# OPERATIONS MANUAL
# PCM-ADIO

**NOTE:** *This manual has been designed and created for use as part of the WinSystems Technical Manuals CD and/or the WinSystems website. If this manual or any portion of the manual is downloaded, copied or emailed, the links to additional information (i.e. software, cable drawings) may be inoperable.*

WinSystems reserves the right to make changes in the circuitry
and specifications at any time without notice.
©Copyright 2008 by WinSystems. All Rights Reserved.

**REVISION HISTORY**

**P/N 403-0311-000**

| ECO Number | Date Code | Rev Level |
| --- | --- | --- |
| ORIGINATED | 040408 | C |
| 06-36 | 060413 | C.1 |
| 08-16 | 080325 | C.2 |

# Table of Contents

**Visual Index – Quick Reference** .......... **i**
- Top View - Connectors i

**Introduction** .......... **1**

**General Information** .......... **1**
- Features 1
- General Description 1

**Functional Capability** .......... **2**
- I/O Address Selection 2
- Interrupt Routing 2
- Digital I/O 3
- PC/104 Bus Interface 4

**Software Summary** .......... **5**
- WS16C48 Programming Reference 7
- Sample Programs 11
- Summary **11**

**C Source Code Listings** .......... **12**

**Cables** .......... **28**

**Software Drivers & Examples** .......... **28**

**Jumper Reference** .......... **29**

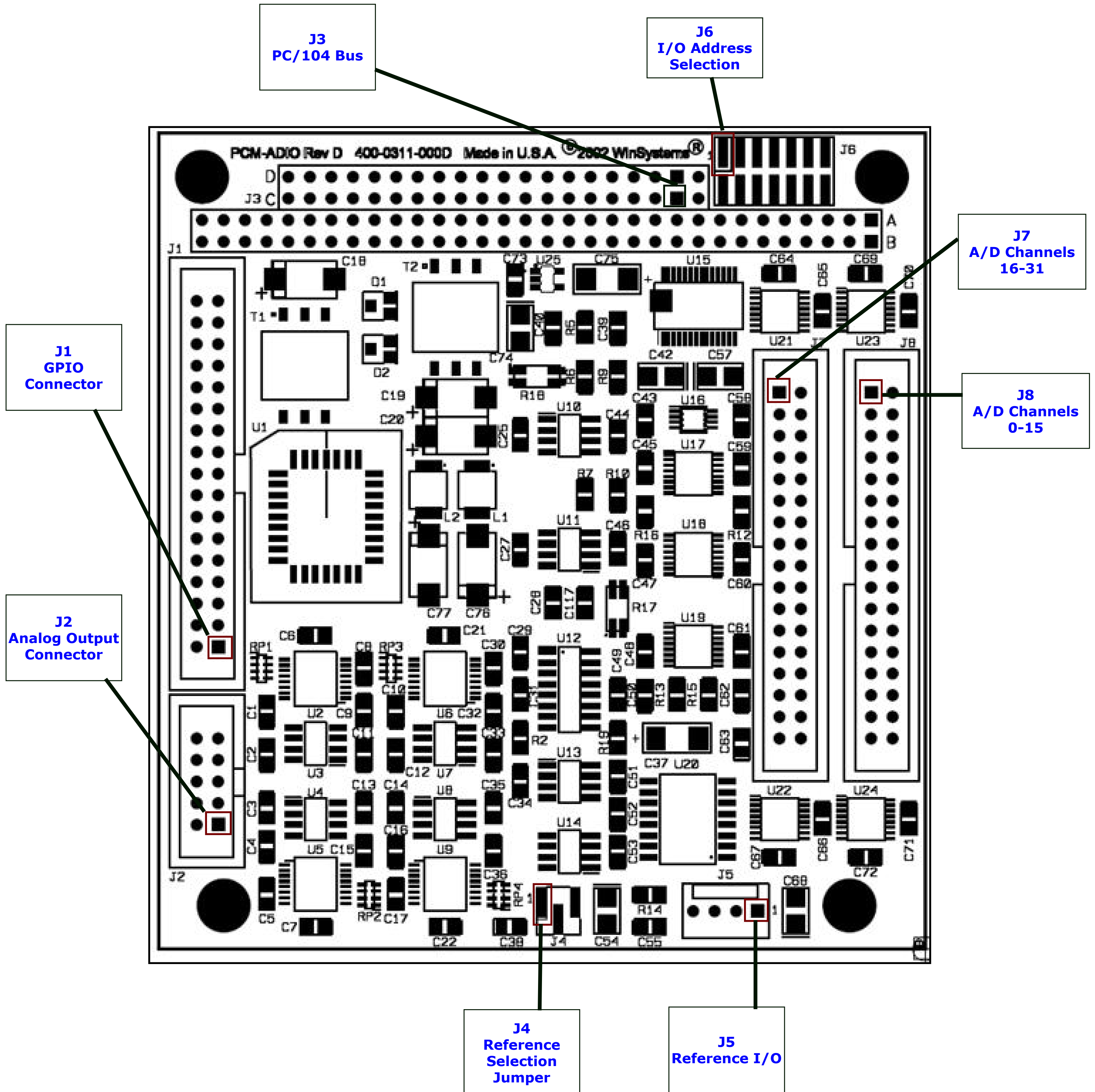
**Specifications** .......... **32**

**WARRANTY REPAIR INFORMATION** .......... **33**

# Visual Index – Quick Reference

**Top View - Connectors**

For the convenience of the user, a copy of the Visual Index has been provided with direct links to connector and jumper configuration data.




**NOTE:** The reference line to each component part has been drawn to Pin 1, where applicable. Pin 1 is also highlighted with a red square, where applicable.

# Introduction

This manual is intended to provide the necessary information regarding configuration and usage of the PCM-ADIO board. WinSystems maintains a Technical Support Group to help answer questions regarding usage or programming of the board. For answers to questions not adequately addressed in this manual, contact Technical Support at (817) 274-7553, Monday through Friday, between 8 AM and 5 PM Central Standard Time (CST).

# General Information

## Features

### Analog Input

- Four 8-channel, 16-bit Analog-to-Digital (A/D) (LTC-1609) with sample-and-hold circuit support
- Conversion Rate: Up to 31 KHz
- Each channel independently software programmable for input type and range
- Input ranges:
  0-1.25V, 0-2.5V, 0-5V, 0-10V, +/-1.25V, +/-2.5V, +/-5V or +/-10V
- Input Protection: +/-25V
- Interrupt I/O supported

### Analog Output

- Four channel, 12-bit Digital-to-Analog (D/A) (LTC-1588)
- Output ranges:
  0-5V, 0-10V, +/-2.5V, +/-5V or +/-10V
- Each channel independently software programmable for output type and range
- Output channels can be updated and cleared individually or simultaneously
- Supports industry standard signal conditioners

### Digital I/O

- 16-bit TTL-compatible general purpose digital I/O
- 12 mA Sink/Source Current
- Software programmable interrupt configuration

### Power

- +5V @ 850 mA required

### Industrial Operating Temperature Range

- -40°C to 85°C

### Form Factor

- PC/104-compliant
- 3.60” x 3.80” (90 mm x 96 mm)

