ACCES I/O PRODUCTS INC
10623 Roselle Street, San Diego, CA 92121
TEL (858)550-9559 FAX (858)550-7322

MODEL AD12-8

USER MANUAL

FILE: MAD12-8.C3a

Notice

The information in this document is provided for reference only. ACCES does not assume any liability arising out of the application or use of the information or products described herein. This document may contain or reference information and products protected by copyrights or patents and does not convey any license under the patent rights of ACCES, nor the rights of others.

IBM PC, PC/XT, and PC/AT are registered trademarks of the International Business Machines Corporation.

Warranty

Prior to shipment, ACCES equipment is thoroughly inspected and tested to applicable specifications. However, should equipment failure occur, ACCES assures its customers that prompt service and support will be available. All equipment originally manufactured by ACCES which is found to be defective will be repaired or replaced subject to the following considerations.

Terms and Conditions

If a unit is suspected of failure, contact ACCES' Customer Service department. Be prepared to give the unit model number, serial number, and a description of the failure symptom(s). We may suggest some simple tests to confirm the failure. We will assign a Return Material Authorization (RMA) number which must appear on the outer label of the return package. All units/components should be properly packed for handling and returned with freight prepaid to the ACCES designated Service Center, and will be returned to the customer's/user's site freight prepaid and invoiced.

Coverage

First Three Years: Returned unit/part will be repaired and/or replaced at ACCES option with no charge for labor or parts not excluded by warranty. Warranty commences with equipment shipment.

Following Years: Throughout your equipment's lifetime, ACCES stands ready to provide on-site or in-plant service at reasonable rates similar to those of other manufacturers in the industry.

Equipment Not Manufactured by ACCES

Equipment provided but not manufactured by ACCES is warranted and will be repaired according to the terms and conditions of the respective equipment manufacturer's warranty.

General

Under this Warranty, liability of ACCES is limited to replacing, repairing or issuing credit (at ACCES discretion) for any products which are proved to be defective during the warranty period. In no case is ACCES liable for consequential or special damage arriving from use or misuse of our product. The customer is responsible for all charges caused by modifications or additions to ACCES equipment not approved in writing by ACCES or, if in ACCES opinion the equipment has been subjected to abnormal use. "Abnormal use" for purposes of this warranty is defined as any use to which the equipment is exposed other than that use specified or intended as evidenced by purchase or sales representation. Other than the above, no other warranty, expressed or implied, shall apply to any and all such equipment furnished or sold by ACCES.

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

The AD12-8 is a multifunction, moderate-speed analog-to-digital converter card with counter/timers and digital I/O ports. This card may be used in IBM Personal Computers and other compatible computers. The card is seven inches long and requires one slot in the computer. All external connections are made through a standard 37-pin D-type connector at the rear of the computer. The following paragraphs describe the functions provided by the AD12-8 card.

Analog Inputs

The card accepts eight single-ended analog input channels. The full scale input for all channels is jumper selectable, either ±10V (0.00488V resolution), or ±5V (0.00244V resolution), or unipolar 0 to 10V (0.00244V resolution). Inputs are single-ended with a common ground and can withstand overvoltages up to ±30 volts and brief transients of several hundred volts. When power is off, the inputs are open-circuited providing fail-safe operation. The analog-to-digital converter (A/D) is a 12-bit successive-approximation type with a sample and hold input. Conversion time is typically 25µS (35µS maximum) and, depending on the speed of the software and computer platform, throughputs of up to 40,000 conversions per second are attainable.

Input System Expansion

The AD12-8 card may be used alone or it can support up to eight AIM-16 or LVDT-8 analog input expansion cards. An 8-bit standard LSTTL logic output from the AD12-8 is used to control the AIM-16 (These outputs are separate from the discrete I/O lines discussed later.). Four bits are used to select one of 16 analog input channels at the AIM-16. Three bits are used to select one of eight gains for the selected input channel. The remaining bit is a digital output (sink current 8mA, source current 0.4mA). When interfacing to the LVDT-8, three bits are used to select one of eight LVDT-8 inputs. Since the eight-input multiplexer on the AD12-8 card is software addressable, an input expansion card may be connected to each input, for a maximum of 128 channels in the system. If more than 128 analog inputs are required, a second AD12-8 with companion input expansion cards can be used.

Discrete Digital I/O

The AD12-8 card provides eight bits of individually jumper-programmed digital input and/or output. The outputs can sink 24mA or source 2.6mA. For ease of use, we suggest that you use the Screw Terminal Accessory Board, model STA-37 or TAD12-8 to make external connections.

Reference Voltage Output

A precision +10.0V (±0.1V) reference voltage output is derived from the A/D converter reference. The reference voltage is buffered and will source up to 220mA, which is sufficient for exciting several strain gage sensors, load cells, or other transducers.

Counter/Timer

The AD12-8 includes a type 8254 counter/timer which has three 16-bit programmable synchronous down counters. This chip is used for event counting, pulse and waveform generation, frequency and period measurement etc. A/D conversion cycles may be initiated by the Counter/Timer by installing the TMR/EXT jumper in the TMR position. Counter #1 is connected to a 1/32 multiple of the computer color Oscillator clock (14.31818MHz). The output of counter #1 is used as the clock for Counter #2 and also Counter #0 when so configured by installing the CLK0 jumper. Counter #0 is fully accessible for general purpose use.

Interrupts

Interrupts are supported from either external inputs or A/D conversion completion. Selection of the interrupt source and interrupt levels (IRQ2-7), is made by jumper. Interrupts are software enabled and disabled. An interrupt request may be canceled by one or more of the following three signals:

a. The computer reset signal.
b. The beginning of an A/D conversion cycle.
c. The writing of a command word to the card. That is, either updating the gain or channel selection multiplexer on the AIM-16 expansion card, or the AD12-8 multiplexer.

Specifications

Analog Inputs

- Channels: Eight single-ended inputs with common ground.
- Voltage Ranges: Jumper selectable, ±10V, ±5V, or 0-10V.
- Resolution: 12 binary bits.
- Accuracy: ±0.02% of reading ±1 LSB.
- Input Impedance: 10 MΩ or 125nA at 25 °C.
- Overvoltage: ±30VDC.
- Linearity: ±1 LSB.
- Temp. Coefficient: ±10 µV/ °C. zero stability.
- ±25 µV/ °C. gain stability.
- Common Mode Rejection: 90 dB when gain = 1 (when used with AIM-16)
  125 dB when gain = 100
- Throughput: 35 microseconds maximum.

Reference Voltage Output

- Voltage: 10.0VDC ±0.1VDC at up to 220mA.

Digital I/O

Inputs

- Logic high: 2.4 to 5.0 VDC at 0.4mA source current.
- Logic low: 0 to 0.4 VDC at 8mA sink current.

Outputs (DIO0 through DIO7)

- Logic high: 2.8V to 5.0V at 2.6 mA source current.
- Logic low: 0V to 0.4V at 24mA sink current.

Outputs (GN0 through GN2 and OP0 through OP3)

- Logic high: 2.4 to 5.0 VDC at 0.4mA source current.
- Logic low: 0 to 0.4 VDC at 8mA sink current.

Interrupt Channel

- Levels: Levels 2 through 7, jumper selectable.
- Enable: Via software.
- Source: End of A/D conversion, or external input.

Programmable Timer

- Type: 8254 programmable interval timer.
- Counters: Three 16-bit down counters, two permanently concatenated with 1/32 multiple of the color oscillator clock as programmable timer. One is uncommitted.
- Output Drive: 2.2mA at 0.45V (5 LSTTL loads).
- Input Gate: TTL/DTL/CMOS compatible.
- Clock Frequency: DC to 8 MHz.
- Active Count Edge: Negative edge.
- Min Clock Pulse Width: 125nS.
- Timer Range: 4 MHz to <1 pulse/hr.

Environmental

- Operating Temp: 0 °C. to 60 °C.
- Storage Temp: -40 °C. to 100 °C.
- Humidity: 0 to 90% RH, non-condensing.
- Size: 7.0 inches long (178 mm), requires full-size slot.
- Power Required: +5VDC: 150 mA typical.
  +12VDC: 10 mA typical.
  -12VDC: 10 mA typical.

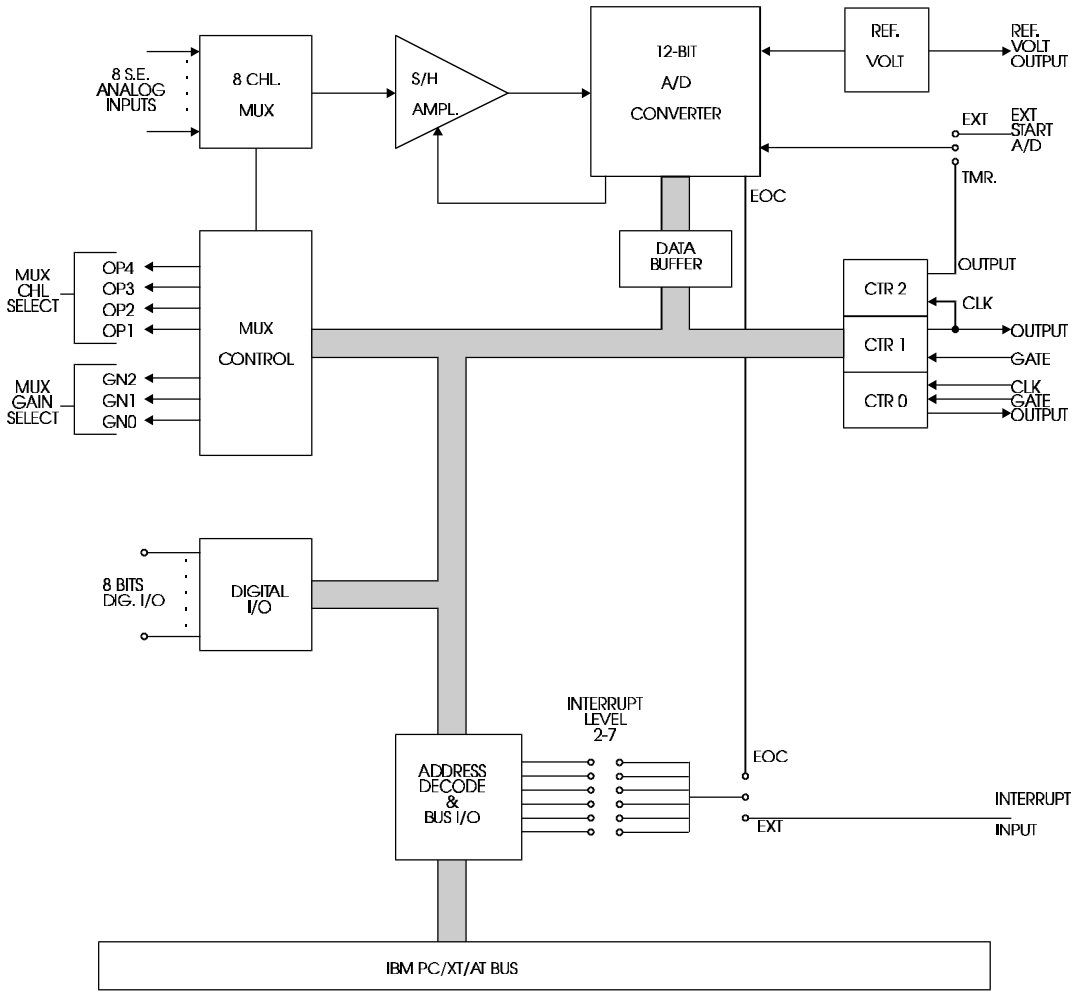

Figure 1-1: AD12-8 Block Diagram

Utility Software

Utility software included with the AD12-8 is provided on a CD. A program named FINDBASE can help you to select the base address for the card. One setup program is a configuration and calibration tool for the AD12-8, and the second setup program is used when an AIM-16 and/or LVDT-8 is used in conjunction with the AD12-8. In the case of the AIM-16, gains are assignable on a channel-by-channel basis. Linearization for all the commonly used thermocouple types as well as for platinum RTDs is also menu selectable.

A driver configuration file is generated or modified by the setup program and can be used to configure the AD12-8 driver. In the section System Software describes the format of this configuration file. This driver has 17 Tasks as will be described in detail in Ad12-8 Driver Reference of this manual.

Sample programs are provided in QuickBASIC, Pascal and "C".

Enhancements

Capabilities of the AD12-8 can be greatly enhanced by use of one or more of the following hardware devices or software packages:

a. STA-37 and TAD12-8 Screw Terminal Cards.

These terminal boards allow direct input/output signal connections via screw terminals. The TAD12-8 also provides LED's to show status of digital I/O bits. Both cards provide a breadboard area with ±12V and +5V computer power. This breadboard area can be used for amplifiers, filters, and other user-assembled circuits.

b. AIM-16 Expansion Multiplexer and Instrumentation Amplifier

The AIM-16 is a 16-channel amplifier/multiplexer that features differential or single-ended input capability and a choice of either DIP switch selectable gains or software programmable gains. The AIM-16 allows multiplexing of 16 analog input signals to a single AD12-8 analog input channel. As described earlier, up to eight AIM-16's can be connected to a single AD12-8 to provide input capability for up to 128 analog inputs.

The AIM-16 includes a low-drift instrumentation amplifier with DIP switch selectable gains of 0.5, 1, 2, 5, 10, 25, 50, 100, 200, 500, and 1000. In addition, these gains can be software programmed to provide individual gains on a channel-by-channel basis.

For thermocouple measurements, a cold junction sensor is provided to allow reference junction compensation, via software, for thermocouple inputs. The reference junction output may be assigned to channel 0 of the AIM-16 or, alternatively, may be jumpered to an unused AD12-8 input channel. Open-thermocouple or "break detect" circuitry is provided.

The AIM-16 may also be used with 3-wire RTD's (AIM-16R), strain gages, and 4-20mA current transmitter inputs. In this latter case, an application-specific version, the AIM-16I, is available.

c. LVDT-8 Multiplexer and Interface Card

This card provides AC excitation and signal conditioning to eight independent LVDT transducers. As many as eight LVDT-8's may be connected to an AD12-8 to accommodate up to 64 transducers.

d. SSH-04/-08 Simultaneous Sample and Hold Card

These cards install external to the computer and contain four and eight identical sample and hold circuits respectively. Each circuit consists of a differential-input, programmable-gain amplifier and a sample and hold amplifier. Outputs of the sample and hold amplifiers are coupled through a multiplexer and a common buffer amplifier.

e. EASY ACCESS

EASY ACCESS is a real-time, menu-driven, data acquisition software package that takes signals from measurement devices and enters them into the computer without need for you to do any application programming.

EASY ACCESS supports up to 128 analog and 120 digital sensor inputs and also gives you the ability to define up to 80 additional "calculated channels". Calculated channels are software channels derived from mathematical computations performed on sensor input data or on other calculated channels.

Data files generated by EASY ACCESS can be formatted for direct link to LOTUS 1-2-3 or other analysis packages, thus providing full analytical power and graphing functions for data reduction. EASY ACCESS is suggested to anyone involved in repetitive data recording, or anyone who wishes to minimize their programming involvement when using the AD12-8.

f. LabTech Notebook is a menu-driven data acquisition software package. It is capable of foreground or background operation and, although more costly than EASY ACCESS, provides more capabilities including ICON setup, variable sample rates on a channel-by-channel basis, additional mathematical and statistical calculations, and compatibility with expanded memory. Both DOS and Windows versions of LabTech Notebook are available.

Chapter 2: Installation

The software provided with this card is contained on either one CD or multiple diskettes and must be installed onto your hard disk prior to use. To do this, perform the following steps as appropriate for your software format and operating system. Substitute the appropriate drive letter for your CD-ROM or disk drive where you see d: or a: respectively in the examples below.

CD Installation

DOS/WIN3.x

1. Place the CD into your CD-ROM drive.
2. Type d:K to change the active drive to the CD-ROM drive.
3. Type installK to run the install program.
4. Follow the on-screen prompts to install the software for this card.

WIN95/98/NT

1. Place the CD into your CD-ROM drive.
2. The CD should automatically run the install program after 30 seconds. If the install program does not run, click START | RUN and type d:install, click OK or press K.
3. Follow the on-screen prompts to install the software for this card.

3.5-Inch Diskette Installation

As with any software package, you should make backup copies for everyday use and store your original master diskettes in a safe location. The easiest way to make a backup copy is to use the DOS DISKCOPY utility.

In a single-drive system, the command is:

diskcopy a: a:K

You will need to swap disks as requested by the system.
In a two-disk system, the command is:

diskcopy a: b:K

This will copy the contents of the master disk in drive A to the backup disk in drive B.

To copy the files on the master diskette to your hard disk, perform the following steps.

01: Place the master diskette into a floppy drive.

02: Change the active drive to the drive that has the diskette installed. For example, if the diskette is in drive A, type a:K.

03: Type installK and follow the on-screen prompts.

Directories Created on the Hard Disk

The installation process will create several directories on your hard disk. If you accept the installation defaults, the following structure will exist.

[CARDNAME]

Root or base directory containing the SETUP.EXE setup program used to help you configure jumpers and calibrate the card.

DOS\PSAMPLES: A subdirectory of [CARDNAME] that contains Pascal samples.
DOS\CSAMPLES: A subdirectory of [CARDNAME] that contains "C" samples.
Win32\language: Subdirectories containing samples for Win95/98 and NT.

WinRisc.exe

A Windows dumb-terminal type communication program designed for RS422/485 operation. Used primarily with Remote Data Acquisition Pods and our RS422/485 serial communication product line. Can be used to say hello to an installed modem.

ACCES32

This directory contains the Windows 95/98/NT driver used to provide access to the hardware registers when writing 32-bit Windows software. Several samples are provided in a variety of languages to demonstrate how to use this driver. The DLL provides four functions (InPortB, OutPortB, InPort, and OutPort) to access the hardware.

This directory also contains the device driver for Windows NT, ACCESNT.SYS. This device driver provides register-level hardware access in Windows NT. Two methods of using the driver are available, through ACCES32.DLL (recommended) and through the DeviceIOControl handles provided by ACCESNT.SYS (slightly faster).

