ARTISAN®
TECHNOLOGY GROUP

Your **definitive** source for quality pre-owned equipment.

Artisan Technology Group
(217) 352-9330 | sales@artisantg.com | artisantg.com

Full-service, independent repair center
with experienced engineers and technicians on staff.

We buy your excess, underutilized, and idle equipment
along with credit for buybacks and trade-ins.

Custom engineering
so your equipment works exactly as you specify.

- Critical and expedited services
- In stock / Ready-to-ship
- Leasing / Rentals / Demos
- ITAR-certified secure asset solutions

Expert team | Trust guarantee | 100% satisfaction

All trademarks, brand names, and brands appearing herein are the property of their respective owners.

Find the *Measurement Computing / ComputerBoards CIO-DAS16/JR* at our website: *Click HERE*

CIO-AD16JR-AT Programmers Guide
8254 Counter Descriptions

A key to programming the CIO-AD16JR-AT is understanding the additional 8254 which has ben added. Specifically, understanding how each of these counters is used in generating interrupts and tracking trigger points is critical.

There are two 8254's on the CIO-AD16JR-AT. The first 8254 is located at base address + 9 through base address + 12 and is identical in function to that on the CIO-AD16 and MetraByte's DAS-16. It is assumed that the functions of this 8254 is fully understood and are not included in this discussion. The second 8254 is located at base address + 16 through base address + 19. For clarification it is suggested that the CIO-AD16JR-AT block diagram and the counter detail diagram be reviewed while reading this section.

In all case counters 0 and 1 are chained together to form a single 32 bit counter. Counter 2 always operates independently as a 16 bit counter. There are two general types of high speed collection, normal and pre/post trigger. Normal collection is when the data collection begins with the occurrence of a specified trigger condition. Pre/post trigger collection is very similar to normal collection except that a predefined number of samples is stored prior to the occurrence of the trigger condition. The counters play different roles depending on the type of data collection that is being performed.

Normal Collection

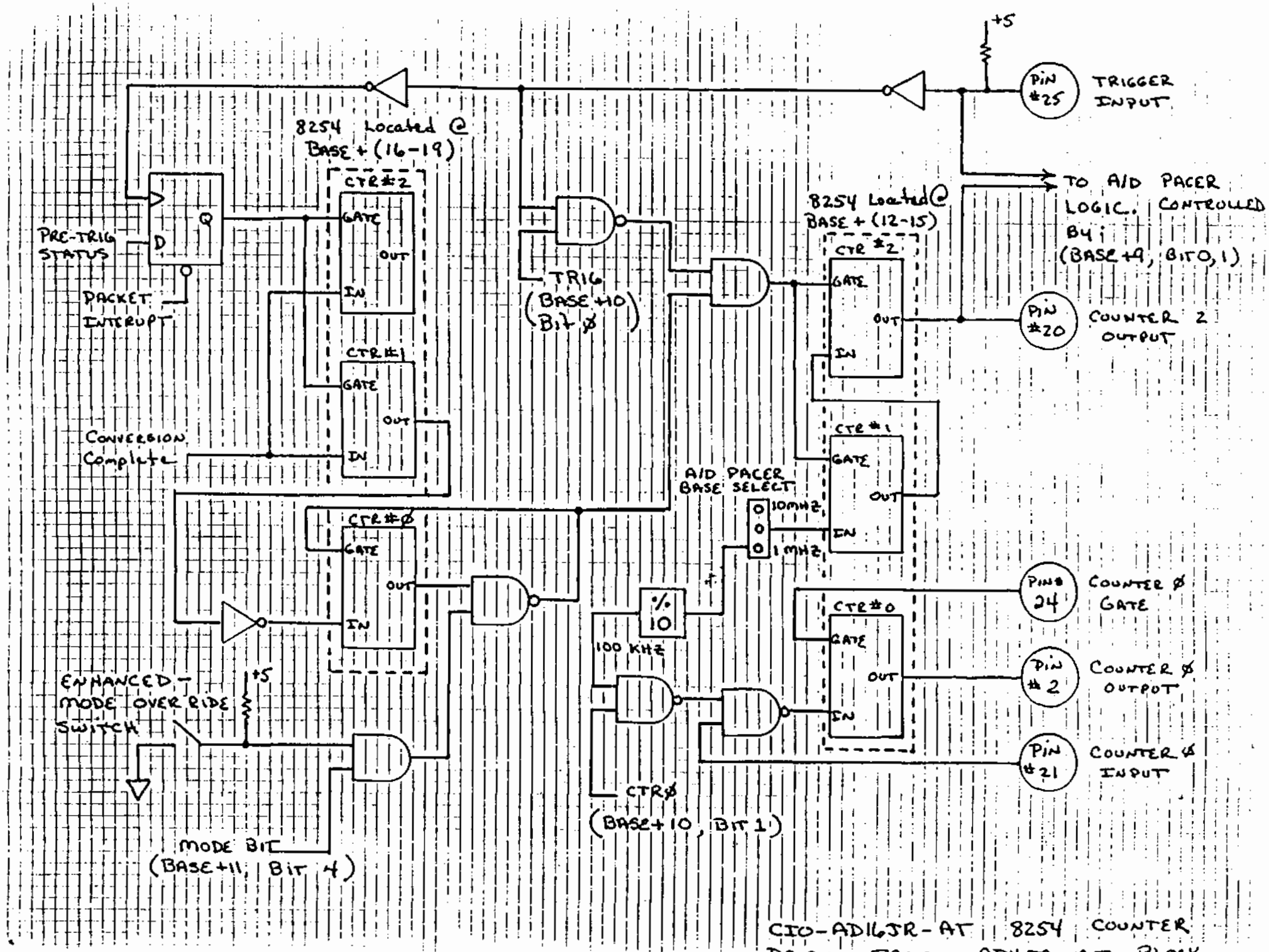
CTR 0 & 1 — Counters 0 and 1 act as a 32 bit total counter with counter 1 as the LSB and counter 0 as the MSB. The FIFO will generate an interrupt for each set of 512 samples which are taken and require servicing. The total counter is used to generate the final interrupt which may or may not be on a 512 sample boundary. In this manner if the remaining number of samples is not 512 the FIFO will still be serviced and the operation completed.