## General Description

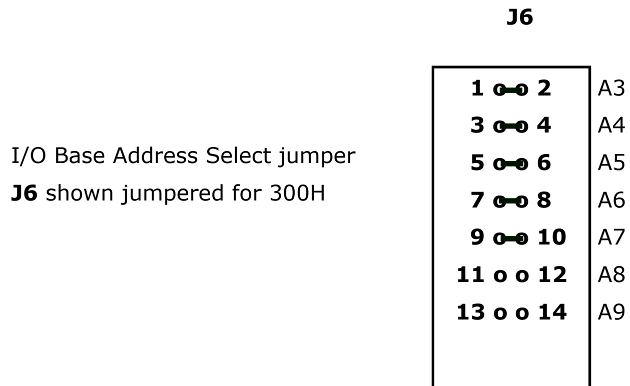
The PCM-ADIO from WinSystems is an intelligent, PC/104-based, analog input, analog output, and digital I/O board designed to meet customer demands for a high-accuracy and high channel count analog I/O board. Based on 16-bit analog-to-digital and 12-bit digital-to-analog converters, the PCM-ADIO includes such unique features as advanced automatic conversion control, automatic calibration for all analog input and output ranges.

# Functional Capability

## I/O Address Selection

The PCM-ADIO requires eight consecutive I/O addresses beginning on an 8-byte boundary. The jumper block at **J6** allows for user selection of the base address. Address selection is made by placing a jumper on the jumper pair for the address bit if a **0** is desired or leaving the jumper pair open if a **1** is required for the desired address. The illustration below shows the relationship between the address bits and the jumper position and a sample jumpering for the default address of 300H.

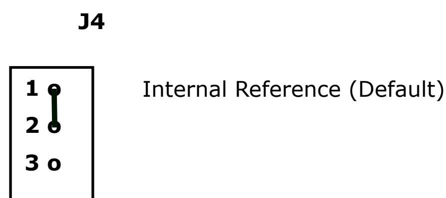
**J6**



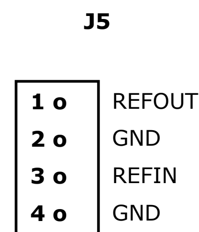

## Reference Voltage

The PCM-ADIO has an on-board precision voltage reference. Support is provided to allow piping the on-board +5V reference to external circuitry via a pin on **J5**. This pin can source up to 10 mA. Support is also provided to allow input of an external +5V reference for use in ratiometric measurements.  The board has been calibrated using the on-board reference. Contact WinSystems for application assistance if an off-board reference source is desired.

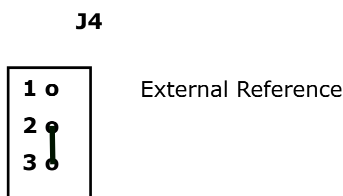
**J4**




**J5**




**J4**



## Digital I/O

### Analog Input Connections

The 32 analog input channels are terminated at **J7** and **J8**. The pin definitions for these connectors are shown here.

**J8**

| Signal | Pins | Signal |
|---|---|---|
| CH0 | **1 o o 2** | GND |
| CH1 | **3 o o 4** | GND |
| CH2 | **5 o o 6** | GND |
| CH3 | **7 o o 8** | GND |
| CH4 | **9 o o 10** | GND |
| CH5 | **11 o o 12** | GND |
| CH6 | **13 o o 14** | GND |
| CH7 | **15 o o 16** | GND |
| CH8 | **17 o o 18** | GND |
| CH9 | **19 o o 20** | GND |
| CH10 | **21 o o 22** | GND |
| CH11 | **23 o o 24** | GND |
| CH12 | **25 o o 26** | GND |
| CH13 | **27 o o 28** | GND |
| CH14 | **29 o o 30** | GND |
| CH15 | **31 o o 32** | GND |
| GND | **33 o o 34** | GND |

**J7**

| Signal | Pins | Signal |
|---|---|---|
| CH16 | **1 o o 2** | GND |
| CH17 | **3 o o 4** | GND |
| CH18 | **5 o o 6** | GND |
| CH19 | **7 o o 8** | GND |
| CH20 | **9 o o 10** | GND |
| CH21 | **11 o o 12** | GND |
| CH22 | **13 o o 14** | GND |
| CH23 | **15 o o 16** | GND |
| CH24 | **17 o o 18** | GND |
| CH25 | **19 o o 20** | GND |
| CH26 | **21 o o 22** | GND |
| CH27 | **23 o o 24** | GND |
| CH28 | **25 o o 26** | GND |
| CH29 | **27 o o 28** | GND |
| CH30 | **29 o o 30** | GND |
| CH31 | **31 o o 32** | GND |
| GND | **33 o o 34** | GND |

**NOTE**:
In differential input mode, only the even channel numbers (0,2,4,...) are used and the signal is applied between the even channel number and the next odd channel input pin.

### Analog Output Connections

The four analog output channels are terminated at **J2**. The pin definitions for **J2** are shown below.

**J2**

| Signal | Pins | Signal |
|---|---|---|
| DAC CH0 | **1 o o 2** | GND |
| DAC CH1 | **3 o o 4** | GND |
| DAC CH2 | **5 o o 6** | GND |
| DAC CH3 | **7 o o 8** | GND |
| GND | **9 o o 10** | GND |

**GPIO Connections**

The 16 bits of General Purpose I/O (GPIO) are terminated at **J1**. The pin definitions for these connectors are shown below.

**J1**

| Signal | Pins | Signal |
|---|---|---|
| Port A - Bit 0[1] | **1 o o 2** | GND |
| Port A - Bit 1[2] | **3 o o 4** | GND |
| Port A - Bit 2 | **5 o o 6** | GND |
| Port A - Bit 3 | **7 o o 8** | GND |
| Port A - Bit 4 | **9 o o 10** | GND |
| Port A - Bit 5 | **11 o o 12** | GND |
| Port A - Bit 6 | **13 o o 14** | GND |
| Port A - Bit 7 | **15 o o 16** | GND |
| Port B - Bit 0 | **17 o o 18** | GND |
| Port B - Bit 1 | **19 o o 20** | GND |
| Port B - Bit 2 | **21 o o 22** | GND |
| Port B - Bit 3 | **23 o o 24** | GND |
| Port B - Bit 4 | **25 o o 26** | GND |
| Port B - Bit 5 | **27 o o 28** | GND |
| Port B - Bit 6 | **29 o o 30** | GND |
| Port B - Bit 7 | **31 o o 32** | GND |
| GND | **33 o o 34** | GND |

[1] This bit may be used as an external sampling clock. When used for this purpose all other bits in Port A are set to input mode.

[2] This bit may be configured as an external DAC reset (All outputs to 0). When used in this mode, all other bits in port A are set to input mode.

## Register Definitions

The PCM-ADIO utilizes eight consecutive I/O addresses for passing commands, data, and parameters to and from the board. All offsets specified are from the base address selected using **J6**.