SAMPLES

Samples for using ACCES32.DLL are provided in this directory. Using this DLL not only makes the hardware programming easier (MUCH easier), but also one source file can be used for both Windows 95/98 and WindowsNT. One executable can run under both operating systems and still have full access to the hardware registers. The DLL is used exactly like any other DLL, so it is compatible with any language capable of using 32-bit DLLs. Consult the manuals provided with your language's compiler for information on using DLLs in your specific environment.

VBACCES

This directory contains sixteen-bit DLL drivers for use with VisualBASIC 3.0 and Windows 3.1 only. These drivers provide four functions, similar to the ACCES32.DLL. However, this DLL is only compatible with 16-bit executables. Migration from 16-bit to 32-bit is simplified because of the similarity between VBACCES and ACCES32.

PCI

This directory contains PCI-bus specific programs and information. If you are not using a PCI card, this directory will not be installed.

SOURCE

A utility program is provided with source code you can use to determine allocated resources at run-time from your own programs in DOS.

PCIFind.exe

A utility for DOS and Windows to determine what base addresses and IRQs are allocated to installed PCI cards. This program runs two versions, depending on the operating system. Windows 95/98/NT displays a GUI interface, and modifies the registry. When run from DOS or Windows3.x, a text interface is used. For information about the format of the registry key, consult the card-specific samples provided with the hardware. In Windows NT, NTioPCI.SYS runs each time the computer is booted, thereby refreshing the registry as PCI hardware is added or removed. In Windows 95/98/NT PCIFind.EXE places itself in the boot-sequence of the OS to refresh the registry on each power-up.

This program also provides some COM configuration when used with PCI COM ports. Specifically, it will configure compatible COM cards for IRQ sharing and multiple port issues.

WIN32IRQ

This directory provides a generic interface for IRQ handling in Windows 95/98/NT. Source code is provided for the driver, greatly simplifying the creation of custom drivers for specific needs. Samples are provided to demonstrate the use of the generic driver. Note that the use of IRQs in near-real-time data acquisition programs requires multi-threaded application programming techniques and must be considered an intermediate to advanced programming topic. Delphi, C++ Builder, and Visual C++ samples are provided.

Findbase.exe

DOS utility to determine an available base address for ISA bus , non-Plug-n-Play cards. Run this program once, before the hardware is installed in the computer, to determine an available address to give the card. Once the address has been determined, run the setup program provided with the hardware to see instructions on setting the address switch and various option selections.

Poly.exe

A generic utility to convert a table of data into an nth order polynomial. Useful for calculating linearization polynomial coefficients for thermocouples and other non-linear sensors.

Risc.bat

A batch file demonstrating the command line parameters of RISCTerm.exe.

RISCTerm.exe

A dumb-terminal type communication program designed for RS422/485 operation. Used primarily with Remote Data Acquisition Pods and our RS422/485 serial communication product line. Can be used to say hello to an installed modem. RISCTerm stands for Really Incredibly Simple Communications TERMinal.

Configuration File

The Configuration File has several purposes; most of which are associated with use of the software drivers provided with your AD12-8 card. These are as follows:

a. Provide means to automatically configure the driver and, thus, avoid need for multiple calls to the driver to do the setup.
b. Allow the setup programs to do the work of configuring the driver. When you use the setup programs to assist in configuring the card, this information is saved in the configuration file and can then be used by the drivers.
c. When you use the EASY ACCESS software package, that software uses the configuration file to configure itself.

The Configuration File, SETUP.CFG, is generated or modified with the setup programs. It can also be generated or modified by an editor or a word processor in the non-document mode. This file is a structured file containing setup information for:

a. AD12-8 plug in PC card.
b. AIM-16 sub-multiplexer card.
c. LVDT-8 sub-multiplexer card.
d. Programmable gain assignments.
e. Curve assignments.

The configuration file must contain 12 lines of information or data. The file is strictly text only (ASCII). Each line of information is made up of a description field, an equal sign (=), and a setup information field. Each line in the configuration file supplies data for a specific parameter and must be in the correct order. Table 2-1 contains an example of a configuration file.

Base Address	=	$300
Extension Card 0	=	1:F:Tt3UUUUUUUUSSSSS
Extension Card 1	=	2:UUUUUUUU
Extension Card 2	=	0:U
Extension Card 3	=	0:S
Extension Card 4	=	0:S
Extension Card 5	=	0:S
Extension Card 6	=	0:S
Extension Card 7	=	0:S
Voltage Range	=	5
Bipolar/Unipolar	=	B
IRQ Channel	=	3

Table 2-1: Configuration File Example

Base Address

Values may be entered as a decimal string, a hexadecimal string, or as a binary string. A decimal string is any string of digits made up of the digits 0 through 9 as follows:

Decimal format DDDDD (e.g. 768)

Hexadecimal strings consist of a string of digits containing the digits 0 through 9 and the letters A through F. Hexadecimal strings may be made up in one of two ways; Pascal or "C" as follows:

Pascal format $HHHH (e.g. $300)
"C" format 0xHHHH (e.g. 0x300)

Binary strings are made up of 0's and 1's preceded by a # as follows:

Binary format #bbbbbbbbbb (e.g. #1100000000)

MUX Extensions

Multiplexer (MUX) channel extension to the A/D board can be made up of an AIM-16 description string, an LVDT-8 description string, or a raw channel description string. The first character following the "=" is a card code that defines which description string follows.

Card Code 0: Raw A/D channel; i.e., no external sub-multiplexer is attached. The 0: may be followed by one of three characters; T, U, or S.

T:	signifies that a reference junction on a sub-multiplexer card is directly attached to the channel.
S:	signifies that the channel is skipped; i.e., not used.
U:	signifies that the channel is unskipped; i.e., treated as a normal A/D input.

Card Code 1: An AIM-16 sub-multiplexer is attached. Immediately following the 1: card code will be a unit-of-measure code. Possible codes are F (units are degrees Fahrenheit), C (units are degrees Celsius), and N (no units specified). Note: If N is used but a thermocouple is installed, then the default is degrees F.

Following the unit-of-measure code is a series of 16 channel-code letters or numbers that determine how the channel is to be used. Any of the following codes may be used for each channel:

S:	The channel is skipped; i.e., not used.
U:	The channel is used (gain code 0).
T:	The channel is used for a thermocouple reference junction
t,k,j,e,r,s,b,a,u:	The channel is used with the indicated thermocouple type or platinum RTD attached. ("a" is used for RTD's with an alpha of 392 and "u" is used for RTD's with an alpha of 385.)
A:	The channel is to have automatic gain ranging. Note that the AIM-16 gain switches must be set for programmable gain for automatic gain ranging to work properly.
0,1,2,3,4,5,6,7:	The channel is to be set to the indicated gain. Gains associated with these code numbers are defined in the gain table contained in TASK 4's reference in the AD12-8 DRIVER REFERENCE. The AIM-16 gain switches must be set for programmable gain for this function to work properly.

Card Code 2: An LVDT-8 sub-multiplexer card is attached. Following the card code is a series of eight channel-code letters that define how the channels are to be used. The letters are U for unskipped and S for skipped.

Voltage Range

Values indicate the voltage range that has been selected by jumpers on the card. Possible values are either 5 or 10.

Bipolar/Unipolar Mode

A "B" signifies that the card is set to bipolar mode while a "U" indicates that the card is set to the unipolar mode.

IRQ Level

Values indicate which IRQ interrupt level will be used. Allowable values are 2 through 7. If the setup program is used, it will put a 0 on this line if no interrupt level is selected. If the driver detects a 0 on this line, it installs a default of IRQ3.

Example

Using the configuration file listed in Table 2-1: Configuration File Example, the information has the following meaning:

a. The base address is set to hex 300 in Pascal format.
b. Channel 0 of the AD12-8 has an AIM-16 sub-multiplexer card attached. The unit of measure for this card is oF. The AIM-16 channel 0 is used for reference junction, channel 1 is a "t" thermocouple type input, channel 2 has a gain code of 3, channels 3 through 10 are used (gain code 0), and the remaining channels are skipped (unused).
c. Channel 1 of the AD12-8 has an LVDT-8 attached, with all eight channels unskipped.
d. Channel 2 of the A/D card is a direct channel that is unskipped.
e. All other A/D channels are direct and skipped.

Chapter 3: Hardware Configuration and Installation

Option Selection

The AD12-8 card features are selected by hardware jumpers. At least one of each of the option categories must be selected if the card is to operate correctly. The setup program provided with the card provides menu-driven pictorial presentations to help you quickly set up the card. You may also refer to the Option Selection Map, and the following sections to set up the card. The card should not be plugged into the computer at this time.

Analog Input Ranges

The available input voltage ranges are ±10V (4.88 mV resolution), ±5V (2.44mV resolution) or unipolar 10V (2.44mV resolution). To select the range, install two jumpers on the card; one at the location labels 10V/5V and one at the location labeled UNIP/BIP.

Range	10V/5V Jumper	UNIP/BIP Jumper
±5V	5V	BIP
±10V	10V	BIP
+10V	5V	UNIP

Counter/Timer

Three 16-bit Counter/Timers are provided on AD12-8. Refer to the Block Diagram in the Functional Description section of this manual for an understanding of the counter/timer configuration and to the Programmable Interval Timer section for a description of applications. Counter #0 is fully accessible to you for event counting, pulse or waveform generation, frequency or period measurement, etc. Counter #1 and Counter #2 are intended for software-programmed, timed A/D Start. The time between measurements may be programmed to almost three hours.

Counter #0 has its clock, gate, and output lines available on the I/O connector. Also, the Counter #0 clock input can be the Counter #1 output if a jumper is installed at the location labeled CLK0. The Counter #0 gate is pulled up to +5V by a resistor.

The clock for Counter #1 is supplied by a 1 ÷ 32 multiple of the 14.31818 Mhz color oscillator. Counter #1's gate is available on the I/O connector and, like the Counter #0 gate is pulled up to +5V by a resistor. The Counter #1 output is accessible at the I/O connector and is also connected to the Counter #2 clock input. The Counter #1 output may also be supplied as a clock input to Counter #0 if the CLK0 jumper is installed as previously noted.

The Counter #2 clock is supplied from the output of Counter #1 as noted previously. The gate is pulled up to +5V and is not accessible at the I/O connector. The Counter #2 output can be used to start preprogrammed A/D conversions if the TMR/EXT jumper is installed in the TMR position.

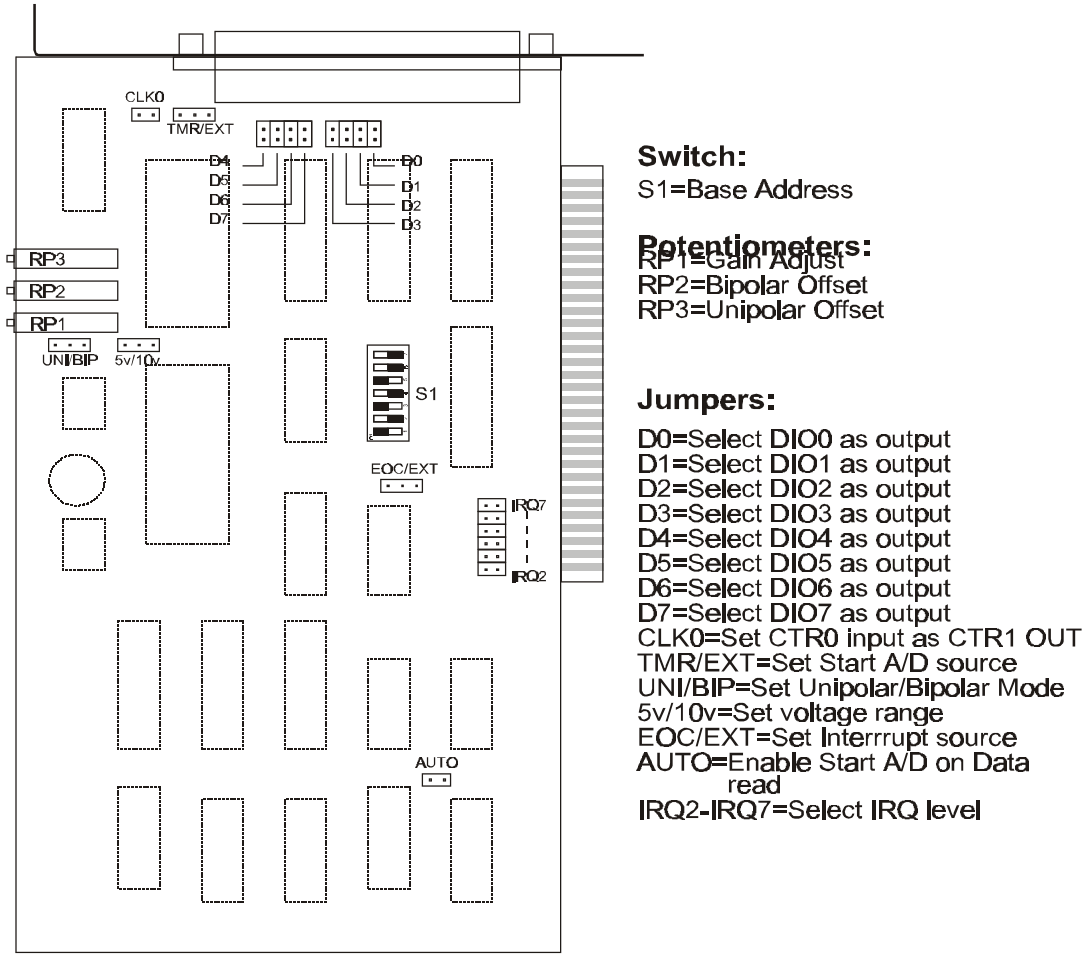

Figure 3-1: Option Selection Map

A/D Conversion Start

An A/D conversion can be started from Counter #2 if the TMR/EXT jumper is in the TMR position. Alternatively, A/D conversions can be initiated by negative pulses on Digital Input 0 if the TMR/EXT jumper is in the EXT position.

A/D conversion start can also be initiated in software by a computer write to BASE ADDRESS + 1 or, if the AUTO jumper is installed, by a computer read of BASE ADDRESS + 0. The AUTO mode is especially useful when high throughput is required.

Interrupts

Interrupts originating from either completion of A/D conversions or from an external source (pin 24) are supported. Interrupts are enabled by setting the IEN bit of Control Register #1 high. Selection of the interrupt source is made by jumper. Install a jumper in the EOC position at jumper location EXT/EOC to select interrupts caused by the end-of-conversion signal. Install a jumper in the EXT position at jumper location EXT/EOC to select external interrupts.

The end-of-conversion interrupt is supported by the AD12-8 software drivers. If you desire to use external interrupts, then you must supply your own interrupt service routine. NOTE: The end-of-conversion interrupt is intended to be used only when A/D conversions are started by the counter/timer(install jumper in the TMR position of the TMR/EXT jumper), or by an external event (install jumper in the EXT position of the TMR/EXT jumper). If the AUTO jumper is installed, or Software Start is used, the use of interrupts will degrade card speed.

Interrupt levels IRQ2 through IRQ7 are available. The desired level is selected by installing a jumper in one of the jumper locations marked IRQ2 through IRQ7.

Digital I/O

The AD12-8 card provides eight individually programmable digital inputs/outputs in addition to eight digital outputs OP0-OP3, GN0-GN2, and USER. These latter eight outputs operate strictly under software control.

Eight jumpers determine the input/output configuration of the eight digital lines identified as DIO0-DIO7. If no jumpers are installed, all eight lines are configured as inputs. Installation of a jumper at locations DO-D7 convert the corresponding line to an output.

Digital inputs are read from a register at BASE ADDRESS + 3. Digital outputs are written to a register at BASE ADDRESS + 0. Note that full feedback is provided for the digital outputs because output status can be read back through the register at BASE ADDRESS + 3.

Selecting a Base Address

You need to select an unused segment of eight consecutive I/O addresses. The base address will be the first address in this segment. The base address may be selected anywhere on a 8-bit boundary within the I/O address range 200-3FF hex (excluding 1F0 through 1F8 in AT's) providing that it does not overlap with other functions. The following procedure will show you how to select the base I/O address.

1. Check Table 3-1: Standard Address Assignments for a list of standard address assignments and then check what addresses are used by any other I/O peripherals that are installed in your computer. (Memory addressing is separate from I/O addressing, so there is no possible conflict with any add-on memory that may be installed in your computer.) We urge that you carefully review the address assignment table before selecting a card address. If the addresses of two installed functions overlap, unpredictable computer behavior will result.
2. From this list (or using the FINDBASE program), select an unused portion of eight consecutive I/O address. Note from the tables that the sections 280-2EF and 330-36F are unused. This address space is good area to select a base address from. Also, if you are not using a given device in listed in the tables, then you may use that base address as well. For example, most computers do not have a prototype card installed. If your computer does not have one, then base address 300 hex is a good choice for a base address.
3. Finally make sure that the base address you have chosen has the last digit as 0 or 8. This insures that your base address is on an 8-byte boundary.

Hex Range	Usage
000-01F	DMA Controller 1
020-03F	INT Controller 1, Master
040-05F	Timer
060-06F	8042 (Keyboard)
070-07F	Real Time Clock, NMI Mask
080-09F	DMA Page Register
0A0-0BF	INT Controller 2
0C0-0DF	DMA Controller 2
0F0	Clear Math Coprocessor Busy
0F1	Reset Coprocessor
0F8-0FF	Arithmetic Processor
1F0-1F8	Fixed Disk
200-207	Game I/O
278-27F	Parallel Printer Port 2
2F8-2FF	Asynchronous Comm'n (Secondary)
300-31F	Prototype Card
360-36F	Reserved
378-37F	Parallel Printer Port 1
380-38F	SDLC or Binary Synchronous Comm'n 2
3A0-3AF	Binary Synchronous Comm'n 1
3B0-3BF	Monochrome Display/Printer
3C0-3CE	Local Area Network
3D0-3DF	Color/Graphic Monitor
3F0-3F7	Floppy Diskette Controller
3F8-3FF	Asynchronous Comm'n (Primary)

Table 3-1: Standard Address Assignments for 286/386/486 Computers

Setting the Base Address

The AD12-8 base address is selected by DIP switch S1 located in the lower right hand portion of the card. Switch S1 controls address bits A3 through A9. Bits A0 through A2 are used for the eight address locations in I/O space required by the AD12-8. The following procedure will show you how to set the base address. See Base Address Example below, for a graphic representation of this example.