CTR 2 — In normal mode counter 2 has no function.

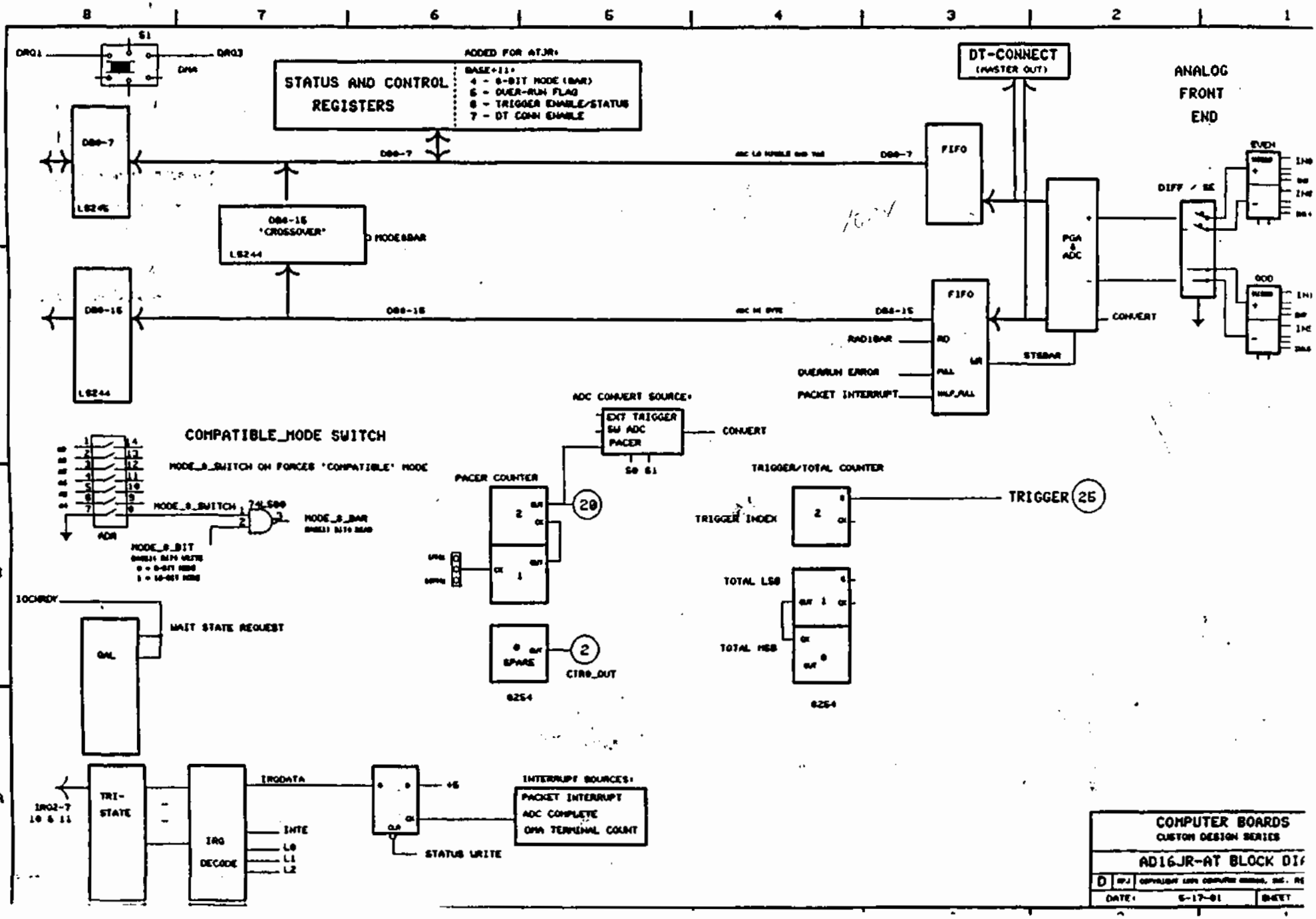
Pre/Post Trigger Collection

CTR 0 & 1 — Counter 0 and 1 act as a 32 bit post-trigger counter with counter 1 as the LSB and counter 0 as the MSB. This is different from the total count described in normal collection in that it is only the number of samples to be taken after the trigger condition. The total count in this case includes both the pre and post trigger portions of the collection. The pre/post trigger counter does perform the same function as the total counter of the normal collection as it allows for serving of the remainder data points at the end of data collection.

CTR 2 — Counter 2 acts as a 16 bit trigger index counter. The counter is initially loaded with the value 512. Upon the trigger condition counter 2 begins to count down until a packet service request is issued by the FIFO. Upon servicing this packet of information it is important to read the contents of counter 2. The value in counter 2 will allow you to determine how many of the points that you have just taken in are pre-trigger and how many are post-trigger.