**Base Address + 0** - Command Register (Write)
This register, when written, transfers a command to the board. All commands consist of a single command byte along with 0 or more parameter bytes loaded into parameter register B – E. A write to this register sets bit 0 in the **Status** register, which will be cleared by the board's firmware once the command has been serviced. Command bytes should never be written when the **Busy** bit is set in the status register or unexpected results will follow.

**NOTE**:
All required parameter registers **must** be loaded **before** writing the command.

**Base Address + 0** - Status Register (Read)
This register provides status information from the on-board firmware. This register is bit mapped and the following bits are currently defined.

| BIT | DEFINITION |
|---|---|
| Bit 7 | **Illegal command or parameter**<br>The command and/or its parameters were not recognized or legal for the command. This bit will be cleared following the writing of a legal command by onboard firmware. |
| Bit 6 | **Reserved** |
| Bit 5 | Data Request<br>This bit, when set, indicates that a block data transfer has been requested and that the transfer has not yet been completed. It's the responsibility of the user to know whether the operation is an input or output operation. |
| Bit 4 | **Interrupt Request**<br>This bit, when set, indicates the board has pending interrupts. |
| Bit 3 | **Reserved** |
| Bit 2 | **Reserved** |
| Bit 1 | **A2D Conversion in Progress**<br>This bit when set, signals that there is an A2D conversion in progress. It can be used for polled-mode single conversion operations. |
| Bit 0 | **Busy**<br>This bit, when set signals that the onboard processor is currently parsing and processing the command request. It is set automatically in hardware and is cleared when the command is complete. No data should be written to the board while this bit is set |

**Base Address + 1** - Parameter B Register (Read/Write)
**Base Address + 2** - Parameter C Register (Read/Write)
**Base Address + 3** - Parameter D Register (Read/Write)
**Base Address + 4** - Parameter E Register (Read/Write)
**Base Address + 5** - Parameter F Register (Read/Write)
**Base Address + 6** - Parameter G Register (Read/Write)

These registers are used to pass parameter values to the board or to obtain results from the board. These registers, when read, do not necessarily reflect the value written. Refer to the Command Definition section for the usage of this register for any particular command.

**Base Address + 7** - Parameter H Register (Read/Write)

This register is the block data register used for transferring blocks (256 bytes) of data to /from the board. This register is designed to allow high speed transfers using **REP INSB** or **REP OUTSB** type 8086 assembly language instructions for maximum throughput. Reads or writes to this register are only meaningful following a block input or block output command and only when the **Data Request** bit (bit 5) in the **Status** register is set.

# Command Definitions

This section defines the command set understood by the PCM-ADIO on-board firmware. To execute a command, it is first necessary to write any of the parameter bytes to the appropriate register and then load the command byte into the command register. The commands are categorized under various types.

### Control and Diagnostic Commands

| **COMMAND 00H - Clear Interrupt**<br>This command clears the interrupt request line to the ISA bus. It must be sent at the exit of an interrupt service routine to signal the board to release the IRQ line. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - N/A | Parameter Register B - N/A |
| Parameter Register C - N/A | Parameter Register C - N/A |
| Parameter Register D - N/A | Parameter Register D - N/A |
| Parameter Register E - N/A | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

| **COMMAND 01H - Set IRQ Number**<br>This command sets the IRQ value (0-15) that the board will assert when an interrupt event occurs. Not all values are usable on all systems. This value must be set before issuing any commands which may generate an interrupt. This value is not saved and must be reloaded after each boot. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - IRQ Number 0-15 | Parameter Register B - N/A |
| Parameter Register C - N/A | Parameter Register C - N/A |
| Parameter Register D - N/A | Parameter Register D - N/A |
| Parameter Register E - N/A | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

| **COMMAND 02H - Test Interrupt**<br>This diagnostic command is used to test that the interrupt line is working. The board will assert the IRQ line set by the **Set IRQ Number** command. Software should then clear the interrupt using the **Clear Interrupt** command. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - N/A | Parameter Register B - N/A |
| Parameter Register C - N/A | Parameter Register C - N/A |
| Parameter Register D - N/A | Parameter Register D - N/A |
| Parameter Register E - N/A | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

| **COMMAND 03H - Parameter Register Test**<br>This diagnostic command simply echoes back the parameter registers. Anything written into Parameter Registers B–G will be echoed back. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - Test value | Parameter Register B - Test wrote |
| Parameter Register C - Test value | Parameter Register C - Test wrote |
| Parameter Register D - Test value | Parameter Register D - Test wrote |
| Parameter Register E - Test value | Parameter Register E - Test wrote |
| Parameter Register F - Test value | Parameter Register F - Test wrote |
| Parameter Register G - Test value | Parameter Register G - Test wrote |

| **COMMAND 04H - Block Data Read Test**<br>This command allows for the reading of the 256 byte ISA test buffer from the board. The values read will be random unless preceded by the **Block Data Write Test** command, in which case the values read should reflect the data previously written. Data is read from the **Block Data Register**. All 256 bytes must be read unless the command is aborted with the **Abort Block Transfer** command. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - N/A | Parameter Register B - N/A |
| Parameter Register C - N/A | Parameter Register C - N/A |
| Parameter Register D - N/A | Parameter Register D - N/A |
| Parameter Register E - N/A | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

| **COMMAND 05H - Block Data Write Test**<br>This command is the companion to the **Block Data Read Test** command above. It allows transferring of 256 arbitrary bytes to an on-board buffer. Its only practical use is as an interface test or could be used by the user as a 256 byte temporary storage buffer. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - N/A | Parameter Register B - N/A |
| Parameter Register C - N/A | Parameter Register C - N/A |
| Parameter Register D - N/A | Parameter Register D - N/A |
| Parameter Register E - N/A | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

| **COMMAND 06H - Abort Block Transfer**<br>This command is used to prematurely end one of the block data transfer commands. It resets the on-board DMA and clears the **Data Request** bit. When an A2D data transfer is aborted with this command, the buffered data is permanently lost and the next data transfer will begin at an address that follows the blocks that would have been transferred had the command completed. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - N/A | Parameter Register B - N/A |
| Parameter Register C - N/A | Parameter Register C - N/A |
| Parameter Register D - N/A | Parameter Register D - N/A |
| Parameter Register E - N/A | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

| **COMMAND 07H - Request Pending Interrupt**<br>This command returns a channel number (0-35 or 38) of an A2D or DAC channel with a pending interrupt. A return of 38 indicates completion of a sequence command. A return value of 0FFH indicates that all requests have been delivered. A2D channel 0 has top priority, with DAC channel 3 having lowest priority. The channel number returned will always be the highest priority channel with a pending interrupt. A channel value, once read, clears the pending interrupt for that channel regardless of any further action to service the channel. ISR routines should always repeat this command until an 0FFH is returned. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - N/A | Parameter Register B - Channel number with pending interrupt |
| Parameter Register C - N/A | Parameter Register C - N/A |
| Parameter Register D - N/A | Parameter Register D - N/A |
| Parameter Register E - N/A | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

| **COMMAND 08H - Clear All Pending Interrupt Requests**<br>This command is used to clear all pending interrupt requests. It accomplishes in a single command what may take a number of Request Pending Interrupt commands to accomplish. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - N/A | Parameter Register B - N/A |
| Parameter Register C - N/A | Parameter Register C - N/A |
| Parameter Register D - N/A | Parameter Register D - N/A |
| Parameter Register E - N/A | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

| **COMMAND 09H - Set Internal Sampling Rate**<br>This command allows for setting the internal sampling rate from 0 (off) up to a maximum of about 20 KHz. When setting the sample rate, it must be recognized that there is overhead associated with increased speed and that the overall board throughput is decreased for an increased sample rate on a lesser number of channels. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - Sample Rate 0-20 KHz | Parameter Register B - N/A |
| Parameter Register C - N/A | Parameter Register C - N/A |
| Parameter Register D - N/A | Parameter Register D - N/A |
| Parameter Register E - N/A | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