1. We will use base address 300 hex as an example. Determine the binary representation for your base address. In our example, 300, the binary representation is 11 0000 0000. The conversion multipliers for each binary bit are contained in the Base Address Example for reference.
2. Locate switch S1 on the lower right side of the card. Note there are seven switches, which will be used to set the seven most-significant bits in the binary representation from step 1).
3. Note from the table below that switch position A9 corresponds to the most significant bit in your binary representation. For each bit in your binary representation, if the bit is a one, turn the corresponding switch off; if the bit is zero, turn the corresponding switch on.

Hex Representation	3		0				0
Binary Representation	1	1	0	0	0	0	0
Conversion Multiplier	2	1	8	4	2	1	8
Switch ID	A9	A8	A7	A6	A5	A4	A3
Switch Setting	OFF	OFF	ON	ON	ON	ON	ON

Table 3-2: Base Address Example

Using the Setup Program to Set the Base Address

The setup program provided with AD12-8 contains an interactive menu-driven program to assist you in setting the base address. The following procedure demonstrates the use of the setup program.

1. Choose a desired base address.
2. Execute the setup program by typing SETA12 and pressing the ENTER key.
3. Select the first item in the menu, "1) Set board address" with the up or down arrow key and press ENTER.
4. Enter the desired base address in hex, the program will display a graphic representation of how you should set the switches. You may press the space bar to try another address.
5. Set DIP switch S1 as shown on the graphic representation.

Installing the AD12-8 Card

The following procedure will show you how to install the AD12-8 inside the computer.

1. Ensure that all options have been set as described in the first part of this chapter. Be sure to pay close attention to base address selection.
2. Turn off the power switch of your computer and remove the power cords from the wall outlet.
3. Remove the computer cover.
4. Locate an unused full length slot, and remove the blank I/O back plate.
5. Insert the card in the slot, and install the I/O back plate screw. To ensure that there is minimum susceptibility to EMI and minimum radiation, it is important that there be a positive chassis ground. Also, proper EMI cabling techniques must be used for input/output wiring.
6. Inspect the installation for proper fit and seating of the card.
7. Replace the computer cover.
8. Reapply power to the computer.
9. Perform the calibration procedure which follows.

Caution

Failure to Remove Power from the Computer Could Result in Electrical Shock, or Damage to Your Computer System.

Calibration and Test

All ACCES cards are calibrated prior to shipment. However, periodic calibration of AD12-8 is recommended to retain full accuracy. The calibration interval depends to a large extent on the type of service that the card is subjected to. For environments where there are frequent large changes of temperature and/or vibration, a three-month interval is suggested. For laboratory or office conditions, six months to a year is acceptable.

A 4-1/2 digit digital multimeter is required as a minimum to perform satisfactory calibration. Also, a voltage calibrator or a stable noise-free DC voltage source that can be used in conjunction with the digital multimeter is required.

Calibration is performed using the SETUP program supplied with your card. This program will lead you through the set up and calibration procedure with prompts and graphic displays that show the settings and adjustment trim pots. This calibration program also serves as a useful test of the AD12-8 A/D functions and can aid in troubleshooting if problems arise.

Calibration Procedure

The following procedure is brief and is intended for use in conjunction with the calibration part of the Setup program.

1. Start the calibration program by typing SETUP and press the K key at the DOS prompt.
2. Use the relevant menu selections to set the switches and jumpers for the manner in which the card will be calibrated; i.e., number of channels, and ±5V range). These settings are used by the calibration portion of the program.
3. Use the arrow key to select option *7.Calibrate A/D*, then press the K key.
4. Use the arrow key to select option *1.Adjust Offset*, from the Action Menu at the top left hand corner of the screen.
5. Following the instructions on the screen, perform the offset adjustment
6. Use the arrow key to select option *2.Adjust Gain*, from the Action Menu.
7. Following the instructions on the screen, perform the gain adjustment
8. Use the arrow key to select option *3.Check*, from the Action Menu.
9. Following the instructions on the screen, perform the calibration check.
10. This completes the calibration procedure.

Chapter 4: Programming

This chapter provides you with information on how to program the AD12-8. First, information is provided on how to program the card using direct register access. Following this is information on using the drivers provided with the AD12-8.

At the lowest level, the AD12-8 can be programmed using direct I/O input and output instructions. In BASICA, these are the INP (X) and OUT X,Y functions. Assembly language and most high level languages have equivalent instructions. Use of these functions usually involves formatting data and dealing with absolute I/O addresses. Although not demanding, this can require many lines of code and requires an understanding of the devices, data format, and architecture of the AD12-8. You may find it easer to design your program using the supplied drivers.

AD12-8 Register Address Map

The AD12-8 uses eight consecutive addresses in I/O space as follows:

Register Address	Read Function	Write Function
BASE ADDRESS + 0	A/D Low Byte	8 bit Digital Output
BASE ADDRESS + 1	A/D High Byte	Start A/D Conversion
BASE ADDRESS + 2	Read status register	Write Control Register #1
BASE ADDRESS + 3	8 bit digital input	Write Control Register #2
BASE ADDRESS + 4	Read Counter #0	Load Counter #0
BASE ADDRESS + 5	Read Counter #1	Load Counter #1
BASE ADDRESS + 6	Read Counter #2	Load Counter #2
BASE ADDRESS + 7	Not Used	Write Counter Control

Table 4-1: AD12-8 Register Address Map

Register Definitions

Control Registers

The AD12-8 card contains two 8-bit control registers. These registers allow software selection and/or control of functions on the AD12-8 card and associated input expansion cards.

Base + 2 Write: Read or write the control register #1.

B7	B6	B5	B4	B3	B2	B1	B0
OP3	OP2	OP1	OP0	IEN	MA2	MA1	MA0

0P0-OP3: These bits correspond to four general-purpose digital output lines. These bits are available on both control registers #1 and #2, in order to save program execution time. These lines can be used for external control functions such as selecting inputs from the AIM-16, or LVDT-8 sub-multiplexer cards.

IEN: This bit enables/disables AD12-8 generated interrupts. 1 = enabled, 0 = disabled.

MA0-MA2: These bits select the analog mutiplexer channel address on the AD12-8 card (Channels 0 through 7).

Base + 3 Write: Read or write the control register #2.

B7	B6	B5	B4	B3	B2	B1	B0
OP3	OP2	OP1	OP0	USER	GN2	GN1	GN0

0P0-OP3: These bits correspond to four general-purpose digital output lines. These bits are available on both control registers #1 and #2, in order to save program execution time. These lines can be used for external control functions such as selecting inputs from the AIM-16, or LVDT-8 sub-multiplexer cards.

USER: This bit is available for use as a digital output bit.

GN0-GN2: These bits correspond to three general purpose digital output lines whose primary purpose is to select one of eight gains on the AIM-16 analog input expansion card.

Status Register

The Status register provides information about the operation of the card.

Base + 2 Read: Read the card status.

B7	B6	B5	B4	B3	B2	B1	B0
EOC	GN2	GN1	GN0	IRQ	MA2	MA1	MA0

EOC: End of conversion. If EOC = 1, an A/D conversion is underway. If EOC is 0, then the A/D data registers contain valid data from the previous conversion and the A/D is ready to perform the next conversion.

GN0-GN2: These bits correspond to three general purpose digital output lines whose primary purpose is to define one of eight gains on AIM-16 sub-multiplexer cards.

IRQ: After generation of an interrupt, the AD12-8 card sets this bit high(1). It is reset to state 0 by a computer reset, or a write to either of the control registers, or the beginning of an A/D conversion.,

MA2-MA0: These bits define the analog multiplexer channel address on the AD12-8 card (channels 0 through 7).

A/D Registers

A/D data are in true binary form and are latched in the A/D registers at the end of each conversion. These are read at BASE ADDRESS and BASE ADDRESS +1 in low-byte/high-byte sequence. The data are available until the end of the next A/D conversion. The status of digital output bits OP0-OP3 are contained in the low byte register.

Base + 0 Read: Contains the lower four bits of the A/D converter output in binary form and the status of digital output bits OP0-OP3 in binary form.

B7	B6	B5	B4	B3	B2	B1	B0
AD3	AD2	AD1	AD0	OP3	OP2	OP1	OP0

AD0-AD3: The lower four bits of the A/D conversion, AD0 is the least-significant bit.

OP0-OP3: The status of digital outputs OP0-OP3.

Base + 1 Read: Contains the upper eight bits of the A/D converter output in binary form.

B7	B6	B5	B4	B3	B2	B1	B0
AD11	AD10	AD9	AD8	AD7	AD6	AD5	AD4

AD4-AD11: The most significant eight bits of the A/D conversion, AD11 is the most-significant bit.

Base + 1 Write: A write to this location starts an A/D conversion. The data written is irrelevant. This causes the EOC bit of the status register to go high until the conversion is complete.

Digital I/O

The AD12-8 card provides 16 bits of digital I/O capability. If the card is to be used with an AIM-16 input multiplexer expansion card, then half of these bits are used for gain programming and multplexer channel selection. If the AIM-16 is not to be used, then these bits may be programmed using the control registers as described earlier in this section.

The AD12-8 card provides eight individually programmable digital inputs/outputs. Eight jumpers determine the input/output configuration of the eight digital lines identified as DIO0-DIO7. If no jumpers are installed, all eight lines are configured as inputs. Installation of a jumper at locations DO-D7 convert the corresponding line to an output.

Digital inputs are read from a register at BASE ADDRESS + 3. Digital outputs are written to a register at BASE ADDRESS + 0.

Base + 0 Write: Write digital output.

B7	B6	B5	B4	B3	B2	B1	B0
DO7	DO6	DO5	DO4	DO3	DO2	DO1	DO0

DO0-DO7: These are digital outputs if so programmed by jumpers D0-D7.

Base + 3 Read: Read digital input.

B7	B6	B5	B4	B3	B2	B1	B0
DI7	DI6	DI5	DI4	DI3	DI2	DI1	DI0

DI0-DI7: These are digital inputs if so programmed by jumpers DO-D7.

Counter/Timer Registers

Base + 4 Write/Read: Counter #0 read or write. When writing, this register is used to load a counter value into the counter. The transfer is either a single or double byte transfer, depending on the control byte written to the counter control register at BASE ADDRESS + 7. If a double byte transfer is used, then the least-significant byte of the 16 bit value is written first, followed by the most significant byte. When reading, the current count of the counter is read. The type of transfer is also set by the control byte.

Additional information about the type 8254 counters is presented in the Programmable Interval Timer section of this manual. However, for a full description of features of this extremely versatile IC, refer to the Intel 8254 data sheet.

Base + 5 Write/Read: Counter #1 read or write. See description for Base + 4 Write/Read.

Base + 6 Write/Read: Counter #2 read or write. See description for Base + 4 Write/Read.

Base + 7 Write: The counters are programmed by writing a control byte into a counter control register at BASE ADDRESS + 7. The control byte specifies the counter to be programmed, the counter mode, the type of read/write operation, and the modulus. The control byte format is as follows:

B7	B6	B5	B4	B3	B2	B1	B0
SC1	SC0	RW1	RW0	M2	M1	M0	BCD

SC0-SC1: These bits select the counter that the control type is destined for.

SC1	SC0	Function
0	0	Program Counter 0
0	1	Program Counter 1
1	0	Program Counter 2
1	1	Read Back Command

RW0-RW1: These bits select the read/write mode of the selected counter.

RW1	RW0	Counter Read/Write Function
0	0	Counter Latch Command
0	1	Read/Write LS Byte
1	0	Read/Write MS Byte
1	1	Read/Write LS Byte, then MS Byte

M0-M2: These bits set the operational mode of the selected counter.

Mode	M2	M1	M0
0	0	0	0
1	0	0	1
2	X	1	0
3	X	1	1
4	1	0	0
5	1	0	1

BCD: Set the selected counter to count in binary (BCD bit = 0) or BCD (BCD bit = 1).

Programming Using the Drivers

Using direct register access to program the AD12-8 is straightforward but the coding can be rather tedious. To assist you in building your application quickly, ACCES provides a driver. This driver is provided in three forms. Which form you use will depend on the programming language that you intend to use in your application. A task reference for this driver is provided in the AD12-8 Driver Reference. The driver file names and their language use are as follows:

A12DRV.BIN	A BASIC loadable driver for use with most interpreted BASIC languages.
A12DRV.OBJ	A Pascal and QuickBASIC linkable driver in object form.
A12DRVC.OBJ	A "C" linkable driver in object form.

Also, to help you understand how to use the driver with your program, sample programs are provided in three languages; "C", Pascal, and QuickBASIC. Sample 3 is provided in Pascal and "C" only. The programs are:

SAMPLE 1	Demonstrates data acquisition using polling.
SAMPLE 2	Demonstrates timer-driven data acquisition using interrupts.
SAMPLE 3	Same as Sample 2 but uses the configuration file to set up the driver.

To access the functions of the driver, a call to a single procedure within the driver is used. The name of the procedure for the driver is A12DRV. The procedure is called with three variables, which are defined as follows:

task: The number of the task to perform. A reference with a list of tasks for each driver are provided in the AD12-8 Driver Reference.

parameters: This is an array of integers which contains information required by the driver. The AD12-8 DRIVER REFERENCE section defines what values need to be passed for each task. The array should hold five integers.

status: An error code is returned in this variable. A zero is returned if there is no error.

When calling the procedure, certain important requirements must be met:

A. The three variables must be declared as global. If they are not, the driver will not be able to find their data segment. Most programming languages only use the data segment for global variables, which is permanent storage. Variables declared in procedures are usually allocated on the stack, which is temporary storage.
B. The driver expects parameters to be integer type variables and will write to and read from the variables on this assumption. The driver will not function properly if non-integer variables are used in the call.
C. The variables should be passed by reference. The driver expects offsets of the variables so that data may be returned when required.
D. The passed variables are positional. That is, the variables must be specified in the sequence (task, parameters, status). Their location is derived sequentially from the variable pointers on the stack.
E. The driver will not function properly if arithmetic functions (+, -, x, etc) are specified within the variable list bracket.

Using the Driver with Turbo or Borland C

The following list shows you how to use the driver with Borland or Turbo C. You may refer to any of the "C" example programs for further illustration.

A. Include the A12DRVC.H header in your program. This simple header provides a function prototype of the procedure call.

```
#include "a12drvc.h"
```

B. Declare the three variables for the driver globally, at the beginning of your program.

```
inttask,params[5],status;
```

C. Make your assignment to these variables as desired for the function you wish to perform. See DRIVER REFERENCE section for details on each task.

D. Make the call to the driver, passing the offset of each parameter.

```
a12drv(FP_OFF(&task),FP_OFF(params),FP_OFF(status));
```

E. Create a project file within the Turbo C environment, and add the name of your program with the .C extension, and the name of the driver with a .OBJ extension.

F. Select "LARGE" memory model under the compiler section of the options menu.

G. Compile and link the program.

Using the Driver with Microsoft C

To use the driver with Microsoft C version 6.0, add the following code to your application code as shown below:

```
_asm
  {
    push DS
    mov AX,ES
    mov DS,AX
  }
/* call driver as normal */
a12drv(FP_OFF(&task),FP_OFF(params),FP_OFF(status));
_asm
  {
    pop DS
  }
```

If you are using a version of Microsoft C prior to version 6.0 use the following code:

```
_asm_emit 0x1E
_asm_emit 0x8E
_asm_emit 0xD8
/* call driver as normal */
a12drv(FP_OFF(&task),FP_OFF(params),FP_OFF(status));
_asm_emit 0x1F
```

These changes work around a peculiarity of Microsoft C, enabling our drivers to locate the variables used in the program.

Using the Driver with Turbo Pascal

The following procedure will show you how to use the driver with Borland Turbo Pascal. You may refer to any of the Pascal example programs for further illustration.

A. Include the following compiler directive at the beginning of your program.

```
{$L a12drv}
```

B. Declare the three variables for the driver globally, at the beginning of the program.

```
type param_array = array[1..5] of integer;
var params       : param_array;
 task,status  : integer;
```

C. Declare the driver function as external in using a prototype declaration

```
procedure a12drv(task:word; param:word; status:word);external;
```

D. Make your assignment to these variables as desired for the function you wish to perform. See DRIVER REFERENCE section for details on each task.

E. Make the call to the driver.

```
a12drv(ofs(task),ofs(params),ofs(status));
```

F. Compile and link the program.

Using the Driver with QuickBASIC

The following procedure will show you how to use the driver with Microsoft QuickBASIC. You may refer to any of the QuickBASIC sample programs for further illustration. The following procedure will allow you to use the driver both in the QuickBASIC environment and from the command line compiler.

A. Declare the three variables for the driver as global.

```
DIM TASK%, STAT%, PARAMS%(5)
```

B. The array dimension statement must be followed by the COMMON SHARED statement for the driver to be able to find the array. Note: Steps A and B are necessary for any array that will be used by the driver. Certain tasks within the driver required the address of a data buffer, so these two steps would need to be performed for those arrays as well.

```
COMMON SHARED PARAM%()
```

C. Now DECLARE the driver routine. This declaration must include a BYVAL statement before the array variable.

```
DECLARE SUB A12DRV(TASK%, BYVAL PARAM%, STAT%)
```

D. Make your assignment to these variables as desired for the function you wish to perform. See Driver Reference section for details on each task.