CIO-AD16JR-AT | 8254 COUNTER DETAIL FROM AD16JR-AT BLOCK


8
7
6
5
4
3
2
1
D
C
B
A
S1
DRQ1
DRQ3
DMA
STATUS AND CONTROL REGISTERS
ADDED FOR ATJR+
BASE+11+
4 - 8-BIT MODE (BAR)
6 - OVER-RUN FLAG
8 - TRIGGER ENABLE/STATUS
7 - DT CONN ENABLE
DT-CONNECT
(MASTER OUT)
ANALOG FRONT END
D00-7
LS245
D00-15 'CROSSOVER'
LS244
MODE8BAR
D00-15
LS244
D00-7
D00-15
FIFO
FIFO
PGA & ADC
DIFF / SE
EVEN
ODD
CONVERT
STSBAR
RADIBAR
RD
OVERRUN ERROR
FULL
PACKET INTERRUPT
HALF_FULL
COMPATIBLE_MODE SWITCH
MODE_8_SWITCH ON FORCES 'COMPATIBLE' MODE
MODE_8_SWITCH
74LS00
MODE_8_BAR
ADR
MODE_8_BIT
0 = 8-BIT MODE
1 = 16-BIT MODE
ADC CONVERT SOURCE+
EXT TRIGGER
SW ADC
PACER
CONVERT
S0 S1
PACER COUNTER
20
SPARE
2
CTR0_OUT
8254
TRIGGER/TOTAL COUNTER
TRIGGER INDEX
TRIGGER 25
TOTAL LSB
TOTAL MSB
8254
IOCHRDY
WAIT STATE REQUEST
GAL
IRQ2-7
10 & 11
TRI-STATE
IRQ DECODE
IRQDATA
INTE
L0
L1
L2
+5
STATUS WRITE
INTERRUPT SOURCES+
PACKET INTERRUPT
ADC COMPLETE
DMA TERMINAL COUNT
COMPUTER BOARDS
CUSTOM DESIGN SERIES
AD16JR-AT BLOCK DI
D
DATE+ 5-17-91
SHEET

Programmer's Guide to

CIO-AD16AT-JR

By Computer Boards
44 Wood Ave
Mansfield, MA 02048
(508) 261-1123

Ver: 10/24/91

Table Of Contents

1. Introduction 1

2. Contents of the Disk 1

3. Overview of CIO-AD16JR-AT 1

4. Details of Operation 4

4.1 Introduction to SAMPLE.ASM 4

4.2 Constants 5

4.3 Macros 5

4.4 Public Routines 5
4.4.1 RepInFore() 6
4.4.2 RepInBack() 6
4.4.3 MovRepInData() 8
4.4.4 PreTrigFore() 9
4.4.5 PreTrigBack() 9
4.4.6 MovePreTrigData() 12
4.4.7 GetStatus() 13
4.4.8 StopIO() 13
4.4.9 AllocBuffer() 13
4.4.10 FreeBuffer() 14

4.5 Private Routines 14
4.5.1 repinint() 14
4.5.2 pretrigint() 15
4.5.3 waitidle() 16
4.5.4 startpacer() 16
4.5.5 stoppacer() 17
4.5.6 ctrset() 17
4.5.7 muxset() 17
4.5.8 div_qd() 17

5. The Test Program TESTEX.C 18

5.1 Introduction to TESTEX.C 18

5.2 Interface to EXAMPLE.ASM 18

5.2 Building TESTEX.EXE 18

5.3 Running TESTEX.EXE 18

5.4 Modifying TESTEX.C 19

1. Introduction

This guide along with the enclosed disk is intended to assist experienced programmer's in writing software that can take advantages of the new and unique features of the Computer Board's CIO-AD16JR-AT board.

This guide should be used in combination with the CIO-AD16JR-AT User's Manual which describes the details of the hardware. This guide assumes that you are already familiar with A/D boards in general.

The CIO-AD16JR-AT is a superset of Computer Board's other AD16 series A/D boards (CIO-AD16, CIO-AD16F, CIO-AD16Jr). It can operate in two different modes - COMPATIBLE and ENHANCED. In COMPATIBLE mode it behaves exactly like a standard AD16 series board. In ENHANCED mode the board behaves quite differently.

This guide describes how to use the ENHANCED mode features. To program the board in the enhanced mode refer to the standard AD16 series manuals.

2. Contents of the Disk

The enclosed disk contains the following items:

* EXAMPLE.ASM - A set of assembly language routines that demonstrate how to use the new features of the CIO-AD16JR-AT board.

* TESTEX.C - A C program that calls the routines in example.asm. This program can be used to test the operation of the board.

* TESTEX.MAK - A make file that assembles EXAMPLE.ASM, compiles TESTEX.C and links them to generate TESTEX.EXE.

* TESTEX.EXE - An executable version of the test program.

* JRGUIDE.DOC - An ASCII version of this document.

3. Overview of CIO-AD16JR-AT

The CIO-AD16JR-AT is actually two different boards in one. When run in COMPATIBLE mode it behaves exactly like an older AD16 series board. When switched to ENHANCED mode the board is programmed differently and supports the following

features:

* REP INS Data Transfer

* 16 Bit Data Transfer

* On Board Packet Buffering

* Hardware Support For Pre-Triggering

* DT Connect Interface

* Large (>64K) Data Transfer At Full Speed.