The table below lists the various rates and the maximum number of channels that can be scheduled at that rate. The total board throughput is also shown.

| **Rate (KHz)** | **Max Channels** | **Overall Throughput** |
|---|---|---|
| 1 | 31 | 31 KHz |
| 2 | 15 | 30 KHz |
| 3 | 10 | 30 KHz |
| 4 | 7 | 28 KHz |
| 5 | 5 | 25 KHz |
| 6 | 4 | 24 KHz |
| 7 | 3 | 21 KHz |
| 8 | 3 | 24 KHz |
| 9 | 2 | 18 KHz |
| 10 | 1 | 10 KHz |
| 11 | 1 | 11 KHz |
| 12 | 1 | 12 KHz |
| 13 | 1 | 13 KHz |

| **COMMAND 0AH - Clear All Buffer Allocations**<br>This command returns all allocated buffer memory to the buffer pool. This command is very useful whenever it's necessary to resize a channel's buffer or otherwise alter the memory buffer allocations. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - N/A | Parameter Register B - N/A |
| Parameter Register C - N/A | Parameter Register C - N/A |
| Parameter Register D - N/A | Parameter Register D - N/A |
| Parameter Register E - N/A | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

There is no command to return the buffer memory of a single channel. This command is the only avenue for restructuring the allocation of the available memory.

| **COMMAND 0BH - Set A2D Buffer Memory Allocation**<br>This command allows the user to specify the number of blocks (256 bytes) that will be allocated to the specified channel to use as buffer memory. Specifying buffer memory on a per channel basis allows maximum versatility and minimum overhead for the host processor especially when higher sampling rates are specified. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - Channel Number (0-31) | Parameter Register B - N/A |
| Parameter Register C - Block Count Request | Parameter Register C - N/A |
| Parameter Register D - N/A | Parameter Register D - N/A |
| Parameter Register E - N/A | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

NOTES:
1. Requesting more than the available remaining blocks will result in a **bad command** error.
2. Use command 0DH to interrogate the system as to the number of block currently free.
3. Use command OAH to reset the block allocations.

| **COMMAND 0CH - Set DAC Buffer Memory Allocation**<br>This command allows the user to specify the number of blocks (256 bytes) that will be allocated to the specified DAC channel to use as an output buffer. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - Channel Number (0-3) | Parameter Register B - N/A |
| Parameter Register C - Block Count Request | Parameter Register C - N/A |
| Parameter Register D - N/A | Parameter Register D - N/A |
| Parameter Register E - N/A | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

NOTES:
1. Requesting more than the available remaining blocks will result in a **bad command** error.
2. Use command 0DH to interrogate the system as to the number of block currently free.
3. Use command OAH to reset the block allocations.

| **COMMAND 0DH - Get Free Allocation Blocks**<br>This command returns the number of blocks (256 bytes) that are currently free for allocation to A2D or DAC channel buffers. Use command 0AH to reset all allocated blocks to the free state. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - N/A | Parameter Register B - Count of free blocks |
| Parameter Register C - N/A | Parameter Register C - N/A |
| Parameter Register D - N/A | Parameter Register D - N/A |
| Parameter Register E - N/A | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

| COMMAND 0EH - Write EEPROM Word<br>This command allows direct manipulation of the EEPROM data. It should be used with care as it is possible to overwrite factory calibration data resulting in unpredictable measurements. This function is used primarily during factory testing. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - EEPROM Address (0-63) | Parameter Register B - N/A |
| Parameter Register C - Upper 8 bits to write | Parameter Register C - N/A |
| Parameter Register D - Lower 8 bits to write | Parameter Register D - N/A |
| Parameter Register E - N/A | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

**WARNING:**
**Altering calibration data in the EEPROM can result in a very poorly functioning board that must be returned to the factory for recalibration.**

| COMMAND 0FH - Read EEPROM Word<br>This command allows direct manipulation of the data stored in onboard EEPROM. Although reads from the EEPROM should never alter the calibration or configuration data, the user should not count on any specific data at any particular location as future revisions could change the way data is stored in the EEPROM. This function is used primarily for factory testing. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - EEPROM Address (0-63) | Parameter Register B - N/A |
| Parameter Register C - N/A | Parameter Register C - Upper 8 bits of data read |
| Parameter Register D - N/A | Parameter Register D - Lower 8 bits of data read |
| Parameter Register E - N/A | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

| **COMMAND 10H - Set External Clock Rate**<br>This command enables an external sample clock to be applied to GPIO Port A Bit 0. This clock can be prescaled by an integral value 1 to 65535. The prescaler value is calculated by:<br>Prescale_value = 1000H - divisor<br><br>For example, to divide by 2:<br>1000H - 2H = FFFEH | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - High Byte of prescale value | Parameter Register B - N/A |
| Parameter Register C - Low Byte of prescale value | Parameter Register C - N/A |
| Parameter Register D - N/A | Parameter Register D - N/A |
| Parameter Register E - N/A | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

NOTE:
The external clock is mutually exclusive with the internal clock. Turning on the external clock will disable the internal clock.

| **COMMAND 11H - Reset System Logic**<br>This command allows software to reset the onboard micro-controller and all of the conversion hardware. It does not reload the actual gate logic. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - N/A | Parameter Register B - N/A |
| Parameter Register C - N/A | Parameter Register C - N/A |
| Parameter Register D - N/A | Parameter Register D - N/A |
| Parameter Register E - N/A | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

| **COMMAND 11H - Reset System Logic**<br>This command allows software to reset the onboard micro-controller and all of the conversion hardware. It does not reload the actual gate logic. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - N/A | Parameter Register B - N/A |
| Parameter Register C - N/A | Parameter Register C - N/A |
| Parameter Register D - N/A | Parameter Register D - N/A |
| Parameter Register E - N/A | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

| **COMMAND 12H - Get Firmware Revision**<br>This command retrieves the revision numbers of the onboard firmware. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - N/A | Parameter Register B - Major Revision No. |
| Parameter Register C - N/A | Parameter Register C - Minor Revision No. |
| Parameter Register D - N/A | Parameter Register D - Sub Revision No. |
| Parameter Register E - N/A | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