E. Make the call to the driver. The CALL statement must explicitly pass the offset of the array variable.

```
CALL A12DRV(TASK%, VARPTR(PARAM%(1)), STAT%)
```

F. To use the program and driver in the environment, you must link a Quick Library. Perform the following command from the command line.

```
LINK /Q A12DRV.OBJ,A12DRV.QLB,,BQLB45.LIB; [ENTER]
```

G. Now load the Quick Library when starting the environment.

```
QB /L A12DRV.QLB [ENTER]
```

H. Use the start command from the run menu to execute the program.

I. To prepare an EXE file from the command line, use the following compile and link commands.

```
BC /o YOURPROG;[ENTER]
LINK YOURPROG+A12DRV;[ENTER]
```

Using the Driver with BASIC

The following procedure will show you how to use the driver with most BASIC languages. You may refer to any of the BASIC sample programs for further illustration.

A. Declare the three variables for the driver as global.

```
10      DIM TASK%, STAT%, PARAMS(5)
```

B. Define a segment within memory to load the driver. You must make sure this segment is not used by BASIC or your program.

```
20      DRIVERSEG = &H5000
30      DEF SEG = DRIVERSEG
```

C. Load the driver into memory starting at offset 0 within the defined segment.

```
40      DRIVER = 0
50      BLOAD "a12drv.bin",DRIVER
```

Make your assignment to these variables as desired for the function you wish to perform. See Driver Reference section for details on each task.

D. Make the call to the driver.

```
60      CALL a12drv(TASK%,PARAMS%(1),STATUS%)
```

Chapter 5: AD12-8 Driver Reference

Using the Driver

This chapter provides detailed information on the functions available from the AD12-8 driver. The chapter is divided into four sections, first is a section detailing the use of this driver, second is a task summary, third is the task reference and last is an error code summary.

The Point List Concept

Most functions of this driver work with a point list. The point list is a list of point addresses in the order that you desire to have conversions performed. A point address is a number specifying the channel of the AD12-8 and the AIM-16. The first 16 point addresses (0-15) refer to the AIM-16 channels for the AIM-16 attached to channel 0 of the AD12-8. The second 16 point addresses (16-31) refer to the 16 channels of the AIM-16 attached to channel 1 of the AD12-8, and so on. Thus, with eight single ended A/D channels, a point address may be as large as 127.

If your are using LVDT-8s, then the first eight point address (0-7) refer to the LVDT-8 attached to channel 0 of the AD12-8. The second eight point addresses are not used (8-15). Thus with eight single-ended A/D channels, a point address may still be as large as 127, but there would only be a maximum of 64 LVDT-8 channels.

You may install point addresses into the point list in any order, or with multiple entries for the same point address. For example the order could be 15-12-12-11-9-127-1-1-0 etc. The order that point addresses are installed in the point list is the order in which you call the driver to install them. Each new entry is appended to the end of the list.

A point list index is used by the driver to keep track of which point address is the next to be converted. After each conversion the index is incremented to the next position in the list. When the index reaches the end of the list it is automatically reset to the start of the list. If you desire to set the list index to the start of the list at any time, you may use TASK 11.

The point list is dynamic. During program operation, if you desire to clear the point list and add a different set of points, this is done quite easily using the tasks provided.

The main advantages of a point list are that conversions can be done in any order and the driver takes care of setting the AIM-16 and/or LVDT-8 channel and the AD12-8 channel, as well as gains, linearization and scaling.

Other Software Features

The driver provides the ability to use the programmable gain feature of the AIM-16. You may assign gains to a given point address directly. Each point address may have its own gain code associated with it. This is useful when differing input ranges are desired using the same AIM-16.

The driver also provides the ability to make a function assignment to each individual point address. You may assign a thermocouple curve or a scaling range to a point address. Look up tables are contained in the driver to convert counts to the proper temperature. Reference junction compensation may also be performed.

The AD12-8 combined with the AIM-16 and this driver provide an excellent tool to handle most kinds of data acquisition signals.

Task Summary

TASK 0:	Driver initialization.
TASK 1:	Check A/D operations.
TASK 2:	Fetch gain code for a given point address.
TASK 3:	Fetch point address from the point list.
TASK 4:	Assign gain code to a range of point addresses.
TASK 5:	Assign range of point addresses to the point list.
TASK 6:	Perform conversion of the given point address.
TASK 7:	Perform conversion on next point address in the point list.
TASK 8:	Perform multiple conversions from the point list using polling.
TASK 9:	Perform multiple conversions from the point list using interrupts.
TASK 10:	Function assignments.
TASK 11:	Reset operations.
TASK 12:	Write digital output.
TASK 13:	Read digital input.
TASK 14:	Load counter/timers.
TASK 15:	Read counter/timers.
TASK 16:	Fetch single, designated point address, high performance buffered.
TASK 17:	Fetch Multiple Points (High Performance), point list and gains are used, but function assignments are not.

Task Reference

Task 0: Initialize

This task provides the driver with information on the card setup. This task should be called once at the beginning of the program, before any other tasks are called. If other tasks are called first, they will return error code 1.

Notes

1. This task also calls TASK 1 to test if the card is functioning.
2. Disables all interrupt and counter activity.
3. Initializes the point list to have point addresses for each channel of the AD12-8, with none for the AIM-16 or LVDT-8 (i.e. point addresses 0, 16, 32, 48 .... 112).
4. Initializes the function list for each point address to a gain code of 0 and no functions performed on conversion counts.
5. Information for the setup of the driver may be obtained either from the configuration file or from the parameters passed to the driver. The configuration file is created by using the SETA12 and SETMUX setup programs, or by a word processor/text editor in the non-document mode. The driver will read the configuration file and set up the driver accordingly if params[0] = 0.

Input

params[0]: Type of initialization
- 0 = Automatic initialization using the configuration file, no other parameters are required.
- 1 = Manual initialization using information provided in the passed parameters.

params[1]: Base Address
params[2]: Interrupt (IRQ) level
params[3]: Voltage range, 5 or 10 volt
params[4]: Input polarity
- 0 = Unipolar
- 1 = Bipolar

Output

Data: None

Error Codes:

status = 0: No error.
status = 1: Invalid task number, task > 17 or driver not initialized.
status = 2: Invalid base address, params[0] > 0x3f8 or < 0x200
status = 3: Card does not respond.
status = 15: Voltage range not 5 or 10.
status = 20: Error opening configuration file.
status = 21: Error reading configuration file.
status = 22: Invalid configuration file data.

Example

```
inttask,params[5],status;       /* these are globally declared variables */

task = 0;
params[0] = 1;                  /* manual initialization */
params[1] = 0x300;              /* base address = 300 hex */
params[2] = 3;                  /* use IRQ3 */
params[3] = 5;                  /* voltage range is 5 volts */
params[4] = 1;                  /* bipolar range */
a12drv(FP_OFF(&task),FP_OFF(params),FP_OFF(&status));          /* call the driver */
```

Task 1: Check A/D Operations

This task checks for proper operation of the analog-to-digital converter.

Notes

1. This task is called internally by the driver when TASK 0 is called.

Input

None

Output

Data: None

Error Codes:

status = 0: No error.
status = 1: Invalid task number, task > 17, or driver not initialized.
status = 3: Card does not respond.

Example

```
inttask,params[5],status;    /* these are globally declared variables */

task = 1;
a12drv(FP_OFF(&task),FP_OFF(params),FP_OFF(&status));/* call the driver */
```

Task 2: Fetch Gain Code for a Point Address

Returns a previously assigned gain code for a given point address.

Notes

None.

Input

params[0]: Point address.

Output

Data:

params[1]: Gain code for the given point address.

Error Codes:

status = 0: No error.
status = 1: Invalid task number, task > 17, or driver not initialized.
status = 5: Invalid point address, or index.

Example

```
inttask,params[5],status;      /* these are globally declared variables */

task = 2;
params[0] = 14;                /* fetch gain code for point address 14*/
a12drv(FP_OFF(&task),FP_OFF(params),FP_OFF(&status));          /* call the driver */
```

Task 3: Fetch Point Address for a Point List Index

Returns a previously assigned point address for a given point list index.

Notes

None.

Input

params[0]: Point list index.

Output

Data:

params[1]: Point address for the given point list index.

Error Codes:

status = 0: No error.
status = 1: Invalid task number, task > 17, or driver not initialized.
status = 5: Invalid point address, or index.

Example

```
inttask,params[5],status;   /* these are globally declared variables */

task = 3;
params[0] = 6;/* fetch point address for the 6th point in the point
 list*/
a12drv(FP_OFF(&task),FP_OFF(params),FP_OFF(&status));/* call the driver */
```

Task 4: Assign Gain Code to Range of Point Addresses

Assigns a given gain code to a given range of point addresses.

Notes

1. The first point address of the range must be less than or equal to the last point address. To assign a gain code to a single point address, make the first and last point address equal.
2. These gain code settings are only meaningful if the AIM-16 is being used.
3. The following are the possible gain codes.

Gain Code	AIM-16 Output Range Switch Settings	
	G/2 OFF	G/2 ON
0	GAIN = 1	GAIN = 0.5
1	GAIN = 2	GAIN = 1
2	GAIN = 10	GAIN = 5
3	GAIN = 50	GAIN = 25
4	GAIN = 100	GAIN = 50
5	GAIN = 200	GAIN = 100
6	GAIN = 400	GAIN = 200
7	GAIN = 1000	GAIN = 500
8	Auto Range	

4. A gain code of 8 indicates an auto range channel. When the point address is read, the driver will first read at a gain of 2 (gain code 1), and from this reading, determine the best gain to use for the second reading to achieve the best resolution.

Input

params[0]: First point in point address range.
params[1]: Last point in point address range.
params[2]: Gain code to assign.

Output

Data: None.

Error Codes:

status = 0: No error.
status = 1: Invalid task number, task > 17, or driver not initialized.
status = 5: Invalid point address, or index.
status = 6: Invalid gain code.

Example

```
inttask,params[5],status;      /* these are globally declared variables */

task = 4;
params[0] = 1;                 /* first point address in range*/
params[1] = 15;                /* last point address in range */
params[2] = 3;                 /* gain code of 3 */
a12drv(FP_OFF(&task),FP_OFF(params),FP_OFF(&status));          /* call the driver */
```

Task 5: Assign Point Addresses to the Point List

Assigns a range of point addresses to the point list.

Notes

1. All point addresses added to the point list are appended to the end of the point list, after any that have been previously added, including the default point address. If you desire to start with an empty list, then use TASK 11, subtask 2 to clear the point list first.
2. If the first point address is larger than the last point address, then the driver will install them in descending order.
3. Point addresses that are not on a 16 boundary (0, 16, 32 ,48 etc) are only meaningful if one or more AIM-16s or LVDT-8s are attached.

Input

params[0]: First point address in range.
params[1]: Last point address in range.

Output

Data: None.

Error Codes:

status = 0: No error.
status = 1: Invalid task number, task > 17, or driver not initialized.
status = 4: Point list error, list full.
status = 5: Invalid point address, or index.

Example

```
inttask,params[5],status;          /* these are globally declared variables */

task = 5;
params[0] = 0;                     /* first point address in range*/
params[1] = 31;                    /* last point address in range, two AIM-16s */
a12drv(FP_OFF(&task),FP_OFF(params),FP_OFF(&status));      /* call the driver */
```

Task 6: Fetch Data from a Point Address

Perform a conversion on the point address indicated.

Notes

1. This task does not use the point list. If you wish to fetch data from the next point in the point list then use TASK 7.
2. Point addresses that are not on a 16 boundary (0, 16, 32 ,48 etc) are only meaningful if the AIM-16 or LVDT-8 is being used.

Input

params[0]: Point address to fetch data from.

Output

Data:

params[1]: Resulting conversion
params[2]: Gain code used.

Error Codes:

status = 0: No error.
status = 1: Invalid task number, task > 17, or driver not initialized.
status = 3: Card does not respond.
status = 5: Invalid point address, or index.

Example

```
inttask,params[5],status;          /* these are globally declared variables */

task =6;
params[0] = 16;                    /* fetch data from point address */
a12drv(FP_OFF(&task),FP_OFF(params),FP_OFF(&status));      /* call the driver */
```

Task 7: Fetch Data from next Point Address in List

Perform a conversion on the point address in the point list indicated by the point list index.

Notes

1. Each time a point is fetched from the list, the list index is incremented. The list index can be reset to the start of the point list by using TASK 11, SUBTASK 1

Input

None.

Output

Data:

params[0]: Point address converted.
params[1]: Resulting conversion data.
params[2]: Gain code used.

Error Codes:

status = 0: No error.
status = 1: Invalid task number, task > 17, or driver not initialized.
status = 3: Card does not respond.
status = 5: Invalid point address, or index.

Example

```
inttask,params[5],status;    /* these are globally declared variables */

task = 7;
a12drv(FP_OFF(&task),FP_OFF(params),FP_OFF(&status));/* call the driver */
```

Task 8: Fetch Multiple Buffered Conversions

Fetch multiple conversions from the point list, using polling.

Notes

1. Each time a point is fetched from the list, the list index is incremented. The list index can be reset to the beginning of the point list by TASK 11, SUBTASK 1.
2. This task uses two buffers, a data buffer and a point/gain buffer. Both buffers should be integer buffers of the same length. The number of conversions must not exceed the length of the shortest buffer, or else other areas of computer memory may be corrupted, causing unpredictable computer behavior. The driver has no criteria to evaluate the validity of the pointer. It is incumbent upon the application program to supply a valid buffer pointer.
3. The point and gain for each analog input is returned in the point/gain buffer. The point address and gain are packed into one integer with the point address in the upper eight bits and the gain in the lower eight bits.
4. The buffers must be declared globally or the driver will not be able to find their segment.

Input

params[0]: Offset of the data buffer address.
params[1]: Offset of the point/gain buffer address.
params[2]: Number of conversions to make.

Output

Data:

params[3]: Number of conversions completed.
The buffers will contain the conversions and the point/gain data respectively.

Error Codes:

status = 0: No error.
status = 1: Invalid task number, task > 17, or driver not initialized.
status = 3: Card does not respond.
status = 5: Point list error, list empty.

Example

```
inttask,params[5],status;          /* these are globally declared variables */
intdatbuf[100],chnbuf[100];        /* these are globally declared variables */

task = 8;
params[0] = FP_OFF(datbuf);        /* pass offset of data buffer */
params[1] = FP_OFF(chnbuf);        /* pass offset of point/gain  buffer */
params[2] = 100;                   /* number of conversions */
a12drv(FP_OFF(&task),FP_OFF(params),FP_OFF(&status));      /* call the driver */
```

Task 9: Interrupt Driven Data Acquisition

Provides subtasks to perform buffered data acquisition using interrupts. Sub task functions include initiating the interrupt conversions, checking for completion and stopping the interrupt process.

Notes

1. Each time a point is fetched from the list, the list index is incremented. The list index can be reset to the beginning of the point list by TASK 11.
2. The initial call to SUBTASK 1 passes a pointer to a data buffer to store the conversion. This buffer is used as temporary storage by the driver. A subsequent call to SUBTASK 3 will place the data into two separate buffers; a data buffer for the conversion results and a point/gain buffer to contain the point address and the gain used for the conversion.

3. The point and gain for each analog input is returned in the point/gain buffer. The point address and gain are packed into one integer with the point address in the upper eight bits and the gain in the lower eight bits. Both the data buffer and the point/gain buffer should be integer buffers of the same length. The number of conversions must not exceed the length of the shortest buffer or else other areas of computer memory may be corrupted causing unpredictable computer behavior. The driver has no criteria to evaluate the validity of the pointer. It is incumbent upon the application program to supply a valid buffer pointer.
4. The data and point/gain buffers must be declared globally or the driver will not be able to find their segment.
5. This task has several functions, each having its own required parameters.
6. If the timers are used to generate the start-conversion signals, then they should be configured using TASK 14, before calling TASK 9.
7. TASK 11, SUBTASK 0 is used to disable interrupts before completion of the scan. When the scan completes normally, the interrupts are disabled automatically.
8. See Sample 2 for a basic example of how to set up and use interrupts.

Input

params[0]: Subtask to perform, 1, 2, or 3.

- 1: Initiate interrupt data acquisition.
  - params[1]: Interrupt level (IRQ)
  - params[2]: Number of conversion to make.
  - params[3]: Offset of the packed data buffer address.
  - params[4]: Segment of the packed data buffer address.
  - params[5]: A/D Trigger mode.
- 2: Check for end of interrupt scan.
- 3: Return unpacked data
  - params[1]: Offset of data buffer.
  - params[2]: Offset of point/gain buffer
  - params[3]: Number of conversions made

Output

Data:

SUBTASK 1: The buffers will contain the conversions and the point/gain data respectively.
SUBTASK 2: params[1] = 0 if scan complete, task number if still in progress.

Error Codes:

status = 0: No error.
status = 1: Invalid task number, task > 17, or driver not initialized.
status = 3: Card does not respond.
status = 5: Point error, point list is empty.
status = 7: Invalid number of conversions, not between 1 and 32767.
status = 10: Interrupt task already active.
status = 11: Interrupt not between 2 and 7.
status = 12: Interrupt already unassigned. (SUBTASK 3)
status = 13: Invalid subtask, not 1, 2 or 3.
status = 14: Invalid trigger mode, not 1 or 2.

Example

```
inttask,params[5],status;                    /* these are globally declared variables */
intdatbuf[100],chnbuf[100];                  /* these are globally declared variables */

task = 9;
params[0] = 1;                               /* initiate interrupt scan */
params[1] = 100;                             /* do 100 conversions on this scan */
params[2] = FP_OFF(datbuf);                  /* pass offset of temporary data buffer */
params[3] = FP_SEG(datbuf);                  /* pass segment of buffer */
a12drv(FP_OFF(&task),FP_OFF(params),FP_OFF(&status));        /* call the driver */
params[0] = 2;                               /* check for end of scan process */
do
  {
a12drv(FP_OFF(&task),FP_OFF(params),FP_OFF(&status));    /* call the driver */
}
while (params[1] != 0);                      /* wait until end of scan*/
                          /* or if you do not want to wait until end of scan */
task = 11
params[0] = 0;                               /* stop interrupt process sub task */
a12drv(FP_OFF(&task),FP_OFF(params),FP_OFF(&status));/* call the driver */
```

Task 10: Thermocouple/Function Assignment

Provides subtasks to assign thermocouple curves and scaling factors to a given point address. A subtask is also provided to use the thermocouple tables to linearize values passed to it.