The easiest way to understand the Jr-AT board operation is to compare it to the operation of conventional A/D boards. Most existing A/D boards are designed to do an A/D conversion and then send that single A/D sample back to the PC in two separate 8 bit I/O operations. The data is usually transferred in one of the following three ways - programmed I/O, Interrupt I/O, DMA Transfer.

Programmed I/O is the simplest. The foreground program polls the board to see when it has completed a conversion and then reads the data with a two I/O port IN instruction.

Interrupt I/O allows sampling to go on in the background. When the board completes an A/D conversion it generates a hardware interrupt. An interrupt handler transfers the sample using two 8 bit I/O opertions.

DMA Transfer uses the PC's DMA hardware to transfer the data from the board to a buffer using two 8 bit I/O operations per sample. The advantage of this is that the CPU is free (for the most part) while the data is being transferred and the DMA controller can move the data faster then an interrupt service routine can. One of the disadvantages is that the PC's standard DMA controller is an 8 bit device so it can never move more then 64 K of data at a whack and it requires that the transfer never cross a page boundary.

The JR-AT board is a completely different design that allows much higher transfer speeds, with no limitations on starting addresses or size of transfers. It is able to do this by taking advantage of two features of the IBM AT design - 16 bit I/O and the 80286 REP INS instruction. Both of these features were not present on the original IBM PC which is why the Jr-AT board requires a 80286 or 80386 machine.

16 Bit Transfer - Starting with the IBM AT, the bus could be run with either 8 or 16 bit I/O operations. Almost all existing A/D boards use 8 bit transfer so that they remain compatible with the original PC. The Jr-AT gives up compatability with the original PC in return for doubling of

the I/O spped.

REP INS Instruction - The 80286 added a few new instructions that did not exist in the 8088 instruction set. Among these is the INS instruction. This instruction is very similiar to the other "string" instructions (MOVS, LODS, STOS etc). It reads from a port specified by the DX register and stores the data at the address pointed to be ES:DI and then increments DI. The most important feature of this instruction is that it can be used with the REP prefix like all other string instructions. With the combined REP INS string instruction you specify the number of times you want to read the port in CX and the REP INS will fill a buffer with successive port I/O reads.

When the REP INS is combined with 16 bit transfers (REP INSW) it is possible to move data at a much higher rate then is possible with the PC's DMA controller. The maximum transfer speed is dependedent on the speed of the CPU but in all cases it is faster then DMA without any of the restrictions and complications associated with the PC's DMA controller.

In order to be able to use this method of data transfer the A/D needs a very different architecture. The most significant difference from a programmer's point of view is that there is a 1024 word First-In-First-Out (FIFO) buffer and three additional counters.

When you initialize the A/D you must load the counters with the total number of samples that you want. Once you start the conversions the board takes each A/D Sample and puts it in the FIFO. The board will then generate an interrupt under two different conditions - whenever the FIFO is half full (512 samples) and whenever the total number of samples have been converted (counter goes to zero).

The data transfer is all done by an interrupt service routine which transfers the data from the board with the REP INSW instruction. The data from each 512 sample "packet" is moved with the a single REP INSW. The last "packet" may have some smaller number of samples since there is no guarantee that the total number of samples requested is a multiple of 512. SO the interrupt service routine must keep track of how many samples it has moved so that it knows when it is on the last packet.

Hardware Pre-Triggering - The board also includes hardware support for full speed "pre-trigger" operation. Pre-triggering is a technique where the user specifies how many A/D samples they want to collect both before and after a trigger occurs. The Jr-AT provides the necessary features to be able to implement pre-triggerring at the maximum sampling rate (333 kHZ).

The key to pre-triggering is the way the Jr-AT uses the counters. You initialize one of the counters with the post-trigger count and the pre-trigger counter with the size of the packet. Once you start the board your interrupt service routine transfers packets into a circular buffer that is at least as big as the the total count (pre-trigger + post trigger). When a trigger occurs the hardware on the board set a status bit and begins decrementing the pre-trigger counter until the end of the packet. It also begins decrementing the post-trigger counter. On each interrupt you check the status bit and if a trigger ocurred during that packet you read the pre-trigger counter. The count it returns will be the index within the packet to the sample that the trigger occurred on. You must save this index and then continue servicing the post-trigger packets.

When the post-trigger count reaches zero you stop sampling. At that point you will have a circular buffer full of data. The data is in order but it does not start at the beginning of the buffer. It starts somewhere in the middle and wraps around from the end to the beginning. Using the pre-trigger index that you read from the counter you can figure out how to spin the data in the buffer so that it starts at the beginning.

4. Details of Operation

4.1 Introduction to SAMPLE.ASM

The SAMPLE.ASM files contains examples of all the routines that you need to use the features of the Jr-AT EXTENDED mode. The code is written in 80286 assembly language and was assembled with Ver 5.1 of Microsoft's Macro Assembler.

This code is based on Computer Board's BASIC language driver. Most of the public routines are very similiar in function to specific "Modes" in the BASIC driver. The equivalent Mode is specified in the section that describes each of these routines. You can refer to the section of the CIO-AD16-AT User Manual that describes those modes for more detailed information about the arguments and restrictions etc.

This code may be used freely and modified to suit your needs. In general though it is intended more as an example of how to program the board rather then as source code that you just plug into your application. If you do wish to use it in your application it should be fairly easy to interface it to whatever language you are writing in. The program TESTEX.C program that is included with this package can call these routines but it does so by setting up all of the arguments in global variables. If you want to modify this code to use the standard calling conventions of the language that you are using you can write a simple layer between SAMPLE.ASM and the host language.