**GPIO Commands**

| **COMMAND 20H - Read GPIO Channel A**<br>This command reads and returns the state of the input pins for the GPIO channel A port. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - N/A | Parameter Register B - Value at Channel A |
| Parameter Register C - N/A | Parameter Register C - N/A |
| Parameter Register D - N/A | Parameter Register D - N/A |
| Parameter Register E - N/A | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

| **COMMAND 21H - Read GPIO Channel B**<br>This command reads and returns the state of the input pins for the GPIO channel B port. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - N/A | Parameter Register B - Value at Channel B |
| Parameter Register C - N/A | Parameter Register C - N/A |
| Parameter Register D - N/A | Parameter Register D - N/A |
| Parameter Register E - N/A | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

| **COMMAND 22H - Write GPIO Channel A**<br>This command outputs the specified value to the channel A I/O pins. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - Value to write | Parameter Register B - N/A |
| Parameter Register C - N/A | Parameter Register C - N/A |
| Parameter Register D - N/A | Parameter Register D - N/A |
| Parameter Register E - N/A | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

| **COMMAND 23H - Write GPIO Channel B**<br>This command outputs the specified value to the channel B I/O pins. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - Value to write | Parameter Register B - N/A |
| Parameter Register C - N/A | Parameter Register C - N/A |
| Parameter Register D - N/A | Parameter Register D - N/A |
| Parameter Register E - N/A | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

| **COMMAND 24H - Set GPIO Direction**<br><br>This command allows setting either GPIO channel for input or output. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - Channel (0=A, 1=B) | Parameter Register B - N/A |
| Parameter Register C - Direction (0=input, 1=output) | Parameter Register C - N/A |
| Parameter Register D - N/A | Parameter Register D - N/A |
| Parameter Register E - N/A | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

## A2D Commands

| **COMMAND 30H - Convert Single Channel**<br>This command schedules a single conversion for the specified channel. The mode must have been specified prior to issuing this command. The actual conversion may not actually occur until the next clock period when an internal or external sampling clock is enabled. Polling the A2D Conversion bit (Bit 1) in the status register will signal when the conversion is complete. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - Channel No. (0-31) | Parameter Register B - Channel 0 |
| Parameter Register C - N/A | Parameter Register C - Data - High Byte |
| Parameter Register D - N/A | Parameter Register D - Data - Low Byte |
| Parameter Register E - N/A | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

| **COMMAND 31H - Convert Single Channel w/Interrupt**<br>This command schedules a single conversion for the specified channel. The mode must have been specified prior to issuing this command. The actual conversion may not actually occur until the next clock period. This command completes before the conversion data is available and a separate command, “Read Conversion Data”, is required to retrieve the data. An interrupt occurs once the data has been converted and is available. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - Channel No. (0-31) | Parameter Register B - N/A |
| Parameter Register C - N/A | Parameter Register C - N/A |
| Parameter Register D - N/A | Parameter Register D - N/A |
| Parameter Register E - N/A | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

| **COMMAND 32H - Start Block Conversion**<br>This command schedules a series of block conversions. The time interval and the number of blocks is also specified. The conversion mode and range should be set up prior to issuing this command. This command is actually very impractical because there's no way to tell when there's data available. This command is provided for interactive purposes only. Use the "Start Block Conversion w/Interrupt" for real applications. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - Channel No. (0-31) | Parameter Register B - N/A |
| Parameter Register C - Frequency High | Parameter Register C - N/A |
| Parameter Register D - Frequency Low | Parameter Register D - N/A |
| Parameter Register E - Block count | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

NOTE:
The frequency parameter is a value from 1 to 65535. It is incremented by 1 on each periodic clock tick. When it reaches 65536, a conversion is performed and the value is reloaded into the counter. Therefore to have a conversion performed on each tick, a value of 65535 would be specified. To have a conversion every 10 ticks a value of 65525 would be specified, etc. The block count value indicates the number of blocks (256 bytes/128 conversions that will be converted.

| **COMMAND 33H - Start Block Conversion w/Interrupt**<br>This command schedules a series of block conversions. The time interval and the number of blocks is also specified. The conversion mode and range should be set up prior to issuing this command. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - Channel No. (0-31) | Parameter Register B - N/A |
| Parameter Register C - Frequency High | Parameter Register C - N/A |
| Parameter Register D - Frequency Low | Parameter Register D - N/A |
| Parameter Register E - Block count | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

NOTE:
The frequency parameter is a value from 1 to 65535. It is incremented by 1 on each periodic clock tick. When it reached 65536 a conversion is performed and the value is reloaded into the counter. Therefore to have a conversion performed on each tick a value of 65535 would be specified. To have a conversion every 10 ticks a value of 65525 would be specified, etc.

The block count value indicates the number of blocks (256 bytes/128 conversions) that will be converted. An interrupt will be generated when the specified number of blocks have been converted.

| **COMMAND 34H - Read Block Data**<br>This command allows reading of the buffered conversion data for the specified channel. A block count is also specified. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - Channel No. (0-31) | Parameter Register B - N/A |
| Parameter Register C - Frequency High | Parameter Register C - N/A |
| Parameter Register D - Frequency Low | Parameter Register D - N/A |
| Parameter Register E - Block count | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

NOTE:
The data is moved from the **Block Data Register**. The **Data Request** bit will remain set in the status register until all of the specified data has been read.

| **COMMAND 35H - Set Channel Mode**<br>This command allows for the specification of the operating mode of the specified channel. The operating mode can be saved to EEPROM and restored on power-up if desired. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - Channel No. (0-31) | Parameter Register B - N/A |
| Parameter Register C - Mode byte (0-1FH) | Parameter Register C - N/A |
| Parameter Register D - N/A | Parameter Register D - N/A |
| Parameter Register E - N/A | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

*Mode Byte Definition*

Bits 7-5: Reserved (0)
Bits 4 : 0=Single-ended, 1=Differential
Bits 3 : 0=Normal Polarity, 1=Reverse Polarity
Bits 2-0: Range 000 = 0-1.25V
001 = 0-2.50V
010 = 0-5.00V
011 = 0-10.0V
100 = ± 1.25V
101 = ± 2.50V
110 = ± 5.00V
111 = ± 10.0V

NOTE:
Differential mode can only be specified on even channel numbers (i.e. 0, 2, 4, 6, etc.) Setting a channel to differential disables the next channel number.

