watch dog timer has a timeout period of 0.75s is because its initial value is equal to 30H. This value is reloaded to the counter from the TIMOUT register on each pass through the scheduler. The value in the counter is then decremented every 16ms. The system reset pulse is sent if this counter is decremented to zero. Hence, if we want to extend the timeout period of the PCI or, rather, create a timeout, we can adjust the value which is reloaded to the watch dog timer counter. For example, loading the TIMOUT register with the following bytes should have the following results:

1. 30H This is the default value, timeout about 0.75 s.

2. 00H This disables the watch dog timer.

3. FFH This enables the timer to its greatest value 4.08 s.

4. 01H This should cause the watch dog timer to time out since it seems to take more than 16 ms for one pass through the scheduler.

On the following page we have written a program to pass a user timeout period to the PCI. A one byte parameter is passed specifying the byte you wish loaded in the TIMOUT register. This will correspond to the timeout value divided by 16 ms. It is suggested that this function be loaded after the main four functions or loaded upon need using cezanne.c to load and execute this function at once. This would insure a binary calling label of 15.

NOTE:

There is a possibility that by loading too small of a timeout out byte the PCI will time out before the timeout function has been completed. Hence, if you are passing 01H as a parameter in this function you might not receive a response back since the firmware will reset quite quickly.

```
;*****************************************
;*****************************************
;     QQ(timeout byte).  This is a quick
;function to reload the timeout to some
;desired value.  To reset the firmware a
;byte of 01H should suffice, while to
;disable the W.D.T. one needs to load 00H
;to the timeout byte.
;*****************************************

;******* PROGRAM CODE & DOCUMENTATION ****
;*****************************************

;******* GET BYTE TO LOAD TO TIMOUT ******

                LD C, 01H
                CALL GETPAR
                RLCA
                JR C, ERROR
;     GET BYTE TO LOAD INTO TIMOUT REGISTER

;******* RELOAD TIMOUT REGISTER **********

                LD HL, TIMOUT
                LD (HL), E
;LOAD THE BYTE READ IN BY GETPAR TO TIMOUT

;******* PASS EXECUTION VERIFICATION BYTE

                LD C, 01H
                LD E, 4FH
                CALL PUTPAR
                RLCA
                JR C, ERROR

;******* FUNCTION TERMINATION ************

                RET

;******* ERROR CHECKING ******************

ERROR                RRCA
                CALL RETERR
                RET

;******* FIRMWARE EQUATES ****************

GETPAR EQUAL 0055H
PUTPAR EQUAL 0064H
RETERR EQUAL 006DH
TIMOUT EQUAL 4061H
```

TIMEOUT.HEX

```
:109300000E01CD550007380F216140730E011E4F2D
:0C931000CD6400073801C90FCD6D00C905
:00000001FF
```

APPENDIX C:

Here we provide a directory listing of the master disk as well as for my account on the MASSCOMP (bstevens).

MSDOS FLOPPY:

ASSEM SUBDIRECTORY:	TURBOC SUBDIRECTORY
TINITIAL.HEX	CHAGALL.C
TDEFINE.HEX	CEZANNE.C
PCIONOFF.HEX	HELPFILE.TXT
MCSAMP.HEX	MENU.TXT
TIMEOUT.HEX	CRUNCHER.HEX
TLIGHT.COD	MANUAL.PCI
	MANUAL.APP

BINCOM SUBDIRECTORY:	BASIC SUBDIRECTORY
TABINI.DAT	PCI.BAS
DEFGRP1.DAT	BASIC.COM
DEFGRP2.DAT	BASICA.EXE
STARTSAM.DAT	GWBASIC.EXE

MAIN DIRECTORY: COMMAND.COM

ON THE MASSCOMP DIRECTORY, /u/bstevens, RELEVANT FILES.

chagall.c	helpfile.txt	menu.txt
gpib3.c	gpib3.exe	
gpib4.c	gpib4.exe	
gpib5.c	gpib5.exe	
gpib6.c	gpib6.exe	

ADDITIONAL NOTES:

As of now there has been no time to check the digital channels. Since the analog channels work fine we suspect that the digital ones will also. If there turns out to be any problems they will be in one of two places.

1. The wiring of the distribution panel.

2. The setup of the TIODRV function in sections 3 & 4 of MCSAMP.

Included on the disk is a function MCSAMP1. This is merely a variation of MCSAMP such that it prints out the analog channel being read before it prints out the actual data. This function should be used in conjunction with gpib5.c on the masscomp in my directory.

gpib4.c should be used for binary protocols.

gpib3.c should be used for ascii protocols.

If you want to alter TDEFINE such that the analog gain can be reentered on seperate calls of the function, you must redefine the code such that the bits corresponding to the analog gain (0 & 1) are reset before the new value of gain is or'ed to those locations.