Notes

1. The built-in NBS tables are designed to convert A/D counts to temperature directly. When using SUBTASK 1, the driver expects the passed counts to be multiplied by 16 if using the bipolar mode, and multiplied by 8 if using a unipolar mode.
2. Curves are assigned to a point address by calling SUBTASK 2 with the ASCII code of one of the curves listed in the table that follows. Also, temperature units are assigned in the same manner.
3. If reference junction compensation using the AIM-16 on board sensor is desired, then assign this sensor to the point list as the first channel of a given AIM-16 (ie. 0, 16,32,48 etc). Make sure that the TMP jumpers are installed on the AIM-16. Finally, assign the curve "T" to this point address using SUBTASK 2. Any other point addresses on the AIM-16 will now be junction compensated automatically by the driver each time a point address is converted.
4. The reference junction circuit on the AIM-16 card generates 24.4 mV/ °C. The counts read in at a gain of 1 are 2.44 millivolts/count. Thus, each count represents 0.1 °C.

5. When thermocouple curves are assigned to a point address, it is also required to set that point address to a particular gain using TASK 4. These gains are presented in the following table. Note that two gain codes are presented for each thermocouple type, the one you use will depend on the setting of the G/2 switch on the AIM-16. If G/2 is OFF, use the lower gain code, if G/2 is ON then use the higher gain code.

T/C Type	Gain	Gain Code	µVolts/Count
b	200	5/6	12.207
e	50	3/4	48.828
j	100	4/5	24.414
k	50	3/4	48.828
r	200	5/6	12.207
s	200	5/6	12.207
t	200	5/6	12.207
RTD Type	**Gain**	**Gain Code**	**µVolts/Count**
a	100	4/5	24.414
u	100	4/5	24.414

6. Temperature is returned in increments of 1/10th degree. For example, 100 degrees would be returned as 1000.
7. SUBTASK 3 can be used to force the driver to return values in units determined by the user rather than counts. For example, you might desire values returned in millivolts. In such a case, assuming the bipolar mode, scale factors of -5000 and +5000 would be passed in the call to SUBTASK 3.
8. For platinum RTDs, there are two curves; "a" for sensors with 392 alpha and "u" for sensors with 385 alpha.
9. TASK 10 does not initiate any conversions, but sets up functions that will be performed automatically whenever conversion are done using tasks 6, 7, 8, 9 or 16.

Input

params[0]: Subtask to perform, 1, 2, 3 or 4.

1: Perform linearization of the given data.
   - params[1]: ASCII code for lower case letter of curve, or upper case T for reference junction.
   - params[2]: counts (see note 1)

2: Assign thermocouple curve to a point address.
   - params[1]: point address
   - params[2]: ASCII code for lower case letter of curve, or upper case T for reference junction.
   - params[3]: ASCII code for upper case letter of the desired temperature units, C or F.

3: Assign scaling factor to a point address.
params[1]: point address
params[2]: Lower scaling term.
params[3]: Upper scaling term.

4: Replicate a point address function assignment to a range of point addresses.
params[1]: source address to replicate
params[2]: first point address in destination range
params[3]: last point address in destination range.

Output

Data:

SUBTASK 1:
params[3]: temperature in tenths of °F.
params[4]: temperature in tenths of °C.

Error Codes:

status = 0: No error.
status = 1: Invalid task number, task > 20, or driver not initialized.
status = 5: Point error, point address out of range.
status = 13: Invalid subtask, not between 1 and 4.
status = 16: Invalid curve.

Example

```
inttask,params[5],status;   /* these are globally declared variables */

task = 10;
/*  linearize the passed value */
params[0] = 1;/* manual linearization subtask */
params[1] = 't';/* for t type thermocouple */
params[2] = 1801;/* counts * 16 at gain of 200 */
a12drv(FP_OFF(&task),FP_OFF(params),FP_OFF(&status));/* call the driver */
/* values returned in params[3] and params[4] */
/* assign curve to a point address */
params[0] = 2;/* curve assignment  subtask */
params[1] = 0;/* first point address on first AIM-16 */
params[2] = 'T';/* T for reference junction */
params[3] = 'F';/* F for degrees F */
a12drv(FP_OFF(&task),FP_OFF(params),FP_OFF(&status));/*call the driver*/
```

```
/* assign a range of ±5 volts in millivolt increments to a point address.
params[0] = 3;/* range assignment subtask */
params[1] = 22;/* point address to assign */
params[2] = -5000;/* lower range */
params[3] = 5000;/* upper range */
a12drv(FP_OFF(&task),FP_OFF(params),FP_OFF(&status));/* call the driver */

/* replicate the assignment for point address 22 to point addresses 23-40 */
params[0] = 4;/* replication  subtask */
params[1] = 22;/* source point address to replicate */
params[2] = 23;/* lower point address in destination range */
params[3] = 40;/* upper point address in destination range */
a12drv(FP_OFF(&task),FP_OFF(params),FP_OFF(&status));/* call the driver */
```

Task 11: Reset Functions

Performs various reset function on the point list and function/curve assignment tables. Also provides a subtask to set the sample and hold settle time.

Notes

1. SUBTASK 5 is provided to set the sample and hold settle time. High speed 80286, 80386, and 80486 computers often will start a conversion before the sample and hold has had time to settle after changing a channel on the AIM-16. A value of 25-50 is usually sufficient for an 80386 machine.

Input

params[0]: Subtask to perform, 1, 2, 3, 4 or 5.
- 1: Reset the point list index to first point address in the point list.
- 2: Clears all point addresses from the point list.
- 3: Resets the point list to the default conditions, as described in TASK 0.
- 4: Clears all curve and scaling assignments.
- 5: Set the sample and hold settle time.

params[1]: settle time count

Output

Data: None.

Error Codes:

status = 0: No error.
status = 1: Invalid task number, task > 17, or driver not initialized.
status = 15: Invalid reset sub task, not between 0 and 5.

Example

```
inttask,params[5],status;   /* these are globally declared variables */

task = 11;
params[0] = 5;/* set settle time  sub task */
params[1] = 50;/* settle time count of 50 */
a12drv(FP_OFF(&task),FP_OFF(params),FP_OFF(&status));/* call the driver */
```

Task 12: Digital Output

Writes to the digital output bits.

Notes

Output values are not checked for proper range. If a value greater than eight bits is sent, the driver will only send the lower eight bits.

Input

params[0]: Value to output.

Output

Data: None.

Error Codes:

status = 0: No error.
status = 1: Invalid task number, task > 17, or driver not initialized.

Example

```
inttask,params[5],status;   /* these are globally declared variables */

task = 12;
params[0] = 15;/* set first  4 output bits high */
a12drv(FP_OFF(&task),FP_OFF(params),FP_OFF(&status));/* call the driver */
```

Task 13: Digital Input

Reads the digital input bits.

Notes

1. Returns the eight bit digital value read from the card digital input register.

Input

None.

Output

Data: params[1]: Digital input value.

Error Codes:

status = 0: No error.
status = 1: Invalid task number, task > 17, or driver not initialized.

Example

```
inttask,params[5],status;   /* these are globally declared variables */

task = 13;
a12drv(FP_OFF(&task),FP_OFF(params),FP_OFF(&status));/* call the driver */
```

Task 14: Counter/Timer Setup

Load the given counter/timer with a count value and mode.

Notes

1. For a complete discussion of the counter/timers, see Programmable Interval Timer.

Input

params[0]: counter number 0, 1 or 2.
params[1]: counter mode, between 0 and 5.
params[2]: counter load count.

Output

Data: None.

Error Codes:

status = 0: No error.
status = 1: Invalid task number, task > 17, or driver not initialized.
status = 8: Invalid counter, not 0, 1 or 2.
status = 9: Invalid counter mode, not between 0 and 5.

Example

```
inttask,params[5],status;   /* these are globally declared variables */

task = 14;
params[0] = 1;/* counter 1 */
params[1] = 3;/* counter mode 3, square wave generator */
params[2] = 100;/* counter load value, acts as divide by 100 */
aa16drv(FP_OFF(&task),FP_OFF(params),FP_OFF(&status));/* call the driver */
```

Task 15: Read Counter/Timer Count

Reads the count of the given counter/timer.

Notes

1. For a complete discussion of the counter/timers, see Programmable Interval Timer.
2. Counter/timer is latched before read.

Input

params[0]: counter number 0, 1 or 2.

Output

Data: params[1]: counter/timer count.

Error Codes:

status = 0: No error.
status = 1: Invalid task number, task > 17, or driver not initialized.
status = 8: Invalid counter, not 0, 1 or 2.

Example

```
inttask,params[5],status;   /* these are globally declared variables */

task = 15;
params[0] = 1;/* counter 1 */
a12drv(FP_OFF(&task),FP_OFF(params),FP_OFF(&status));/* call the driver */
```

Task 16: High Speed Conversions, Single Point Address

Fetch multiple conversions a single point.

Notes

1. This task will use the gain set up in TASK 4, or the function assignments set up in TASK 10.
2. This task uses a single data buffer. The number of conversions must not exceed the length of the buffer, or else other areas of computer memory may be corrupted, causing unpredictable computer behavior. The driver has no criteria to evaluate the validity of the pointer. It is incumbent upon the application program to supply a valid buffer pointer.
3. The buffer must be declared globally or the driver will not be able to find its segment.

Input

params[0]: Offset of the data buffer address.
params[1]: Number of conversions to make.
params[2]: Point address to convert.

Output

Data:

params[3]: Number of conversions completed.
The buffer will contain the conversions.

Error Codes:

status = 0: No error.
status = 1: Invalid task number, task > 17, or driver not initialized.
status = 3: Card does not respond.

Example

```
inttask,params[7],status;   /* these are globally declared variables */
intdatbuf[100],chnbuf[100];   /* these are globally declared variables */

task = 16;
params[0] = FP_OFF(datbuf);/* pass offset of data buffer */
params[1] = 100;/* number of conversions */
params[2] = 5;/* channel 5 on the first AIM-16 */
a12drv(FP_OFF(&task),FP_OFF(params),FP_OFF(&status));/* call the driver */
```

Task 17: High Speed Conversions, Multiple Point Addresses

Fetch multiple conversions from the point list using pipelining features of the AD12-8.

Notes

1. The point list is used to determine which point addresses to convert.
2. This task will use the gain set up in TASK 4, but will not use the function assignments set up in TASK 10.
3. This task uses two buffers, a data buffer and a point/gain buffer. Both buffers should be integer buffers of the same length. The number of conversions must not exceed the length of the shortest buffer, or else other areas of computer memory may be corrupted, causing unpredictable computer behavior. The driver has no criteria to evaluate the validity of the pointer. It is incumbent upon the application program to supply a valid buffer pointer.
4. The point and gain for each analog input is returned in the point/gain buffer. The point address and gain are packed into one integer with the point address in the upper eight bits and the gain in the lower eight bits.
5. The buffers must be declared globally or the driver will not be able to find their segment.
6. Using this task, a 25MHz "386" computer will achieve throughput approaching 34,000 samples per second.

Input

params[0]: Offset of the data buffer address.
params[1]: Offset of the point/gain buffer address.
params[2]: Number of conversion to make.

Output

Data:

params[3]: Number of conversions completed.
The buffers will contain the conversions and the point/gain data respectively.

Error Codes:

status = 0: No error.
status = 1: Invalid task number, task > 17, or driver not initialized.
status = 3: Card does not respond.
status = 5: Point list error, list empty.

Example

```
inttask,params[5],status;   /* these are globally declared variables */
intdatbuf[100],chnbuf[100];   /* these are globally declared variables */

task = 16;
params[0] = FP_OFF(datbuf);/* pass offset of data buffer */
params[1] = FP_OFF(chnbuf);/* pass offset of point/gain  buffer */
params[2] = 100;/* number of conversions */
a12drv(FP_OFF(&task),FP_OFF(params),FP_OFF(&status));/* call the driver */
```

Summary of Error Codes

1. Invalid task number: The task number does not fall within the range of 0 through 17. This error code also occurs if any task is selected before a successful initialization with TASK 0.
2. Invalid base address: The base I/O address does not fall within the range of 100 hex through 3F8 hex.
3. A/D failed: The EOC (end-of-conversion) signal did not change state. This is usually because the base address has not been set properly.
4. Point list is full: The point list can only hold 128 entries.
5. Invalid point address: The point address does not fall within the range of 0 through 127.
6. Invalid gain code: The gain code does not fall within the range of 0 through 8.
7. Invalid TASK 11 subtask: The subtask does not fall within the range of 0 through 5.
8. Invalid counter/timer number: The counter/timer number is not 0, 1 or 2.
9. Invalid mode: The counter/timer mode does not fall within the range of 0 through 5.
10. Interrupt process already set: The interrupt process can only be set up once. A subsequent request has been made to set up the interrupt process without previously resetting the mode.
11. Invalid interrupt number: The interrupt number does not fall within the range of 2 through 7.
12. No interrupt process is set: A request has been made to reset an interrupt process that has not previously been set.
13. Invalid SUBTASK number: SUBTASK specified is outside the valid range for a given task.
14. Invalid data buffer: Data buffer is not valid for interrupt driven data acquisition.
15. Invalid voltage range: Voltage range specified should be either 5 or 10 volts.
16. Invalid curve specified: The curve code letter specified is not a valid code.
17. Invalid A/D channel number.
18. Invalid temperature units: Temperature units not degrees F or C.
19. Unused.
20. Error opening configuration file.
21. Error reading configuration file.
22. Configuration file data are not correct.

Chapter 6: AD12-8 Windows Driver Reference

Driver Features

The Windows driver is implemented as a dynamic-link-library(DLL). There is no need to explicitly link this driver to your application, this linkage is performed at run-time by Windows.

There is a single point list in the driver. Thus, your point list can read points in the system in any desired order. For more information on point lists, read the section in entitled The Point List Concept.

This driver supports thermocouples of types "B","E","J","K","R","S" and "T". (The driver also supports both 385 and 392 alpha RTDs.) You may use these thermocouples at any gain supported by the AIM-16 and the driver will compensate accordingly. Also, support for the AIM-16 reference junction is provided. The reference junction may be assigned to channel 0 of its AIM-16 or to a dedicated AD12-8 channel. If you assign the reference junction to a discrete AD12-8 channel, the driver provides the means to indicate which AIM-16 to apply the reference compensation.

You may find using the auto-configuration feature of the driver helpful and time saving. Use the setup program provided to configure the card the way you intend to use it, then use auto-configuration when you initialize the driver. All points will be set up for gain and curves, and, installed in the point list.

Using the Driver

The following are the steps that you should take to use the driver with your application. The driver has been tested with Microsoft Visual BASIC, version 3.0 and Borland C/C++, version 3.1 compilers. The discussion that follows is intended for those compilers, but the principles should apply to any compiler that uses the Pascal calling convention.

Tell Your Application about the Driver

The application needs to know the function prototypes of the routines in the driver that you will use.

In C or C++, the function prototype will look like the example below.

```
extern "C"  int PASCAL AD128_Init(int config,int addr,int range);
```

The extern "C" portion of the declaration tells the compiler not to use the C++ type of function call. C++ and C function calls are not compatible. A file that defines all function prototypes for the driver, as well as useful constants, is included in the CPPWIN directory. You may include this file in your own application. Place the line of code below in the main global module of your program.

```
#include "A12DRV.H"
```

In Borland C/C++, you must also add the file A12DRV.LIB to your project file. See your documentation for creating and managing a project file.

In Visual BASIC, all that is needed is the function prototype which will look like the example below.

```
Declare Function AD128_Init Lib "c:\ad128s\vbwin\a12drv.dll" (ByVal config As Integer, ByVal
BaseAddr As Integer, ByVal range As Integer, card As Integer) As Integer
```

The above function prototype is written on multiple lines in this manual but, in Visual BASIC, the entire function prototype must appear on a single line. A file is included in the VBWIN directory that declares all functions and provides useful constants. Add the file A12DRV.GBL to your Visual BASIC project file. See your Visual BASIC manuals for information on project files.

You should place a copy of the A12DRV.DLL file in your working directory. In Visual BASIC, the function prototypes contain a path name so that Windows will know where to find the DLL. The A12DRV.GBL file function prototypes are configured for the default installation directory structure. If you change the defaults, then you must edit the A12DRV.GBL file to reflect the location of the DLL.

Initialize the Driver

Now that the application knows about the driver, you must initialize the driver for operation. This is done with a call to the AD128_Init function. The driver may not work properly if you make any other driver calls before the AD128_Init call.

Setup the Configuration for All Points in Your System

Each point in the system must be configured using the AD128_SetPointConfig call. This call allows you to set various options for conversions, such as scaling, curves and gains. If you need to set up a sequential series of points with the same configuration, a single call will suffice, using the start and stop parameter to define the range of points. The driver will return an error if you attempt to convert a point before setting it up with a call to AD128_SetPointConfig.

Install the Desired Points into the Point List

There is a single point list in the system. The point list will still allow you to install points in any order you desire.

Use of the point list is optional. Conversions may be made by direct access to any point in the system. Also, if the point list is used, direct access to a given point is still available. Points are added the point list by calls to AD128_AddPoints. This routine will append the range of points indicated to the end of the existing point list, or if the point list is empty, create a point list.

Perform Conversions

Several routines are provided to perform A/D conversions. AD128_GetDirectPoint is used for direct conversions, without the point list, to convert a single designated point address. AD128_GetIndexPoint performs a single conversion on a given point index. The point index indicates which point in the list to convert. AD128_GetNextPoint performs a single conversion on the next point in the point list, then increments the point list index. The next call to AD128_GetNextPoint will then convert the next point.

AD128_PollScan converts a series of points from the point list and stores the conversions in an internal buffer. A subsequent call to AD128_PostProcess will process the data for curve and scaling functions and then transfer them to a buffer provided by your program. Post processing is used to increase the system throughput.