| **COMMAND 36H - Read Channel Mode**<br>This command allows the programmer to read the current mode byte associated with the specified channel. This can be used to check that the mode has been properly set. See the **Set Channel Mode** command for mode byte bit definitions. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - Channel No. (0-31) | Parameter Register B - Channel No. |
| Parameter Register C - N/A | Parameter Register C - Mode Byte |
| Parameter Register D - N/A | Parameter Register D - N/A |
| Parameter Register E - N/A | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

| **COMMAND 37H - Stop Block Conversion**<br>This command stops block conversions on the specified channel. This is useful when it's desirous to reprogram the channel for a different, mode, rate, etc. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - Channel No. (0-31) | Parameter Register B - N/A |
| Parameter Register C - N/A | Parameter Register C - N/A |
| Parameter Register D - N/A | Parameter Register D - N/A |
| Parameter Register E - N/A | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

| **COMMAND 38H - Save Mode Data to EEPROM**<br>This command saves the channel modes to the onboard EEPROM which will be retrieved by the board at power up. The mode for ALL channels is saved with this command. There is not a way to save a single channel only. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - N/A | Parameter Register B - N/A |
| Parameter Register C - N/A | Parameter Register C - N/A |
| Parameter Register D - N/A | Parameter Register D - N/A |
| Parameter Register E - N/A | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

| **COMMAND 39H - Read Conversion Data**<br>This command reads out a single conversion for the specified channel. This command is primarily used with the **Convert Single Channel w/interrupt** command to read out the data after the interrupt occurs. It can also be used with block mode conversions but it will only return the first value in the buffer and the buffer pointers are not updated. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - Channel No. (0-31) | Parameter Register B - Channel 0 |
| Parameter Register C - N/A | Parameter Register C - Data - High Byte |
| Parameter Register D - N/A | Parameter Register D - Data - Low Byte |
| Parameter Register E - N/A | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

NOTE:
The **Block Data Register** is not used with this command. The data is returned in Register C: Register D.

| **COMMAND 3AH - Fast Convert Single Channel**<br>This command schedules a single A2D conversion for the specified channel. The mode must have been specified prior to issuing this command. This command is intended for repetitive conversions on the same channel as the internal multiplexers and other setup steps are bypassed in order to get faster follow-up conversions on the same channel. The setup of the channel mode along with at least one Single Channel Convert command should be executed prior to this command. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - Channel No. (0-31) | Parameter Register B - Channel 0 |
| Parameter Register C - N/A | Parameter Register C - Data - High Byte |
| Parameter Register D - N/A | Parameter Register D - Data - Low Byte |
| Parameter Register E - N/A | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

| **COMMAND 3BH - Fast Convert Single Channel w/Interrupt**<br>This command schedules a single conversion for the specified channel. The mode must have been specified prior to issuing this command. The actual conversion may not actually occur until the next clock period. This command completes before the conversion data is available and a separate command, **Read Conversion Data**, is required to retrieve the data. An interrupt occurs once the data has been converted and is available. This command like the previous one is intended for repetitive manual conversions on the same channel. This command bypasses the normal front-end setup that ordinarily occurs upon a request for an A2D conversion. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - Channel No. (0-31) | Parameter Register B - N/A |
| Parameter Register C - N/A | Parameter Register C - N/A |
| Parameter Register D - N/A | Parameter Register D - N/A |
| Parameter Register E - N/A | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

| **COMMAND 3CH - Stop All Block Conversions**<br>This command stops all currently scheduled conversions. This is useful when a new set of samples or rates is needed and is a quick command to start new conversion schedules from scratch. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - N/A | Parameter Register B - Channel No. |
| Parameter Register C - N/A | Parameter Register C - Data - High Byte |
| Parameter Register D - N/A | Parameter Register D - Data - Low Byte |
| Parameter Register E - N/A | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

| **COMMAND 3DH - Retrieve Calibration Values for Mode**<br>This command returns the calibration values used with the mode specified. This is primarily used for initial board factory calibration but may also be needed for troubleshooting or fine tuning the calibration. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - Mode value (0-31) | Parameter Register B - Mode Value |
| Parameter Register C - N/A | Parameter Register C - FS Trim Value-High Byte |
| Parameter Register D - N/A | Parameter Register D - FS Trim Value-Low Byte |
| Parameter Register E - N/A | Parameter Register E - Zero Offset-High Byte |
| Parameter Register F - N/A | Parameter Register F - Zero Offset-Low Byte |
| Parameter Register G - N/A | Parameter Register G - N/A |

| **COMMAND 3EH - Set Calibration Values for Mode**<br>This command sets the internal trim and offset DACs for the specified mode to the provided values. This command is used primarily by factory calibration routines. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - Mode value (0-31) | Parameter Register B - Mode Value |
| Parameter Register C - FS Trim Value-High Byte | Parameter Register C - N/A |
| Parameter Register D - FS Trim Value-Low Byte | Parameter Register D - N/A |
| Parameter Register E - Zero Offset-High Byte | Parameter Register E - N/A |
| Parameter Register F - Zero Offset-Low Byte | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

| **COMMAND 3FH - Save Calibration Values to EEPROM**<br>This command saves the calibration data for ALL modes to the onboard EEPROM. This calibration data will be retrieved at the next power up. This command is used by the factory to save the original calibration values derived during production calibration. A user program should NEVER alter these factory settings unless so instructed by WinSystems' technical support. It is possible to make the board, basically unusable, if bad values are saved in the calibration EEPROM. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - N/A | Parameter Register B - N/A |
| Parameter Register C - N/A | Parameter Register C - N/A |
| Parameter Register D - N/A | Parameter Register D - N/A |
| Parameter Register E - N/A | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

| **COMMAND 49H - Start A2D Conversion Sequence**<br>This command instructs the controller to run a sequence of conversions starting with the Register **B** parameter (0-31) and ending with the Register **C** parameter (0-31). An interrupt occurs when the sequence is complete. A request interrupt will return a 26H for the channel number. The data is then read out using the Block Data register following a Read Sequence Data command. Note that start channel must be less than end channel. This command can be used to convert from from 2 to 32 successive channels. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - Start Channel | Parameter Register B - N/A |
| Parameter Register C - End Channel | Parameter Register C - N/A |
| Parameter Register D - N/A | Parameter Register D - N/A |
| Parameter Register E - N/A | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

| **COMMAND 4AH - Read Sequence Data**<br>This command instructs the controller to present the buffered data from a start sequence command. The number of bytes retrieve will be twice the number of channels converted. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - N/A | Parameter Register B - N/A |
| Parameter Register C - N/A | Parameter Register C - N/A |
| Parameter Register D - N/A | Parameter Register D - N/A |
| Parameter Register E - N/A | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

NOTE:
The data is moved from the **Block Data Register**. The **Data Request** bit will remain set in the status register until all of the sequence data has been read.

**DAC Commands**

| **COMMAND 40H - Direct DAC Output**<br>This command sets the specified DAC channel (0-3) to the value provided. The DAC output range is also set by this command. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - DAC Channel No. (0-31) | Parameter Register B - N/A |
| Parameter Register C - DAC Mode Byte | Parameter Register C - N/A |
| Parameter Register D - DAC Data - High Byte | Parameter Register D - N/A |
| Parameter Register E - DAC Data - Low Byte | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

*DAC Mode Bytes*

08H = 0 – 5V
09H = 0 – 10V
0AH = ± 5V
0BH = ± 10V
0CH = ± 2.5V
0DH = -2.5V – 7.5V

NOTE:
The DAC data must be loaded such that the 12 bits of DAC data occupies the UPPER 12 bits of the 16-bit value loaded. The lower nibble will always be 0. This is for compatibility with the 16-bit DAC versions. It’s often easier to treat all DAC data as 16 bits and the lower nibble will be ignored on 12-bit DAC boards.