The routines in this file are broken down (rather arbitrarily) into public and private. The private routines are ones that are called by other routines within SAMPLE.ASM and that will your application program will probably not be required to call.

Note that the source code begins with a few Microsoft MASM directives.

.model small,c - This directive specifies the details of the calling interface to ecah routine. Since the TESTEX.C program was written in C the SAMPLE.ASM file specifies c in the .model directive. This could be changed to use another languages if you desire. If you leave it set to C note that the assembler prepends all symbols with an underscore character (_).

.286 - This must be specified in order to use the REP INSW instruction.

4.2 Constants

The file begins with many different groups of symbolic constants defined.

Register Definitions - These define the offsets from the base address to each of the Jr-AT's registers. These are used as arguments to the HIN and HOUT macros.

Hardware Bit Definitions - These define individual bits within some of thd Jr-At's registers. Only those bits that this code needs are defined here.

Operation Modes, TriggerModes, Cycle Modes define the legal values that can be passed for those arguments.

4.3 Macros

The HOUT, HIN, and WORDOUT macros are used for all I/O with the Jr-AT board. These macros take care of converting the symbolic names for the registers to physical port addresses.

4.4 Public Routines

The public routines are those that routines that an application program may need to call. Most of these routines are based on one of the "Modes" described in the BASIC Programming chapter of the CIO-AD16JR-AT User's Manual. refer to that manual for more detailed information about what they do. Refer to the header at the top of each routine for more information on the routine's arguments.

4.4.1 RepInFore() - A/D Input in Foreground

BASIC Equivalent - Mode 40

Description - Allocates a buffer, starts up a background analog input, waits for it complete, then moves the data into the caller's buffers (data and channel tags).

Code Details:

* Saves the current buffer address so that it can be restored later.

* Call DOS to allocate a buffer.

* Starts up Background Sampling

* Reads the status and waits until the background operation completes.

* Call StopIO to shut everything down.

* Calls MoveRepInData to move the data to the caller's channel tag and data array. The interrupt service routine stores the data in the format that it comes down from the board in. Each sample in the buffer consists of a channel tag and A/D value packed into one word in the buffer. MoveRepInData() pulls those words apart and stores them separately in the caller's channel and data arrays.

* Free the allocated buffer.

* Restore the old buffer.

4.4.2 RepInBack() - A/D Input In Background

BASIC Equivalent - Mode 41

Description - Sets up the board, installs an interrupt handler, waits for a trigger and then starts sampling. This routine moves raw data from the A/D board to the buffer at address bufseg:0. After the sampling is complete you should use MoveRepInData to separate the A/D data from the channel tags.

Code Details:

* If a background A/D input is already running it exits with an error.

* (jl) Sets up the channels that will be scanned (muset()).

* Shuts down the pacer clock. This is important to do early beacuse their once you start enabling things there are many cases where you can accidentely get a conversion when you don't intend to if the pacer is running.

* Save the current value of DS in the code segment so that the interrupt service routines can restore it.

* (j2-j4) Check for arguments for legal values

* (j5-j6) Disable the board's interrupts. The control register contains a number of different things including the interrupt level and the board's interrupt enable bit. This code sets the int level and clears the enable bit. It is a bit confusing because the board handles interrupts 2-7 and 10,11. As far as the board is concerned ints 10 and 11 map to ints 0 and 1 in the control register.

* The INT bit (which indicates that an interrupt is pending is cleared by writing any value to the status register.

* The current buffer pointer (curptr) which the interrupt service routine uses is set to 0.

* The gain and the REPIN_ENABLE bit is written. This should put the board into EXTENDED mode. It is read back to see if it did go into extended mode. If not it means that the compatibility switch on the board is turned on which keeps the board locked into COMPATIBLE mode, so it exits with an error.

* (j7) Calculate the PC's Interrupt number based on the interrupt level. The hardware interrupts levels map to software interrupt numbers as follows:

Int Level	Int Number
2 - 7	0ah - 0fh
10, 11	72h, 73h

* (j8-j9) Call DOS to set up vector to new interrupt servic routine.

* Enable clock pulses to pacer counter (pacer is still off though).

* (j10) If set for internal triggering then wait for the digital input D) to go high.

* (j11) Read the Mux register and write it back out again. This clears out the FIFO and resets the multiplexor.

* (12-j14)Build an interrupt mask byte. If it's

interrupt 10 or 11 then you have to subtract 8 from it and send it to the second interrupt controller.

* (j15) Enable the DT Connect feature if requested.#iwe Th:

a

* This board has three new counters in addition to the standard AD16 counters. The BCNTRS are these new ones. Set #0 BCNTR to mode 0. This forces the counter's output high. We need to do this before writing the S0 and S1 bits.

* Write the S0 and S1 bits

* Counters #0 and #1 are cascaded together to count 32 bits. An interrupt is generated when counter #0 reaches 0. Since we are only dealing with 16 bit counts we need to initialize the MSW (counter #0) to 1 and the LSW (counter #1) to the requested total count. That way when the requested number of counts occurs the MSW will be decremented to 0 and generate the interrupt.

* Counter #1 must be set to mode 0 to set it's output high and then to mode 2 to set it low before loading the LSW of the count. This causes it's output which is cascaded into counter #0 to be in the right state before counting begins.

* Clear the INT bit by writing to the status register.

* Enable the board's interrupts.

* start the pacer clock and the sampling begins.