AD128_IRQScan is similar to AD128_PollScan except that interrupts are generated by the AD12-8 at the end of each conversion, rather than using a polling methodology to determine the end-of-conversion.

Use of Other Functions

Other functions can be called to control the digital bits and counters in the same manner as calls to the A/D routines.

Task Summary

AD128_Init:	Initializes the driver.
AD128_Shutdown:	Terminates the driver and frees all Windows resources.
AD128_SetPointConfig:	Configures a range of points.
AD128_FetchPointConfig:	Fetches the configuration for a given point.
AD128_AddPoints:	Adds a range of points to the point list.
AD128_ResetListIndex:	Sets the point list index to the top of the list.
AD128_ClearPointList:	Clears all points from the point list.
AD128_DelPtListIndexes:	Deletes range of point list indexes from the point list.
AD128_SetSettleTime:	Set the sample-and-hold settle time.
AD128_GetNextPoint:	Converts the next point in the list as indicated by the list index.
AD128_GetIndexPoint:	Converts a point in the list based on a provided list index.
AD128_GetDirectPoint:	Converts a point directly using the supplied point address.
AD128_IRQStatus:	Checks the status of an active interrupt process.
AD128_IRQTerminate:	Terminates an active interrupt process.
AD128_IRQScan:	Performs buffered data acquisition using interrupts.
AD128_PollScan:	Performs buffered data acquisition using polling.
AD128_PostProcess:	Processes and returns data from a batch process.
AD128_FetchLastGain:	Fetches the last gain used for conversion of a given point.
AD128_DigitalOut:	Writes to the digital output bits.
AD128_DigitalIn:	Read from the digital input bits.
AD128_SetCounter:	Sets up a given counter.
AD128_ReadCounter:	Reads back a given counter.
AD128_RateGenerator:	Generates a given frequency from the counter 1/2 pair.
AD128_DisableCounter:	Disables a given counter.
AD128_MeasureFreq:	Measures frequency.
AD128_MeasurePeriod:	Measures period.

Task Reference

AD128_Init

Function: Initializes the driver, and sets up the data structures.

Syntax:

Visual BASIC: AD128_Init(Byval config as Integer, Byval addr as Integer, Byval range as Integer,) as Integer

C: int PASCAL AD128_Init(int config,int addr,int range);

config: If config = 0, the calling program will configure channel information for the card by making calls to the driver functions to set up the point list, gains, functions etc. If config = 1, the configuration information is fetched from the file called setup.cfg.

addr: The base address for this card. The base address should be between 200 and 3F8 hex.

range: If range = 0, the voltage range is bipolar 5 volts, if range = 1, bipolar 10 volts or if range = 2, unipolar 5 volt.

Notes: This function needs to be called once.

Error Codes:

INVALID_PTR:	The card_number pointer was not valid.
CONFIG_CODE:	The auto-config parameter was not 0 or 1.
BASE_ADDRESS:	The base address is not >= 200 and <= 3f8.
VOLTAGE_RANGE:	The voltage range was not 1,2 or 3.
WINDOWSERROR:	Problem allocating Windows memory.
CARD_INACTIVE:	The card did not respond to a test.
AD128_SUCCESS:	Operation was performed without error.

AD128_Shutdown

Function: Releases all data structures used by the driver.

Syntax:

Visual BASIC: AD128_Shutdown() as Integer

C: int PASCAL AD128_Shutdown();

Notes: This function needs only to be called once, immediately before your programs exists.

Error Codes: AD128_SUCCESS: Operation was performed without error.

AD128_SetPointConfig

Function: Initializes a range of points for curve, scaling, gain and units.

Syntax:

Visual BASIC: AD128_SetPointConfig(Byval start as Integer,Byval stop as Integer, Byval curve as Integer, Byval low as Single,Byval hi as Single, Byval units as Integer, Byval gain as Integer, Byval ref_channel as Integer) as integer

C: int PASCAL AD128_SetPointConfig(int start,int stop,int curve float low, float hi,int units,int gain,int ref_channel);

start: Starting point address in the range of points.

stop: Ending point address in the range of points.

curve: An ASCII "b","e","j","k","r","s","t" indicates that the respective points are to be installed for that particular thermocouple type. A "T" indicates that the point (s) are for reference junction compensation. A "a" or "u" represents 392 and 385 alpha type RTD's respectively. If no curve is desired, the curve variable should be zero.

low: Lower value in the scaling range.

hi: Upper value in the scaling range.

units: A zero indicates oC and a one indicates oF.

gain: Gain code for the point range. The chart below lists the possible gain codes and their respective gains.

Gain Code	AIM-16 Output Range Switch Settings	
	G/2 OFF	G/2 ON
0	GAIN = 1	GAIN = 0.5
1	GAIN = 2	GAIN = 1
2	GAIN = 10	GAIN = 5
3	GAIN = 50	GAIN = 25
4	GAIN = 100	GAIN = 50
5	GAIN = 200	GAIN = 100
6	GAIN = 400	GAIN = 200
7	GAIN = 1000	GAIN = 500
8	Auto Range	

ref_channel: If the curve type is "T" for reference junction compensation, the A/D channel that the reference junction compensation should be applied to is passed here. For example if a zero is passed for this parameter, then all AIM-16 channels attached to AD12-8 channel 0 will be compensated, provided they are set up as thermocouples.

Notes: If no scaling is desired, than pass zero for the hi and low parameters.

If a curve is desired, then it is recommended that you do not use scaling. If values are passed for hi and low, then the driver will scale the results, after the temperature conversion.

The bipolar 5 volt range is recommended for thermocouples.

The following chart indicates the recommended gain for each thermocouple type. You are not restricted to these values, the driver will compensate for any gain code you desire. If you do not use the recommended gains however, a small round off error may result.

The auto-range gain code causes the driver to take an initial sample at a gain of two, then from the result, determine the best gain to use to obtain the best resolution.

T/C Type	Gain	Gain Code	µVolts/Count
b	200	5/6	12.207
e	50	3/4	48.828
j	100	4/5	24.414
k	50	3/4	48.828
r	200	5/6	12.207
s	200	5/6	12.207
t	200	5/6	12.207
RTD Type	**Gain**	**Gain Code**	**µVolts/Count**
a	100	4/5	24.414
u	100	4/5	24.414

Error Codes:

POINT_ERROR:	One or both point addresses are > 127.
CHANNEL_ERROR:	The reference channel is not from zero to eight.
GAIN_ERROR:	The gain code is not from zero to eight.
UNIT_ERROR:	The units parameter is not zero or one.
CURVE_ERROR:	The curve parameter is not valid.
AD128_SUCCESS:	Operation was performed without error.

AD128_FetchPointConfig

Function: Returns the values previously set up for a given point.

Syntax:

Visual BASIC: AD128_FetchPointConfig(Byval addr as integer, curve as Integer, low as Single, hi as Single, units as Integer, gain as Integer) as Integer

C: int PASCAL AD128_FetchPointConfig(int addr,int *curve, float *low, float *hi, int *units,int *gain);

addr: Point address of the desired point configuration.
*curve: An integer point to return the ASCII code of the curve.
*low: Float pointer to return lower scaling value.
*hi: Float pointer to return upper scaling value.
*units: Integer pointer to return temperature units. A zero indicates degrees C and a one indicates degrees F.
*gain: Integer to return gain code.

Notes: None.

Error Codes: INVALID_PTR: One or more of the return pointers is invalid.

AD128_AddPoints

Function: Adds a range of points to the point list.

Syntax:

Visual BASIC: AD128_AddPoints(Byval start as Integer, Byval stop as Integer)as Integer

C: int PASCAL AD128_AddPoints(int start,int stop);

start: Beginning point address of the range.
stop: Ending point address of the range.

Notes: If the starting point address is greater than the stop point address, the points are installed into the point list in reverse order.

Error Codes:

POINT_ERROR: start or stop is not between 0 and 128.
WINDOWSERROR: A memory error occurred in Windows.
AD128_SUCCESS: Operation was performed without error.

AD128_ResetListIndex

Function: Resets the point list index pointer to the first point in the point list.

Syntax:

Visual BASIC: AD128_ResetListIndex() as Integer

C: int PASCAL AD128_ResetListIndex();

Notes: This function is relevant to programs that are performing one conversion from the point list at a time, rather than a batch process such as polling. Single conversion routines that use the point list use the point list index to determine the next point to convert.

Error Codes: AD128_SUCCESS: Operation was performed without error.

AD128_ClearPointList

Function: Removes all points from the point list, all point list pointers are set to NULL and all memory used by the point list is freed.

Syntax:

Visual BASIC: AD128_ClearPointList() as Integer

C: int PASCAL int PASCAL AD128_ClearPointList();

Notes: None.

Error Codes: AD128_SUCCESS: Operation was performed without error.

AD128_DelPtListIndexes

Function: Removes all points from the point list, between a start and stop point list index.

Syntax:

Visual BASIC: AD128_DelPtListIndexes(Byval start as Integer, Byval stop as Integer) as Integer

C: int PASCAL AD128_DelPtListIndexes(int start,int stop);

start: Starting point in the point list index range.

stop: Ending point in the point list index range.

Notes: The start and stop parameters are NOT point addresses, but indexes into the point list. If start is 5 and stop is 10, then the fifth through tenth point in the list will be removed.

Error Codes: AD128_SUCCESS: Operation was performed without error.

AD128_SetSettleTime

Function: Sets the sample-and-hold settle time delay for the AIM-16.

Syntax:

Visual BASIC: AD128_SetSettleTime(Byval settle as Integer) as Integer

C: int PASCAL AD128_SetSettleTime(unsigned settle);

settle: Number of settle time counts.

Notes: The sample-and-hold settle time is primarily intended for when an AIM-16 is used in the system. When there is to be a significant change for the gain used in the preceding conversion, the sample-and-hold circuitry many need extra time to sample. This is normally in cases when the gains vary widely from channel-to-channel. Symptoms of a need for this extra time are when the values returned by the driver seem to be close, but not as accurate as expected. To determine the amount of extra time needed, start with 25 and vary the number that yields the expected accuracy. The number should be as small as practicable so as not to adversely affect throughput of the system.

Error Codes: AD128_SUCCESS: Operation was performed without error.

AD128_GetNextPoint

Function: Performs an A/D conversion on the next point in the point list, and increments the point list index to the next position in the point list.

Syntax:

Visual BASIC: AD128_GetNextPoint(addr as Integer, result as Single) as Integer

C: int PASCAL AD128_GetNextPoint(int *addr,float *result);

*addr: Pointer for the driver to return the point address of the converted point.

*result: A floating point pointer to return the results of the conversion.

Notes: The value returned has all curve functions and scaling applied to the result.

Error Codes:

INVALID_PTR:	One or more of the return pointers is invalid.
LIST_EMPTY:	The point list has not points in it.
POINT_UNINSTALL:	The next point in the point list has not been installed with a call to AD128_SetPointConfig().
CARD_INACTIVE:	The card does not respond, no end-of-conversion.
AD128_SUCCESS:	Operation was performed without error.

AD128_GetIndexPoint

Function: Performs an A/D conversion on a given point in the point list.

Syntax:

Visual BASIC: AD128_GetIndexPoint(Byval index as Integer, addr as Integer, result as Single) as Integer

C: int PASCAL AD128_GetIndexPoint(int index,int *addr,float *result);

index:	The location of the point in the point list to convert.
*addr:	Pointer for the driver to return the point address of the converted point.
*result:	A floating point pointer to return the results of the conversion.

Notes: The value returned has all curve functions and scaling applied to the result.

The index parameter represents a position in the point list rather than a point address. For instance, if a 5 is passed, then the fifth entry in the point list is converted.

Error Codes:

INVALID_PTR:	One or more of the return pointers is invalid.
LIST_ERROR:	The point location was not found.
POINT_UNINSTALL:	The next point in the point list has not been installed with a call to AD128_SetPointConfig().
CARD_INACTIVE:	The card does not respond, no end-of-conversion.
AD128_SUCCESS:	Operation was performed without error.

AD128_GetDirectPoint

Function: Performs an A/D conversion on a given point address.

Syntax:

Visual BASIC: AD128_GetDirectPoint(Byval addr as Integer, result as Single) as Integer

C: int PASCAL AD128_GetDirectPoint(int addr,float *result);

addr:	The point address of the point to convert.
*result:	A floating point pointer to return the results of the conversion.

Notes: The value returned has all curve functions and scaling applied to the result.
The point list is not used. The routine converts the point address passed.

Error Codes:

INVALID_PTR:	One or more of the return pointers is invalid.
POINT_UNINSTALL:	The next point in the point list has not been installed with a call to AD128_SetPointConfig().
CARD_INACTIVE:	The card does not respond, no EOC.
AD128_SUCCESS:	Operation was performed without error.

AD128_IRQTerminate

Function: Terminates an active interrupt process.

Syntax:

Visual BASIC: AD128_IRQTerminate() as Integer
C: int PASCAL AD128_Terminate();

Notes: The value returned has all curve functions and scaling applied to the result.

Error Codes:

IRQ_UNINSTALL:	No interrupt process is active.
AD128_SUCCESS:	Operation was performed without error.

AD128_IRQStatus

Function: Returns the status of a pending interrupt process.

Syntax:

Visual BASIC: AD128_IRQStatus(scan as Integer, conv as Integer) as Integer
C: int PASCAL AD128_IRQStatus(int *scan,int *conv);

*scan: Pointer for the driver to return the number of scans completed so far.
*conv: Pointer for the driver to return the number of conversions completed during the current scan.

Notes: When the values returned in scan and conv are greater than or equal to the values you passed to AD128_IRQScan, then the process is complete.

Error Codes:

INVALID_PTR:	One or more of the return pointers is invalid.
AD128_SUCCESS:	Operation was performed without error.

AD128_IRQScan

Function: Scans a portion, or the entire point list a given number of times. This is a batch process that will read a conversion on each interrupt that is generated. The interrupts are generated by the EOC signal.

Syntax:

Visual BASIC: AD128_IRQScan(Byval scans as Integer, Byval convs as Integer, Byval index as Integer, Byval IRQ as integer, Byval process as integer, Byval hWnd as integer) as Integer

C: int PASCAL AD128_IRQScan(int scans,int convs,int index,unsigned IRQ, unsigned process, HWND hWnd);

scans:	The number of conversion scans of the point list to perform.
convs:	The number of conversions to perform per scan.
index:	The starting index of the point list.
IRQ:	The IRQ level to use.
process:	The source of start conversion, if 0 then software start, 1 is timer and 2 is external.
hWnd:	A handle to the calling window.

Notes: The total number of conversions taken is equal to the scans parameter multiplied by the convs parameter.

After setting up the process, AD128_IRQScan will exit, before conversions are complete. The process will run as a background task.

Desired portions of the point list may be converted by setting the index parameter. The index parameter is a numerical value that indicates which position in the point list to start each scan. Each succeeding scan will start at this point in the point list. To start at the beginning of the point list use a zero for this parameter.

For the source of start conversions, you may pass one of the provided constants that are located in the include files, A12DRV.H for C and A12DRV.GBL for Visual BASIC. The constants are SOFTWARE, TIMER (TTIMER for Visual BASIC) and EXTERNAL.

If the counter/timer is to be used to start conversions, then before calling the AD128_IRQScan routine, you should set up the timers by using AD128_RateGenerator to set the desired conversion frequency.

If a window handle is passed in the hWnd parameter, than the driver will post a message to that window upon completion of all conversions. If you do not wish to use this feature, pass a 0 in Visual BASIC or a NULL in C. Alternatively, you may poll from time-to-time to check if all conversions are complete by calling AD128_IRQStatus.

The data taken are stored in the driver with no curve functions or scaling performed. After calling this routine, a call to AD128_PostProcess() will return the data with all functions and scaling performed. The call to AD128_PostProcess() should have identical values for scans, convs and index as the call to AD128_ IRQScan().

Error Codes:

LIST_EMPTY:	The point list is empty.
INVALID_CONV:	The scans and/or convs parameter < 1.
INVALID_IRQ:	The IRQ parameter is not <= 2 and <= 7.
PROCESS_ERROR:	Start conversion source is not 0, 1, 2.
ACTIVE_PROCESS:	A batch process is currently active in the driver.
WINDOWSERROR:	A Windows related error occurred.
LIST_ERROR:	An error occurred traversing the point list.
AD128_SUCCESS:	Operation was performed without error.

AD128_PollScan

Function: Scans a portion, or the entire point list a given number of times. This is a batch process that will poll the card until each conversion is complete.

Syntax:

Visual BASIC: AD128_PollScan(Byval scans as Integer, Byval convs as Integer, Byval index as Integer) as Integer

C: int PASCAL AD128_PollScan(int scans,int convs,int index);

scans: The number of scans of the point list to perform.
convs: The number of points to convert.
index: The first point in the point list to be scanned

Notes: The total number of conversions taken is equal to the scans parameter multiplied by the convs parameter.

Desired portions of the point list may be converted by setting the index parameter. The index parameter is a numerical value that indicates which position in the point list to start each scan. Each succeeding scan will start at this point in the point list. To start at the beginning of the point list use a zero for this parameter.

This routine will not exit until the entire process is complete.

The data taken are stored in the driver with no curve functions or scaling performed. After calling this routine, a call to AD128_PostProcess() will return the data with all functions and scaling performed. The call to AD128_PostProcess() should have identical values for scans, convs and index as the call to AD128_ PollScan().

Error Codes:

LIST_EMPTY:	The point list is empty.
INVALID_CONV:	The scans and/or convs parameter < 1.
ACTIVE_PROCESS:	A batch process is currently active in the driver.
WINDOWSERROR:	A Windows related error occurred.
LIST_ERROR:	An error occurred traversing the point list.
POINT_UNINSTALL:	The next point in the point list has not been installed with a call to AD128_AddPoints().
CARD_INACTIVE:	The card does not respond, no end-of-conversion.
AD128_SUCCESS:	Operation was performed without error.

AD128_PostProcess

Function: Takes conversions stored in the driver's internal buffer, performs curve functions and scaling on the data, and transfers the results to another buffer supplied by the caller.