| **COMMAND 46H - Clear DAC Output**<br>This command clears the specified DAC outputs specified in the channel mask to 0. DAC channel 0 is bit 0, DAC Channel 1 is bit 1, etc. | |
|---|---|
| **Write** | **Read** |
| Parameter Register B - Channel mask (0-f) | Parameter Register B - N/A |
| Parameter Register C - N/A | Parameter Register C - N/A |
| Parameter Register D - N/A | Parameter Register D - N/A |
| Parameter Register E - N/A | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

| **COMMAND 47H - Enable External DAC Clear**<br>This command sets GPIO Channel A to input mode and enables GPIO Channel A Bit 1 to act as a master DAC clear. All DAC outputs return to 0 when GPIO Port A Bit 1 is set high. | |
| --- | --- |
| **Write** | **Read** |
| Parameter Register B - N/A | Parameter Register B - N/A |
| Parameter Register C - N/A | Parameter Register C - N/A |
| Parameter Register D - N/A | Parameter Register D - N/A |
| Parameter Register E - N/A | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

| **COMMAND 48H - Disable External DAC Clear**<br>This command disables GPIO Port A bit 1 as a master DAC clear input. GPIO Port A is left in input mode following the command but bit 1 may now serve as general I/O. | |
| --- | --- |
| **Write** | **Read** |
| Parameter Register B - N/A | Parameter Register B - N/A |
| Parameter Register C - N/A | Parameter Register C - N/A |
| Parameter Register D - N/A | Parameter Register D - N/A |
| Parameter Register E - N/A | Parameter Register E - N/A |
| Parameter Register F - N/A | Parameter Register F - N/A |
| Parameter Register G - N/A | Parameter Register G - N/A |

**Command Summary**

## Control and Diagnostic Commands

00H - Clear Interrupt
01H - Set IRQ Number
02H - Test Interrupt
03H - Parameter Register Test
04H - Block Data Read Test
05H - Block Data Write Test
06H - Abort Block Transfer
07H - Request Pending Interrupt
08H - Clear All pending Interrupt Requests
09H - Set Sample Rate
0AH - Clear All Buffer Allocations
0BH - Set A2D Buffer Memory Allocation
0CH - Set DAC Buffer Memory Allocation
0DH - Get Free Allocation Blocks
0EH - Write EEPROM Word
0FH - Read EEPROM Word
10H - Set External Sample Rate
11H - Reset System Logic
12H - Get Firmware Version

## GPIO Commands

20H - Read GPIO Channel A
21H - Read GPIO Channel B
22H - Write GPIO Channel A
23H - Write GPIO Channel B
24H - Set GPIO Direction

## A2D Commands

30H - Convert Single Channel
31H - Convert Single Channel w/interrupt
32H - Start Block Conversion
33H - Start Block Conversion w/interrupt
34H - Read Block Data
35H - Set Channel Mode
36H - Read Channel Mode
37H - Stop Block Conversion
38H - Save Mode Data to EEPROM
39H - Read Conversion Data
3AH - Fast Convert Single
3BH - Fast Convert Single w/interrupt
3CH - Stop All Block Conversions
3DH - Retrieve Calibration Values for Mode
3EH - Set Calibration Values for Mode
3FH - Save Calibration Values to EEPROM
49H - Start Sequence Conversions
4AH - Read Sequence Data

## DAC Commands

40H - Direct DAC Output
46H - Clear DAC Output
47H - Enable External DAC Clear
48H - Disable External DAC Clear

## PC/104 Bus Interface

The PCM-ADIO connects to the processor through the PC/104 bus connector at **J3**. The pin definitions for the 8-bit and 16-bit extension of **J3** are provided here for reference. Refer to the PC/104 Bus Specification for specific signal and mechanical specifications.

| Signal | Pins | Signal |
|---|---|---|
| GND | **D0 o o C0** | GND |
| MEMCS16# | **D1 o o C1** | SBHE# |
| IOCS16# | **D2 o o C2** | LA23 |
| IRQ10 | **D3 o o C3** | LA22 |
| IRQ11 | **D4 o o C4** | LA21 |
| IRQ12 | **D5 o o C5** | LA20 |
| IRQ15 | **D6 o o C6** | LA19 |
| IRQ14 | **D7 o o C7** | LA18 |
| DACK0# | **D8 o o C8** | LA17 |
| DRQ0 | **D9 o o C9** | MEMR# |
| DACK5# | **D10 o o C10** | MEMW# |
| DRQ5 | **D11 o o C11** | SD8 |
| DACK6# | **D12 o o C12** | SD9 |
| DRQ6 | **D13 o o C13** | SD10 |
| DACK7# | **D14 o o C14** | SD11 |
| DRQ7 | **D15 o o C15** | SD12 |
| +5V | **D16 o o C16** | SD13 |
| MASTER# | **D17 o o C17** | SD14 |
| GND | **D18 o o C18** | SD15 |
| GND | **D19 o o C19** | KEY |

| Signal | Pins | Signal |
|---|---|---|
| IOCHK# | **A1 o o B1** | GND |
| SD7 | **A2 o o B2** | RESET |
| SD6 | **A3 o o B2** | +5V |
| SD5 | **A4 o o B4** | IRQ9 |
| SD4 | **A5 o o B5** | -5V |
| SD3 | **A6 o o B6** | DRQ2 |
| SD2 | **A7 o o B7** | -12V |
| SD1 | **A8 o o B8** | SRDY# |
| SD0 | **A9 o o B9** | +12V |
| IOCHRDY | **A10 o o B10** | KEY |
| AEN | **A11 o o B11** | SMEMW# |
| SA19 | **A12 o o B12** | SMEMR# |
| SA18 | **A13 o o B13** | IOW# |
| SA17 | **A14 o o B14** | IOR# |
| SA16 | **A15 o o B15** | DACK3# |
| SA15 | **A16 o o B16** | DRQ3 |
| SA14 | **A17 o o B17** | DACK1# |
| SA13 | **A18 o o B18** | DRQ1 |
| SA12 | **A19 o o B19** | REFRESH# |
| SA11 | **A20 o o B20** | BCLK |
| SA10 | **A21 o o B21** | IRQ7 |
| SA9 | **A22 o o B22** | IRQ6 |
| SA8 | **A23 o o B23** | IRQ5 |
| SA7 | **A24 o o B24** | IRQ4 |
| SA6 | **A25 o o B25** | IRQ3 |
| SA5 | **A26 o o B26** | DACK2# |
| SA4 | **A27 o o B27** | TC |
| SA3 | **A28 o o B28** | BALE |
| SA2 | **A29 o o B29** | +5V |
| SA1 | **A30 o o B30** | OSC |
| SA0 | **A31 o o B31** | GND |
| GND | **A32 o o B32** | GND |

# = Active Low Signal

**NOTES**:
1. Rows C and D are not required on 8-bit modules.
2. B10 and C19 are key locations. WinSystems uses key pins as connections to GND.
3. Signal timing and function are as specified in ISA specification.
4. Signal source/sink current differ from ISA values.