4.4.3 MovRepInData() - Move data into caller's buffers

BASIC Equivalent - Mode 9

Description - The raw data that is read from the A/D is one word per sample. The four lower bits of the LSB contain the channel tag. The upper 4 bits of the LSB and the 8 bits of the MSB contain the A/D data. This routine separates the raw data in the buffer at address bufseg:0 into two arrays of integers. The array at dataseg:dataptr is filled with the A/D data and the array at chanseg:chanptr is filled with the channel tags.

Code Details:

* If the count is zero, there is no data to move so just exit.

* (j1,j2) Set up pointers and count to move data from bufseg:0 to dataseg:dataptr.

* (j3-j5) Move data word by word. For each word, mask out the channel tag bits and shift the remaining bits into the proper position (the 12 LSBs).

* (j6) Set up pointers and count to move data from bufseg:0 to chanseg:chanptr.

* (j7) Move data word by word. For each word, mask out the A/D data. What's left is the channel tag.

4.4.4 PreTrigFore() - Pre-Trigger Input in Foreground

BASIC Equivalent - Mode 42

Description - Allocates a buffer, starts up a background pre-trigger input, waits for it complete, then moves the data into the caller's buffers (data and channel tags).

Code Details:

* Saves the current buffer address so that it can be restored later.

* Call DOS to allocate a buffer.

* Starts up Background pre-triggered sampling

* Reads the status and waits until the background operation completes.

* Call StopIO to shut everything down.

* Calls MovePreTrigData to move the data to the caller's channel tag and data array. The interrupt service routine stores the data in the buffer at bufseg:0. This data is in packed (channel tag+A/D Data) format. The data is also out of order in the buffer (i.e. the first pre-trigger sample is not at the start of the buffer). MovePreTrigData() rotates the data to the correct order and pulls it apart into a separate A/D data (dataseg:dataptr) and channel tag array (chanseg:chanptr).

* Free the allocated buffer.

* Restore the old buffer.

4.4.5 PreTrigBack() - Pre-Trigger Input In Background

BASIC Equivalent - Mode 43

Description - Sets up the board, installs an interrupt handler, and then starts pre-triggered sampling. This routine moves raw data from the A/D board to the buffer at address bufseg:0. After the sampling is complete you must use MovePreTrigData to separate the A/D data from the channel tags and to rotate it into the correct order.

Code Details:

* If a background A/D input is already running it exits with an error.

* (j1) Sets up the channels that will be scanned (muset()).

* Start up the A/D pacer clock.

* (j2) Save the current value of DS in the code segment so that the interrupt service routines can restore it.

* (j2-j4) Check for arguments for legal values.

* Calculate the address of the end of the buffer.

* (j5) Check that the trigger line is low before starting. If not, exit with an error.

* Initialzie the variables that the interrupt service routine uses.

* Enable the pacer clock's gate

* (j7) Disable the board's interrupts. The control register contains a number of different things including the interrupt level and the board's interrupt enable bit. This code sets the int level and clears the enable bit. It is a bit confusing because the board handles interrupts 2-7 and 10,11. As far as the board is concerned ints 10 and 11 map to ints 0 and 1 in the control register.

* (j8) The gain and the REPIN_ENABLE bit is written. This should put the board into EXTENDED mode. It is read back to see if it did go into extended mode. If not it means that the compatibility switch on the board is turned on which keeps the board locked into COMPATIBLE mode, so it exits with an error.

* (j9, j10) Read the current interrupt mask register value. This is complicated by the fact that int levels 10 and 11 are handled by a different interrupt controller and so thet are in a different mask register.

* Calculate the PC's Interrupt number based on the interrupt level. The hardware interrupts levels map to software interrupt numbers as follows:

Int Level	Int Number
2 - 7	0ah - 0fh
10, 11	72h, 73h

* (j11) Call DOS to set up vector to new interrupt service routine.

* Clear the INT bit by writing any value to the status regsiter.

* (j12) Enable the selected interrupt on the interrupt controller by clearing it's bit in the interrupt mask register. Once again if the int level is 10 or 11 it is on a different 8259.

* Reset the FIFO and the mux by writing to the mux register.

* Enable pre-triggering by setting the bit in MISC_REG.

* This board has three new counters (the BCNTRS) in addition to the standard AD16 counters. Counters #0 and #1 are cascaded together to count 32 bits. This counter begins decrementing once the pre-trigger occurs. It should be loaded with the post trigger count (total count - pre-trigger count). An interrupt is generated when counter #0 reaches 0. Since we are only dealing with 16 bit counts you must to initialize the MSW (counter #0) to 1 and the LSW (counter #1) to the requested post-trigger count.

  That way when the requested number of counts occurs the MSW will be decremented to 0 and generate the interrupt.

* Counter #1 must be set to mode 0 and then mode to Mode 5 again. This will clock the MSW (counter 0) and force it to load it's count (1).

* Clear the INT bit by writing anything to the status register.

* Enable the interrupts on the board.

* Turn on the pacer.

4.4.6 MovePreTrigData() - Move pre-trigger data

BASIC Equivalent - Mode 44

Description - The raw data that is read from the A/D is one word per sample. The four lower bits of the LSB contain the channel tag. The upper 4 bits of the LSB and the 8 bits of the MSB contain the A/D data. The pre-trigger interrupt service routine treats the buffer as a circular buffer. The trigger sample can occur any where in the buffer. This also means that the first sample can occur anywhere in the buffer.

This routine takes the pre-trigger data from the circular buffer and pulls apart the A/D data from the channel tags and stores it in the user's array in the proper order.

Code Details:

* If the count is zero, there is no data to move so just exit.