Syntax:

Visual BASIC: AD128_PostProcess(Byval scans as Integer, Byval convs as Integer, Byval index as Integer, buffer(0) as Single) as Integer

C: int PASCAL AD128_PostProcess(int scans,int convs,int index,float *buffer);

scans:	The number of conversion scans of the point list to perform.
convs:	The number of conversion to perform per scan.
index:	The starting index of the point list.
*buffer:	Point to a floating point data buffer where the driver can place the results.

Notes: The total number of transfers made is equal to the scans parameter multiplied by the convs parameter.

The function is used to transfer data taken in a batch process, such as AD128_PollScan(). It will perform all curve and scaling for each point of data taken in the batch process.

Error Codes:

BUFFER_EMPTY:	The driver's internal data buffer is empty.
LIST_EMPTY:	The point list is empty.
INVALID_CONV:	The scans and/or convs parameter is > 1.
WINDOWSERROR:	A Windows related error occurred.
LIST_ERROR:	An error occurred traversing the point list.
INVALID_BUFFER:	Pointer to a buffer to return the processed data.
POINT_UNINSTALL:	The next point in the point list has not been installed with a call to AD128_SetPointConfig().
CARD_INACTIVE:	The card does not respond, no EOC.
AD128_SUCCESS:	Operation was performed without error.

AD128_FetchLastGain

Function: Fetches the last gain code used for a given point.

Syntax:

Visual BASIC: AD128_FetchLastGain(Byval addr as Integer, gain as Integer) as Integer

C: int PASCAL AD128_FetchLastGain(int addr,int *gain);

addr: The point address of the desired point.

*gain: A pointer to a integer for the return of the gain code.

Notes: This function is mainly useful in cases where points have been assigned an auto gain code. This is the only way for a user program to determine the gain used for a point that has been set to auto gain

Error Codes:

INVALID_PTR:	The gain code pointer is invalid.
AD128_SUCCESS:	Operation was performed without error.

AD128_DigitalOut

Function: Writes a byte out to the digital output port.

Syntax:

Visual BASIC: AD128_DigitalOut(Byval value as Integer) as Integer

C: int PASCAL AD128_DigitalOut(int value);

value: The value to write to the digital output port.

Notes: Even though the value parameter is a 16 bit value, the driver will only use the lower eight bits of the number.

Each bit position in the eight bits used is one digital output bit.

The direction of the eight digital bits on the AD12-8 is programmable by jumper selection on the card. See Hardware Configuration and Installation for details.

Error Codes: AD128_SUCCESS:Operation was performed without error.

AD128_DigitalIn

Function: Reads a byte from the digital input port.

Syntax:

Visual BASIC: AD128_DigitalIn(value as Integer) as Integer

C: int PASCAL AD128_DigitalIn(int *value);

*value: Pointer to return the value read from the digital input port.

Notes: Even though the value parameter is a 16 bit value, the driver will only use the lower eight bits of the number.

Each bit position in the eight bits used is one digital output bit.

The direction of the eight digital bits on the AD12-8 is programmable by jumper selection on the Card. See Hardware Configuration and Installation for details.

Error Codes:

INVALID_PTR:	The gain code pointer is invalid.
AD128_SUCCESS:	Operation was performed without error.

AD128_SetCounter

Function: Set up the indicated counter.

Syntax:

Visual BASIC: AD128_SetCounter(Byval counter as Integer, Byval mode as Integer, Byval loadvalue as integer, Byval bcd as integer) as Integer

C: int PASCAL AD128_SetCounter(int counter,int mode, unsigned loadvalue, int bcd);

counter:	Counter number to set up.
mode:	Counter mode, possible values are 0 to 5.
loadvalue:	Initial counter load contents.
bcd:	Number system to count by. If bcd = 1 then counter counts in BCD if 0, counter counts in binary.

Notes: The counter will begin counting whenever its gate is brought and held high.
For a more complete discussion of the counter/timers please see Programmable Interval Timer.

Error Codes:

INVALID_COUNTER:	Counter number is not 0,1 or 2.
INVALID_MODE:	Counter mode is not 0 through 5.
AD128_SUCCESS:	Operation was performed without error.

AD128_ReadCounter

Function: Reads the contents of the specified counter.

Syntax:

Visual BASIC: AD128_ReadCounter(Byval counter as Integer, value as Integer)as Integer

C: int PASCAL AD128_ReadCounter(int counter, unsigned *value);

counter: Counter number to read.

*value: Pointer to return the counter contents.

Notes: The counter is latched before it is read.
For a more complete discussion of the counter/timers please see Programmable Interval Timer.

Error Codes:

INVALID_PTR:	The pointer was invalid.
INVALID_COUNTER:	Counter number is not 0,1 or 2.
AD128_SUCCESS:	Operation was performed without error.

AD128_RateGenerator

Function: Configures Counters 1 and 2 to generate a desired frequency.

Syntax:

Visual BASIC: AD128_RateGenerator(Byval freq as Single) as Integer

C: int PASCAL AD128_RateGenerator(float freq);

freq: The desired frequency.

Notes: Mode 2 is used for the counters.
The frequency range for this function is 0.001 to 100kHz.
For a more complete discussion of the counter/timers please see Programmable Interval Timer.

Error Codes:

FREQ_ERROR:	The desired frequency is out of range.
AD128_SUCCESS:	Operation was performed without error.

AD128_DisableCounter

Function: Disables the given counter to a high or low level.

Syntax:

Visual BASIC: AD128_DisableCounter(Byval counter as Integer, Byval state as Integer) as Integer

C: int PASCAL AD128_DisableCounter(int counter,int state);

counter: Number of the counter to disable.

state: If state = 0 then the output is disabled to a low state. Any other value disables the counter output to a high state.

Notes: For a more complete discussion of the counter/timers please see Programmable Interval Timer.

Error Codes:

INVALID_COUNTER:	Counter number is not 0, 1 or 2.
AD128_SUCCESS:	Operation was performed without error.

AD128_MeasureFreq

Function: Measures an unknown frequency.

Syntax:

Visual BASIC: AD128_MeasureFreq(Byval range as Integer, freq as Single) as Integer

C: int PASCAL AD128_MeasureFreq(int range,float *freq);

range: Range of the unknown frequency.

*freq: Pointer to return the measured frequency.

Notes: The following external connections are required:

1. The counter 1 output, pin 23 should be connected to digital input bit 0, pin 25, and to the gate of counter 0, pin 21.
2. The unknown frequency should be connected between the clock of counter 0, pin 2, and digital ground, pin 11.

   Use range = 0, if the unknown frequency is less than 500KHz. Use range = 1 for frequencies greater than 500KHz, and up to 8 MHz.

   The error of this function at 1Khz is 3%, and increases rapidly for frequencies less than 1KHz. As frequencies increase above 1KHz, the error decreases, for example at 10KHz, the error is only 0.3%.

   The frequency range for this function is 0.001 to 100kHz.

Error Codes:

AD128_SUCCESS:	Operation was performed without error.
INVALID_PTR:	The point to return the frequency was not valid.

AD128_MeasurePeriod

Function: Measures an unknown period.

Syntax:

Visual BASIC: AD128_MeasurePeriod(Byval range as Integer, period as Single) as Integer

C: int PASCAL AD128_MeasurePeriod(int range,float *period);

range: Range of the unknown frequency.

*period: Pointer to return the measured frequency.

Notes: See AD128_MeasureFreq for setup and usage instruction, as this function is an inverse of AD128_MeasureFreq().

Error Codes:

AD128_SUCCESS:	Operation was performed without error.
INVALID_PTR:	The point to return the period was not valid.

Summary of Error Codes

0:	AD128_SUCCESS:	The function was completed without error.
1:	RESERVED	
2:	CONFIG_CODE:	A configuration code passed to the initialization routine was not a 0 for manual configuration or a 1 for auto configuration.
3:	BASE_ADDRESS:	The base address provided to the driver was not between 200 hex and 3F8 hex.
4:	VOLTAGE_RANGE:	The voltage range code was not 0 for bipolar 5V, 1 for bipolar 10V or 2 for unipolar 10V.
5:	CARD_INACTIVE:	The AD12-8 does not respond to commands, probably because the base address passed to the driver is not correct, or it is set incorrectly on the card.
6:	RESERVED	
7:	POINT_ERROR:	The point address passed is < 0 or > 127.
8:	POINT_INSTALL:	A point address passed to the driver for conversion or to be placed into the point list has not been installed.
9:	LIST_EMPTY:	A function using the point list was called before any points were added to the list.
10:	LIST_ERROR:	A index into the point list has been passed to the driver that does not exist.
11:	INVALID_CONV:	The number of conversions or scans passed to the driver are < 1.
12:	INVALID_BUFFER:	The buffer passed was not large enough to hold all data in the driver's internal buffer.
13:	POINT_UNINSTALL:	A point was not set up with AD128_SetPointConfig().
14:	ACTIVE_PROCESS:	A request was made to start a batch process when one was already active.
15:	BUFFER_EMPTY:	A post processing request was made when the driver's internal data buffer was empty.
16:	INVALID_PTR:	A pointer passed to the driver for returning data was not a valid pointer.
17:	GAIN_ERROR:	A gain code passed to the driver was not between zero and eight.
18:	UNIT_ERROR:	A temperature unit code passed to the driver was not 0 for °F. or 1 for °C.
19:	CURVE_ERROR:	A curve passed to the driver was not T,b,e,j,k,r,s,t,a or u.
20:	CHANNEL_ERROR:	Reference channel was < 0 or > 8.
21:	RESERVED:	Reserved for future use.
22:	FILE_ERROR:	An error occurred opening or reading the configuration file.
23:	CONFIG_ERROR:	A configuration error occurred attempting to configure the driver with the configuration file. There is probably an error in the contents of the file.
24:	INVALID_COUNTER:	The counter number passed to the driver was not between zero and two.

25:	FREQ_ERROR:	A requested frequency for the rate generator function was out of range.
26:	INVALID_MODE:	A counter mode passed to the driver was not 0,1,2,3,4,5.
27:	TIMEOUT:	Card did not respond to a start conversion.
99:	WINDOWSERROR:	An error occurred generated by Windows, this is probably a memory error that the driver is not capable of dealing with. Try running Windows in standard mode or adding more memory. The drivers memory requirements increase based on the number of points in the point list and the number of AD12-8 cards you are configuring with the driver.

Chapter 7: A/D Converter Applications

Connecting Analog Inputs

The AD12-8 provides eight channels of single-ended input. Single-ended configuration means that you have only one input relative to ground. A differential input provides two inputs and the signal corresponds to the voltage difference between these two inputs. The single-ended configuration is suitable only for "floating" sources; i.e., a signal source that does not have any connection to ground at the source. To use differential connections to the AD12-8, the AIM-16 multiplexer card must be added. The AIM-16 supplies 16 channels of differential input.

Thus, if the signal source has one side connected to a local ground, the AD12-8/AIM-16 combinations should be used. A differential input responds only to difference signals between the high and low inputs. In practice, the signal source ground will not be at exactly the same voltage as the computer ground where the AD12-8/AIM-16 combination is because the two grounds are connected through ground returns of the equipment and the building wiring. The difference between the ground voltages forms a common mode voltage (i.e., a voltage common to both inputs) that a differential input rejects up to a certain limit. In the case of the AD12-8/AIM-16 combination, the common mode limit is ±10V.

It's important to understand the difference between input types, how to use them effectively, and how to avoid ground loops. Misuse of inputs is the most common difficulty that users experience in applying and obtaining the best performance from data acquisition systems.

Comments on Noise Interference

Noise is generally introduced into analog measurements from two sources: (a) ground loops and (b) external noise. In both cases, use of good wiring practice will reduce and sometimes eliminate the noise. A key point with regard to ground or return wiring is that in an analog/digital "system", digital circuits should have a separate ground system from analog circuits with only a single common point. The reason for separate ground busses is that digital circuits, by their very nature, generate considerable high frequency noise as they rapidly change state.

Ground loops occur when AC noise and DC offset are added in series with a grounded signal source if the source ground is at a different potential than the A/D's analog ground. If there is an ohmic resistance between the source ground and the A/D's ground, the resultant current flow causes a voltage to be developed and a "ground loop" exists. If the signal is measured in a single-ended mode, that voltage is added to the source signal thereby creating an error. The best way to avoid ground loop errors is to use good wiring practice as described above. If this is not possible, use of a differential measurement mode will minimize errors.

Input Range and Resolution Specifications

Resolution of an A/D converter is usually specified in number of bits; i.e. 8 bits, 12 bits, etc. Input range is specified in volts; i.e. 0-5V, ±10V, ±20mV, etc. To determine the voltage resolution of an A/D converter, simply divide the full scale voltage range by the number of parts of resolution. For example, for a unipolar range of 0-10V, a 12-bit A/D resolves the input into 4096 parts. Thus, voltage resolution (the "weight" of one bit) is 2.44mV.

If an amplifier is incorporated in the circuit providing gain, then divide the voltage resolution by the gain of the amplifier, then divide by 4096. For example, a 12-bit A/D with ±5V full-scale input range and an amplifier gain of 100 will provide an overall input resolution of about 24.4μV.

Current Measurements

Current signals can be converted to voltage for measurement by the A/D converter by addition of a shunt resistor installed across the input terminals. For example, to accommodate 4-20mA current transmitter inputs, connect a 250Ω shunt resistor across the A/D input terminals. The resultant 1-5V signal can then be measured. The ACCES STA-37 screw terminal accessory board, for example, includes a breadboard area with plated through holes that allow insertion of shunt resistors. If an AIM-16 multiplexer expansion card is being used, pre-wired pads are provided on the AIM-16. If all the inputs are 4-20mA range current inputs from current transmitters, then there is a configuration of the multiplexer expansion board called AIM-16I. That model includes the shunt resistors and has offset and gain set such that the "live zero" is compensated for and the full 12-bit resolution of the A/D is realized.

Note: Accuracy of measurement will be directly affected by the accuracy of these resistors. Accordingly, precision resistors should be used. Also, if the ambient temperature will vary significantly, these precision resistors should be low temperature coefficient wire-wound resistors.

Measuring Large Voltages

Voltages larger than the input range of the A/D can be measured by using a voltage divider. As above, it is necessary to use precision resistors. Also if the raw voltage is a direct analog of a parameter being measured, then it will be necessary to apply a scale factor in software in order to arrive at the correct engineering units.

Adding More Analog Inputs

You can add sub-multiplexers to any or all of the analog inputs of AD12-8. ACCES' AIM-16 provides capability for 16 channels per input plus a common instrumentation amplifier. Up to eight AIM-16's can be added to one AD12-8 providing a total input capability up to 128 channels.

Precautions - Noise, Ground Loops, and Overloads

Unavoidably, data acquisition applications involve connecting external things to the computer. DO NOT get inputs mixed up with the AC line. An inadvertent short can instantly cause extensive damage. ACCES cannot accept liability for this kind of accident. As an aid to avoid this problem:

a. Avoid direct connection to the AC line.

b. Make sure that all connections are secure so that signal wires are not likely to come loose and short to high voltages.

c. Use isolation amplifiers and transformers where necessary. There are two types of ground connections on the rear connector of AD12-8. These are called Power Ground and Low Level Ground. Power ground is the noisy or dirty ground that is meant to carry all digital signals and heavy (power supply) currents. Low Level Ground is the signal ground for all analog input functions. It is only meant to carry signal currents (less than a few milliamperes) and is the ground reference for the A/D converter. Due to connector contact resistance and cable resistance there may be many millivolts difference between the two grounds even though they are connected together and to the computer and power line grounds on the AD12-8 card.

Chapter 8: Programmable Interval Timer

The AD12-8 contains a type 8254 programmable counter/timer which allows you to implement such functions as a Real-Time Clock, Event Counter, Digital One-Shot, Programmable Rate Generator, Square-Wave Generator, Binary Rate Multiplier, Complex Wave Generator, and/or a Motor Controller. The 8254 is a flexible but powerful device that consists of three independent, 16-bit, presettable, down counters. Each counter can be programmed to any count as low as 1 or 2, and up to 65,535 in binary format, depending on the mode chosen.

On the AD12-8 these three counters are designated Counter #0, Counter #1, and Counter #2. Counter #0 is un-dedicated, with the gate, output and clock connections fully accessible via the I/O connector. Counter #1 receives clock inputs from a 1/32 multiple of the high speed computer color oscillator clock (14.31818 MHz). The output of counter #1 is used as the clock for counter #2, thus counters #1 and #2 are cascaded together to form a 32-bit counter. The output of counter #2 is also available at the I/O connector, pin 23. If the CLK0 jumper is installed, the output of counter #1 can be used as an alternate input clock for counter #0. Outputs of counter #2 are used to initiate A/D conversions if the TMR/EXT jumper is in the TMR position.

Operational Modes

The 8254 modes of operation are described in the following paragraphs to familiarize you with the versatility and power of this device. For those interested in more detailed information, a full description of the 8254 programmable interval timer can be found in the Intel (or equivalent manufacturers) data sheets. The following conventions apply for use in describing operation of the 8254:

Clock:	A positive pulse into the counter's clock input.
Trigger:	A rising edge input to the counter's gate input.
Counter Loading:	Programming of a binary count into the counter.

Mode 0: Pulse on Terminal Count

After the counter is loaded, the output is set low and will remain low until the counter decrements to zero. The output then goes high and remains high until a new count is loaded into the counter. A trigger enables the counter to start decrementing. This mode is commonly used for event counting with Counter #0.

Mode 1: Retriggerable One-shot

The output goes low on the clock pulse following a trigger to begin the one-shot pulse and goes high when the counter reaches zero. Additional triggers result in reloading the count and starting the cycle over. If a trigger occurs before the counter decrements to zero, a new count is loaded. Thus, this forms a re-triggerable one-shot. In mode 1, a low output pulse is provided with a period equal to the counter count-down time.

Mode 2: Rate Generator

This mode provides a divide-by-N capability where N is the count loaded into the counter. When triggered, the counter output goes low for one clock period after N counts, reloads the initial count, and the cycle starts over. This mode is periodic, the same sequence is repeated indefinitely until the gate input is brought low. This mode is used on the AD12-8 card in counters 1 and 2 to generate periodic A/D start commands. This mode also works well as an alternative to mode 0 for event counting.

Mode 3: Square Wave Generator

This mode operates periodically like mode 2. The output is high for half of the count and low for the other half. If the count is even, then the output is a symmetrical square wave. If the count is odd, then the output is high for (N+1)/2 counts and low for (N-1)/2 counts. Periodic triggering or frequency synthesis are two possible applications for this mode. Note that in this mode, to achieve the square wave, the counter decrements by two for the total loaded count, then reloads and decrements by two for the second part of the wave form.

Mode 4: Software Triggered Strobe

This mode sets the output high and, when the count is loaded, the counter begins to count down. When the counter reaches zero, the output will go low for one input period. The counter must be reloaded to repeat the cycle. A low gate input will inhibit the counter. This mode can be used to provide a delayed software trigger for initiating A/D conversions.

Mode 5: Hardware Triggered Strobe

In this mode, the counter will start counting after the rising edge of the trigger input and will go low for one clock period when the terminal count is reached. The counter is retriggerable. The output will not go low until the full count after the rising edge of the trigger.

Programming

On the AD12-8, the 8254 counters occupy the following addresses:

BASE ADDRESS + 4: Read/Write Counter #0
BASE ADDRESS + 5: Read/Write Counter #1
BASE ADDRESS + 6: Read/Write Counter #2
BASE ADDRESS + 7: Write to Counter Control register

The counters are programmed by writing a control byte into a counter control register at BASE ADDRESS + 7. The control byte specifies the counter to be programmed, the counter mode, the type of read/write operation, and the modulus. The control byte format is as follows:

B7	B6	B5	B4	B3	B2	B1	B0
SC1	SC0	RW1	RW0	M2	M1	M0	BCD

SC0-SC1: These bits select the counter that the control byte is destined for.

SC1	SC0	Function
0	0	Program Counter 0
0	1	Program Counter 1
1	0	Program Counter 2
1	1	Read Back Command

RW0-RW1: These bits select the read/write mode of the selected counter.

RW1	RW0	Counter Read/Write Function
0	0	Counter Latch Command
0	1	Read/Write LS Byte
1	0	Read/Write MS Byte
1	1	Read/Write LS Byte, then MS Byte

M0-M2: These bits set the operational mode of the selected counter.

Mode	M2	M1	M0
0	0	0	0
1	0	0	1
2	X	1	0
3	X	1	1
4	1	0	0
5	1	0	1

BCD: Set the selected counter to count in binary (BCD bit = 0) or BCD (BCD bit = 1).

Reading and Loading the Counters

If you attempt to read an active counter, you will most likely get erroneous data. This is partly caused by carries rippling through the counter during the read operation. Also, the low and high bytes are read sequentially rather than simultaneously and, thus, it is possible that carries will be propagated from the low to the high byte during the read cycle. To circumvent these problems, you should perform a counter-latch operation in advance of the read cycle. To do this, load the RW1 and RW2 bits with zeroes. This instantly latches the count of the selected counter(selected via the SC1 and SC0 bits) in a 16-bit hold register. A subsequent read operation on the selected counter returns the held value. Latching is the best way to read an active counter without disturbing the counting process. You can only rely on directly-read counter data if the counting process is suspended while reading, by bringing the gate low, or by halting the input pulses.

For each counter you must specify in advance the type of read or write operation that you intend to perform. You have a choice of loading/reading (a) the high byte of the count, or (b) the low byte of the count, or (c) the low byte followed by the high byte.

Programming Examples

Using Counter #0 as a Pulse Counter

Note that the counters are "down" counters so, when resetting them, it's better to load them with a full count value of 65,535 rather than zero.

```
outportb(BASEADDRESS + 7,0x30);        /* counter 0, mode 0 */
outportb(BASEADDRESS + 4,0xff);        /* counter 0 low load byte */
outportb(BASEADDRESS + 4,0xff);        /* counter 0 high load byte */
```

Reading Counter #0

```
outportb(BASEADDRESS + 7,0x30);        /* counter 0, latch command */

/* read in both bytes of the latched value and combine into an integer */
value = inportb(BASEADDRESS + 4) +  (inportb(BASEADDRESS + 4) * 256;
```

Programming Examples Using the A12DRV Driver

In practice, TASKS 14 and 15 of the A12DRV driver can be used to perform equivalent operations to the above examples with fewer programming steps.

For counting pulses, the counter configuration is not of great importance because you will only be using the countdown capabilities of the counter. Mode 2 is as good as any other choice for pulse counting. As in the previous example, load Counter #0 with a full scale count of 65,535 (hex FFFF) using TASK 14 of the driver. While loading the counter, counting can be inhibited by holding the gate input, pin 21, low.

```
task = 14;                                  /* counter setup mode task */
params[0] = 0;                              /* setup counter #0 */
params[1] = 2;                              /* set counter #0 mode to 2 */
params[2] = 0xffff;                         /* set counter #0 count to ffff hex (65535) */
a12drv(FP_OFF(task),FP_OFF(params),FP_OFF(status));    /* call the driver */
```

Next, apply the number of pulses to be counted. The gate input, pin 21, must now be high or can be taken high for some fixed time interval to control the number of pulses counted. You can read the new count using TASK 15 of the driver:

```
task = 15;                                  /* read counter count task */
params[0] = 0;                              /* read counter #0 */
a12drv(FP_OFF(task),FP_OFF(params),FP_OFF(status));    /* call the driver */
```

Upon return, params[1] contains the counter contents.

Triggering A/D Conversions Periodically

When you are using the A/D converter on AD12-8, one of the key uses for the programmable interval timer is to provide start pulses for periodic sampling. The TMR/EXT jumper should be in the TMR position. You can set up an output frequency by using Task 14 to load Counters #1 and #2 with the required divisors.

For example, assume that a trigger rate of 8.3 KHz is needed. First, work out the overall division ratio from the color oscillator clock, which is divided by 32 before being applied to counter #1's clock input.

```
14,318,180 / 32 = 447,443          counter #1's clock frequency
447,443 / 8300 = 53.9              desired counts
```

The closest frequency obtainable would be :

```
447,443 / 54 = 8.286KHz            closest frequency
```

Now, divide the 54 between the two counters (6 * 9 = 54):

```
task = 14;                                  /* counter setup mode task */
params[0] = 1;                              /* setup counter #1 */
params[1] = 2;                              /* set counter #1 mode to 2 */
params[2] = 6;                              /* set counter #1 count to 6 */
a12drv(FP_OFF(task),FP_OFF(params),FP_OFF(status));    /* call the driver */

task = 14;                                  /* counter setup mode task */
params[0] = 0;                              /* setup counter #2 */
params[1] = 2;                              /* set counter #2 mode to 2 */
params[2] = 9;                              /* set counter #2 count to 9 */
a12drv(FP_OFF(task),FP_OFF(params),FP_OFF(status));    /* call the driver */
```

Generating Square Waves of Programmed Frequency

Frequency of output is a direct function of the frequency of the clock input and of the count loaded into the counter. The minimum count (or divisor) is 2 and the maximum is 65535.

Calculating what divisor to use for a specific output frequency is straightforward. If, for example, you desire a 1KHz output and your clock is 5MHz, divide by 1000 and find that the count to be loaded into counter #0 should by 5000.

Measuring Frequency and Period

The two previous sections show how to count pulses and generate output frequencies. It is also possible to measure frequency by raising the gate input of Counter #0 for a known time interval and counting the number of clock pulses accumulated for that interval. The gating signal can be derived from Counters #1 and #2 operating in a square wave mode.

Counter #0 can also be used to measure pulse width or half period of a periodic signal. The signal should be applied to the gate input of Counter #0 and a known frequency applied to the Counter #0 clock input. During the interval when the gate input is low, Counter #0 is loaded with a full count of 65,535. When the gate input goes high, the counter begins decrementing until the gate input goes back low at the end of the pulse. The counter is then read and the change in counts is a linear function of the duration of the gate input signal. If Counter #0 receives 10 microsecond duration clock pulses (100 KHz), the maximum pulse duration that can be measured is 65,535*10-5 = 655 milliseconds.

Generating Time Delays

There are four methods of using Counter #0 to generate programmable time delays.

Pulse on Terminal Count

After loading, the counter output goes low. Counting is enabled when the gate goes high. The counter output will remain low until the count reaches zero, at which time the counter output goes high. The output will remain high until the counter is reloaded by a programmed command. If the gate goes low during countdown, counting will be disabled as long as the gate input is low.

Programmable One-Shot

The counter need only be loaded once. The time delay is initiated when the gate input goes high. At this point the counter output goes low. If the gate input goes low, counting continues but a new cycle will be initiated if the gate input goes high again before the timeout delay has expired; i.e., is re-triggerable. At the end of the timeout, the counter reaches zero and the counter output goes high. That output will remain high until re-triggered by the gate input.

Software Triggered Strobe

This is similar to Pulse-on-Terminal-Count except that, after loading, the output goes high and only goes low for one clock period upon timeout. Thus, a negative strobe pulse is generated a programmed duration after the counter is loaded.

Hardware Triggered Strobe

This is similar to Programmable-One-Shot except that when the counter is triggered by the gate going high, the counter output immediately goes high, then goes low for one clock period at timeout, producing a negative-going strobe pulse. The timeout is re-triggerable; i.e., a new cycle will commence if the gate goes high before a current cycle has timed out.

Appendix A: Linearization

A common requirement encountered in data acquisition is to linearize the output of non-linear transducers such as thermocouples, RTDs, etc. The starting point for any linearizing algorithm is knowledge of the calibration curve (input/output behavior) of the sensor. This may be derived experimentally, or may be available in manufacturer's data, or in standard tables.

There are several approaches to linearization. The two most common are (a) piecewise linearization using look-up tables and (b) use of a mathematical function to approximate the non-linearity. Amongst the mathematical methods, polynomial expansion is one of the easiest to implement. The utility program, POLY.EXE, allows you to generate up to a 10th order polynomial approximation. For most practical applications, a fifth-order polynomial approximation is usually adequate.

Before you start the program have the input/output data or calibration data handy. This will be in the form of x and f(x) values where x is the input to your system and f(x) is the resulting output. To run the program, type POLY and K at the command line. The program will then prompt you for the desired order of the polynomial, then the number of pairs that you wish to use to generate the polynomial. You then enter the data pairs and the polynomial is computed and displayed.

For example, given the following data points, let's generate a 5th order polynomial to approximate this function:

x	0	1	2	3	4	5	6	7	8	9	10
f(X)	3	2	3	5	3	4	3	2	2	3	2

The order of the polynomial that you desire will be 5 and the number of data points that you enter will be 11. After the data points are entered, the program gives the following output:

For the polynomial f(x) = C(0) + C(1)x1 + C(2)x2 + C(3)x3 + C(4)x4 + C(5)x5, the coefficients will be:

COEFFICIENT (5) : -0.003151
COEFFICIENT (4) : 0.081942
COEFFICIENT (3) : -0.740668
COEFFICIENT (2) : 2.635998
COEFFICIENT (1) : -2.816607
COEFFICIENT (0) : 2.956044

QUALITY OF SOLUTION (sum of the errors squared): 2.797989

The goal is to make the quality as close to 0 as possible. The program checks the resulting polynomial with the data pairs that you entered. It computes the f(x) values for each x value entered using the polynomial, subtracts the result from the supplied value of f(x), and then squares the result. The squared results are then summed to compute the QUALITY. If the computed f(x) values were exact, this value would be 0. But, since this is an approximation, this value will usually be something greater than 0.

The QUALITY can be used to indicate how good a particular solution is. If the range of points is very wide or if the points make transition from negative to positive values, then QUALITY will suffer. For these cases, it may be better to use multiple polynomials rather than just one. For example, the following data are taken from the NIST tables for type T thermocouples:

x	-6.258	-5.603	-4.468	-3.378	-1.182	0	2.035
f(x)	-270	-200	-150	-100	-50	0	50

4.277	6.702	9.286	12.01	14.86	17.82	20.87
100	150	200	250	300	350	400

If we take all the data and compute one 5th order polynomial, the QUALITY is 473.543732; not very good. Now divide the data into two polynomials; one on the negative side including 0 and one on the positive side also using 0. The results will show a QUALITY of 90.732620 for the negative side and a QUALITY of 0.005131 for the positive side. Thus, by using two polynomials, you have made the positive side very accurate and dramatically improved the negative side.

Accuracy of the negative side can be further improved by adding points. For example, add the following pairs to the negative side of the polynomial for a type T thermocouple:

x	-6.181	-5.167	-4.051	-2.633
f(x)	-250	-175	-125	-75

If you run the new data, the QUALITY is improved to 69.555611, but still perhaps not as good as you would like.

Thus, you may use the QUALITY as a means to determine how good the polynomial is. You can experiment with both order and number of data points until you are satisfied with the solution. Incidentally, this example also shows that the smaller the range of x values, the better the solution.

The computational method used is a least squares solution using Gauss Elimination with partial pivoting to improve accuracy.

Appendix B: Cabling and Connector Information

AD12-8 Output Connector Pin Assignments

Connections are made to the AD12-8 card via a 37-pin D type connector that extends through the back of the computer case. The female mating connector can be a Cannon #DC-37S for soldered connections or insulation displacement flat cable types such as AMP #745242-1 may be used. The wiring may be directly from the signal sources or may be on ribbon cable from screw terminal accessories such as ACCES I/O Products Inc. part numbers STA-37 or TAD12-8.

Pin	Name	Function
1	+12VDC	+12VDC Power from the Computer Bus
2	CTR0 CLK	Counter 0 Clock
3	GN0	LSB Digital Output, gain control for AIM-16
4	USER	Digital Output, User
5	GN1	Bit 1 Digital Output, gain control for AIM-16
6	GN2	MSB Digital Output, gain control for AIM-16
7	OP0	LSB Digital Output, sub-multiplexer channel select
8	OP1	Bit 1 Digital Output, sub-multiplexer channel select
9	OP2	Bit 2 Digital Output, sub-multiplexer channel select
10	OP3	MSB Digital Output, sub-multiplexer channel select
11	PWR GND	Power (Digital) ground
12	DIO7	Digital Output, Bit 7, jumper selected as input or output
13	DIO6	Digital Output, Bit 6, jumper selected as input or output
14	DIO5	Digital Output, Bit 5, jumper selected as input or output
15	DIO4	Digital Output, Bit 4, jumper selected as input or output
16	DIO3	Digital Output, Bit 3, jumper selected as input or output
17	CTR0 OUT	Counter 0 Output
18	L.L. GND	Low Level (Analog) Ground
19	VREF	+10.0VDC (220mA) A/D reference output
20	-12VDC	-12VDC Power from the Computer Bus
21	CTR0 GATE	Counter 0 Gate
22	CTR1 GATE	Counter 1 Gate
23	CTR1 OUT	Counter 1 Output
24	INT	Interrupt input, positive edge trigger
25	DIO0	Digital Output, Bit 0, jumper selected as input or output
26	DIO1	Digital Output, Bit 1, jumper selected as input or output

27	DIO2	Digital Output, Bit 2, jumper selected as input or output
28	REF GND	A/D Reference Return
29	+5VDC	+5VDC Power from the Computer Bus
30	CH7 IN	Chl 7 Analog Input
31	CH6 IN	Chl 6 Analog Input
32	CH5 IN	Chl 5 Analog Input
33	CH4 IN	Chl 4 Analog Input
34	CH3 IN	Chl 3 Analog Input
35	CH2 IN	Chl 2 Analog Input
36	CH1 IN	Chl 1 Analog Input
37	CH0 IN	Chl 0 Analog Input

Table B-1: Output Connector Pin Assignments

Appendix C: Basic Integer Variable Storage

Data are stored in integer variables (% type) in binary form. Each integer variable uses 16 bits or two bytes of memory. Sixteen bits of binary data is equivalent to 0 to 65,535 decimal. Numbers are represented as follows:

Number	High Byte								Low Byte							
	B 7	B 6	B 5	B 4	B 3	B 2	B 1	B 0	B 7	B 6	B 5	B 4	B 3	B 2	B 1	B 0
+32767	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
+10000	0	0	1	0	0	1	1	1	0	0	0	1	0	0	0	0
+1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
-1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
-10000	1	1	0	1	1	0	0	0	1	1	1	1	0	0	0	0
-32768	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Integer variables are the most compact form of storage for the 12-bit data from the A/D converter and the 16-bit data from the interval timer. Therefore, to conserve memory and disk space and to optimize execution time, all data exchange via the CALL is through integer type variables.

This poses a programming problem when handling unsigned numbers in the range 32,768 to 65,535. If you wish to input or output an unsigned integer greater than 32,767, then it is necessary to work out what its 2's complement signed equivalent is. For example, if 50,000 decimal is to be loaded into a 16-bit counter, an easy way to convert this to binary is to enter BASIC and execute PRINT HEX$(50000). This returns C350 which, in binary form is 1100 0011 0101 0000. Since the most significant bit is a one, this would be stored as a negative integer and, in fact, the correct integer variable value would be 50,000 - 65,536 = -15,536.

Thus, the programming steps to switch between integer and real variables for representation of unsigned numbers between 0 and 65,535 is:

-From real variable N (where 0 <= N <= 65,535) to integer variable N%:

```
xxx10 IF N<=32767 THEN N% = N ELSE N% = N-65536
```

-From integer variable N% to real variable N:

```
xxx20 IF N% >= 0 THEN N=N% ELSE N = N%+65536
```

Customer Comments

If you experience any problems with this manual or just want to give us some feedback, please email us at: ***manuals@accesioproducts.com.***. Please detail any errors you find and include your mailing address so that we can send you any manual updates.

ACCES I/O PRODUCTS, INC.
10623 Roselle Street, San Diego CA 92121
Tel. (619)550-9559 FAX (619)550-7322
www.accesioproducts.com