* Calculate where in the circular buffer the first point is. Trigadr contains the address of the trigger sample. Multiply the number of pre-trigger samples X 2 to get the offset to the first point.

* If the offset is bigger then the trigger address it means that the data wraps from the end of the buffer to the beginning and that the starting address is higher then trigadr.

* In the case where it wrapped, the difference between the offset and the trigger address is the offset from the end of the buffer to the first point.

* (j4) The data will have to be moved in two chunks. The first chunk will be from the starting address to the end of the buffer. The number of words to move is (bufend-star) /2

* Set up the destination pointer (ES:DI) to point at dataseg:dataptr.

* Do the block move

* Then move the remaining block which starts at the beginning of the buffer. So set the source pointer (DS:SI) to bufseg:0

* Do the 2nd block move - Now the raw data is in the dataseg:dataptr array in the correct order.

* If the channel tags are supposed to be moved then

move and mask the raw data into the channel array.

* Finally we have to convert the data in the data array from it's raw format to A/D data format.

* Check whether the Bipolar bit is set, which means that the data must be complemented.

* (j7-j10) Get each word from the dataseg:dataptr array, AND out the channel tag, shift it right 4 bits, and if the Bipolar bit was set then complement it, and store it back in the array.

4.4.7 GetStatus() - Get status of background operation

BASIC Equivalent - Mode 8

Description - Returns 0 if the background operation is complete or 1 if it still in progress.

4.4.8 StopIO() - Stops the background operation

BASIC Equivalent - Mode 7

Description - Stops any background process that is currently exceuting.

Code Details:
* It Disables the interrupt by writing to the PC's 8259 interrupt mask register. Then disables the interrupt line on the A/D board.

4.4.9 AllocBuffer() - Allocates a buffer for the raw data

BASIC Equivalent - None

Description - Calls DOS to allocate a temporary buffer to read the raw data into. The buffer size is the next even multiple of 1024 learger then count*2. This is so that the buffer is always big enough to read whole packets into. This is necessary for the pre-trigger buffering.

Code Details
* First the count-1 is divided by 512 and then incremented. This yields the number of packets that are required.

* Each packet is 1024 bytes (512 samples X 2) but the DOS alloc function requires the count to be in terms of paragraphs (16 bytes). Combining the x1024 and the /16 yields a multiply by 64 - SAL AX,6

4.4.10 FreeBuffer() - Frees the allocate buffer

BASIC Equivalent - None

Description - Calls DOS to free the previously allocated buffer.

4.5 Private Routines

The private routines are ones that are used internally. They should never have to be called directly from an application program.

4.5.1 repinint() - REP INS interrupt handler

Description - This is the interrupt service routine that is called for standard REP INS transfer. This interrupt handler is installed by RepInBack. On each interrupt it downloads a packet from the A/D board.

Code Details:

* Restores EXAMPLE.ASM's data segment

* Sets up pointer into buffer ES:DI = bufseg:curptr

* Sets up the number of words to move. Normally this is set to PACKETSIZE (512) but on the last packet it will be less if the total count is not an even multiple of 512.

* Does the REP INSW input into the buffer.

* (j2) Updates the current count and checks whether it was the last packet.

* If it was the last packet then -
  * Checks if recycl is set to CONTINUOUS. If so it resets back to the beginning of the buffer.

  * (J3) If it is not CONTINUOUS mode then it disables the interrupts, (j4) restores the interrupt mask register, (j5) switches back to COMPATIBLE mode, and checks for overrun errors.

* Resets the INT bit on the board

* Sends the End-Of_interrupt command to the PC's interrupt controller. If the int level is 10 or 11 it is using the cascaded controller so it must send the EOI command to both 8259s.

4.5.2 pretrigint() - Pre-triggerring interrupt handler

Description - This is the interrupt service routine that is called for pre-triggered input. This interrupt handler is installed by PreTrigBack. On each interrupt it downloads a packet from the A/D board.

Code Details:

* Restores EXAMPLE.ASM's data segment

* Sets up pointer into buffer ES:DI = bufseg:curptr

* Sets up the number of words to move. If the trigger has already occurred then it subtracts the current count from the total count. If the difference is less then the PACKETSIZE then it is the last packet so it sets the count equal to that difference.

* (J1) If the trigger has not occurred then it always moves a full packet cx=PACKETSIZE.

* (j2) Move the data with REP INSW into buffer.

* If the trigger already occurred then add the number of bytes that was just moved to the current count.

* (j3) If the trigger has already occurred then check if it occurred in this packet. If so -

    * Read the pre-trigger counter (BCNTR2).

    * If the count=PACKETSIZE it is not a real trigger so ignore it. This can happen occasionally.

    * If it is a real trigger calculate it's address within the buffer. The value read back from the counter is an index within the current packet. Multiply that by 2 to get the byte offset within the packet. Add that to the current buffer pointer to get the offset within the buffer. Store it in trigadr.

    * Add the number of samples before the trigger in this packet to the current count. At this point current count is keeping track of the number of pre-trigger samples.

    * Check whether the required number of samples were taken before the trigger occurred. If not (j4) exit with an error.

    * (j5) Otherwise set currcount equal to the pre-trigger count. From here on current count will be used to count the total number of samples.

    * Calculate how many samples came after the trigger in the packet. Add that to the current count.

* (j7) DI register points to the byte after the last REP INSW move. Check whether it is pointing at the end of the buffer. If so reset it to 0.

* (j8) Save the current buffer pointer.

* If trigger has already occurred then check if all done count==curcount.

* If so then

    * Resets the INT bit on the board

    * Sends the End-Of_interrupt command to the PC's interrupt controller. If the int level is 10 or 11 it is using the cascaded controller so it must send the EOI command to both 8259s.

* Resets the INT bit on the board

* Sends the End-Of_interrupt command to the PC's interrupt controller. If the int level is 10 or 11 it is using the cascaded controller so it must send the EOI command to both 8259s.

4.5.3 waitidle() - Waits for the A/D to be idle

Description - Resets the MUX and then waits in a loop for the A/D busy line to clear. If it doesn't eventually clear then it exits with an error.

4.5.4 startpacer() - Sets pacer freq and starts it up

Description - Sets the counters so that the pacer clock runs at the desired frequency. The frequency is generated by dividing down a 10 Mhz clock, so all frequencies cannot be generated. This routine calculates the divisor to generate a frequency as near as possible to the requested one. It then calculates the real frequency and returns it in Khz and hz.

Code Details:
* Checks for legal speeds

* (j3) Calculate frequency - hkz x 1000 + hz and store it in freq.

* Divide 10Mhz by the requested frequency and store the result in product.

* Now the job is to find two 16 bit numbers that when multiplied together are approximately equal to product. Start by setting one of them (ctr1) to 1.

* (j4) This is a loop that divides the product by the current ctr1 value. If the result is > 32767 (15 bits) then it incrments ctr1 and tries again.

* Take the ctr1 and ctr2 values and write then to the counter (CALL CTR2SET).

* Divide 10MHZ by the product of CTR! x CTR2 to get the actual frequency that the pacer is running at.

* Divide the frequency by 1000 to get the khz and hz values.

4.5.5 stoppacer() - Stops the pacer clock

Description - Stops the pacer clock by setting the counter mode.

4.5.6 ctrset() - Load the pacer's counters

Description - Sets the modes for each of the pacer counters and loads the divide down values.

Code Details - The pacer runs by dividing the 10 Mhz clock down via a 32 bit counter. The 32 bit counter is made up of two 16 bit counters that are cascaded together. This routine sets the modes of each of these counters and then loads their initial counts.

4.5.7 muxset() - Sets the channels for the multiplexor

Description - The multiplexor controls the channels that will be scanned by the A/D. This routine sets the lower channel and upper channel of the channel range that will be scanned

4.5.8 div_qd() - Divide a quad word by a double word

Description - This routine is part of the Spontaneous Assembly Library. You may use and/or modify it so long as you reproduce their copyright notice;

5. The Test Program TESTEX.C

5.1 Introduction to TESTEX.C

The TESTEX.C is a simple C program that demonstrates how to call the routines in EXAMPLE.ASM. It is also a useful tool for testing the EXTENDED features of the Jr-AT board.

The program was written with Version 6.0 of MicroSoft C. It should be very easy to move it to another C compiler. The obvious MicroSoft library dependenies were isolated into a few simple routines at the end of the source file.

5.2 Interface to EXAMPLE.ASM

As written, the interface between TESTEX.C and the routines in EXAMPLE.ASM is very simple - global variables. This is done to make it easy to understand the code so that you can use whatever type of interface you desire for the language that you are using.

The EXAMPLE.ASM code declares .model small,c which provides a simple to use method of setting up the interface between the assembly language code and it's caller. The .model directive takes care of setting up the appropriate segment names and also pre-pending all function names with an underscore character (_) as MicroSoft C does.

Refer to the Microsoft MASM documentation on the .model directive for help in interfacing to other languages.

5.2 Building TESTEX.EXE

A make file called TESTEX.MAK is provided that will build TESTEX.EXE by assembling EXAMPLE.ASM, compiling TESTEX.C and linking them together. The make file was written for Microsoft's nmk.

5.3 Running TESTEX.EXE

The program presents a menu of the four new analog I/O functions RepInFore(), RepInBack(), PreTrigFore() and PreTrigBack().

Choose one by number and it should execute the function and collect 1024 samples at a 1 kHz sample rate. If you select a pre-trigger function it will collect 512 pre-trigger samples and 512 post-trigger samples.

If you select a REP INS function then the trigger line (digital input D0) must be either tied to TTL high OR floating. If not the program will wait until it is.

If you select the Pre_triggered functions then the trigger line (digital input D0) must be tied to TTL LOW when you

make the selection. Once you select from the menu the program begins sampling. When the digital input goes HIGH the Jr-AT hardware will treat that as the trigger signal and sample 512 more points.

Once data acquisition is complete, the data will be displayed on the screen in three columns - sample number, channel tag, A/D value.

5.4 Modifying TESTEX.C

All of the data acquistion parameters are defined as constants at the start of the program. If you wish to change the channel numbers, sample rate, count, etc, - change these constants.

Artisan Technology Group is an independent supplier of quality pre-owned equipment

Gold-standard solutions

Extend the life of your critical industrial, commercial, and military systems with our superior service and support.

We buy equipment

Planning to upgrade your current equipment? Have surplus equipment taking up shelf space? We'll give it a new home.

Learn more!

Visit us at **artisantg.com** for more info on price quotes, drivers, technical specifications, manuals, and documentation.

Artisan Scientific Corporation dba Artisan Technology Group is not an affiliate, representative, or authorized distributor for any manufacturer listed herein.

We're here to make your life easier. How can we help you today?

(217) 352-9330 | sales@artisantg.com | artisantg.com

ARTISAN® TECHNOLOGY GROUP