## PCM-ADIO Programming Reference

### Introduction

Previous sections of this manual have documented the PCM-ADIO register set and the command set expected by the onboard microcontroller firmware. This section introduces a library of **C** language functions that may be compiled and linked with user code. Alternately, this library may be used for its source code and a subset of these functions may be extracted and the desired functionality included directly in the application code as desired. The source listings for all of the documented functions are shown in the C Source Code Listings and are also included in source form on the accompanying driver/utilities diskette. These functions were tested and compiled under MS-DOS/ROM-DOS/Win-98 Command prompt, with the Borland C/C++ compiler Version 3.1.

There should be very little effort needed to port these functions to other O/S environments or compilers as needed. Check with the WinSystems' applications Engineering department for the status on any other operating systems driver support which may have become available since this manual was created.

### Library Usage

Using the supplied C function library is accomplished by including the supplied header file ADIO_LIB.H and compiling the ADIO_LIB.C along with the application code. The header file exports 3 globals that will be of use by the application.

unsigned adio_base_address[4]

char *adio_error_string

int adio_error_code

*adio_base_address[]* is an array of I/O addresses with an entry for each of the four supported boards. Before any access to the library functions is allowed a base address **MUST** be specified for the board(s) in use such as :

**adio_base_adress[BOARDNUM] = 0x300;**

where BOARDNUM is a value from 0 to 3 and 0x300 is replaced with the actual base address for which the board is jumpered.

The other two globals, *adio_error_code*, and *adio_error_string* are used for error detection and error handling. All of the supplied library function will set or clear adio_error_code before returning. Checking *adio_error_code* for a nonzero value after any library function call is the most consistent way to check for error conditions. If an error condition occurs within the library function, a diagnostic error message will be formatted into *adio_error_string* which may be displayed, logged, or handled, in any manner desired by the application program.

The following section defines all of the functions supported by the library.

# Function Definitions

### DAC Commands

| **CLEAR_INTERRUPT** – Clears a pending interrupt |
|---|
| Syntax<br><br>**int adio_clear_interrupt(int boardnum)**<br><br>Description<br><br>*This command clears any pending interrupt and disasserts the IRQ line.  It must be used at the conclusion of any interrupt service routines to allow for future interrupts to occur.*<br><br>Argument(s):<br>boardnum (0-3) – The board number for this command.<br><br>Return value(s):<br>0 = Command Completed (without error)<br>1 = Command Timeout Error |

| **SET_IRQ** – Specifies IRQ Line to use |
|---|
| Syntax<br><br>**int adio_set_irq(int boardnum, int irq_num)**<br><br>Description<br><br>*This command allows for specification of the IRQ line to use when signaling the PC/104 computer of the completion of an event.*<br><br>Argument(s):<br>boardnum (0-3)– The board number for this command.<br><br>irq_num - The IRQ number to use (0-15).<br><br>Return value(s):<br>0 = Command Completed (without error)<br>1 = Command Timeout Error |

**TEST_INTERRUPT** – Tests the board interrupt

Syntax

**int adio_test_interrupt(int boardnum)**

Description

*This command is used primarily in system testing. It causes the IRQ line specified by SET_IRQ to be asserted under software control.*

Argument(s):
boardnum (0-3)– The board number for this command.

Return value(s):
0 = Command Completed (without error)
1 = Command Timeout Error

**PARAM_TEST** – Parameter register echo test

Syntax

**int adio_param_test(int boardnum, unsigned char *inregs, unsigned char *outregs)**

Description

*This command is used primarily for factory testing. An array of six (6) values, called inregs, is passed to the board and six (6) values, called outregs, are returned. The input vlaues and the output vlaues should match if the board is functioning properly. This function does NOT compare the values. It is up to the calling program to check the return values.*

Argument(s):
boardnum (0-3)– The board number for this command.
inregs - A pointer to an array of 6 unsigned integers.
outregs - A pointer to an array of 6 unsigned integers.

Return value(s):
0 = Command Completed (without error)
1 = Command Timeout Error

**BLOCK_READ_TEST** – Tests data transfers from the board

Syntax

**int adio_block_read_test(int boardnum, unsigned char *buff)**

Description

*This command tests the internal DMA transfer mechanism from the microcontroller's memory to the PC/104 bus data register. The transfer consists of 256 bytes from a special buffer on the board. This command is usually used in conjunction with the block_write_test function which would have written a known pattern to the internal buffer. Used together, these two functions test the block data transfer capabilities to and from the board.*

Argument(s):
boardnum (0-3)– The board number for this command.
buff - A pointer to a 256 byte unsigned character array.

Return value(s):
0 = Command Completed (without error)
1 = Command Timeout Error

**BLOCK_WRITE_TEST** – Tests data transfers to the board

Syntax

**int adio_block_write_test(int boardnum, unsigned char *buff)**

Description

*This command complements the block_read_test command. Together, they allow testing of the internal DMA data transfer capability to and from on-board buffer memory. This command is used to transfer a block of 256 arbitrary bytes into the on-board buffer. Those bytes can be read back using the block_read_test command.*

Argument(s):
boardnum (0-3)– The board number for this command.
buff - A pointer to a 256 byte unsigned character array.

Return value(s):
0 = Command Completed (without error)
1 = Command Timeout Error

**ABORT_BLOCK** – Abort a block transfer in progress

Syntax

**int adio_abort_block(int boardnum)**

Description

*This command allows for the early termination of a block transfer. It resets the DRQ bit to allow further commands to be processed.*

Argument(s):
boardnum (0-3)– The board number for this command.

Return value(s):
0 = Command Completed (without error)
1 = Command Timeout Error

**GET_PENDING_INT** – Get pending interrupt

Syntax

**int adio_get_pending_int(int boardnum)**

Description

*This command returns a value from 0 to 35 signaling that an A2D channel (0-31) or a DAC channel (32-35) has an interrupt pending. A return value of 38 indicates completion of a sequence command. A return value of FFH (255) indicates there are no more pending interrupts. This function always returns the highest priority interrupt (starting with A2D Channel 0). The process of reading out a pending interrupt number removes it from the queue. An interrupt service routine should call this function repeatedly until FFH is returned before issuing the clear_interrupt command and exiting. There is no error return from this command. Examining the global variable adio_error_ code may be used to determine if there was an error in the command.*

Argument(s):
boardnum (0-3)– The board number for this command.

Return value(s):
0-35, 38 = Channel number requiring service
FFH = No channels with pending interrupts

**CLEAR_PENDING_INTS** – Clear all pending interrupts

Syntax

**int adio_clear_pending_ints(int boardnum)**

Description

*This command clears all pending interrupts with a single call instead of repeated calling of get_pending_interrupt.*

Argument(s):
boardnum (0-3)– The board number for this command.

Return value(s):
0 = Command Completed (without error)
1 = Command Timeout Error

**SET_SAMPLE_RATE** – Set and enable internal sample clock rate

Syntax

**int adio_set_sample_rate(int boardnum, int sample_rate)**

Description

*This command turns on the internal sample rate clock to the rate specified. The rate is specified in KHz and can be up to 20 KHz. A value of 0 turns off the internal sample clock. Block A2D conversion commands will use this sample clock to schedule periodic conversions.*

Argument(s):
boardnum (0-3)– The board number for this command.
sample-rate - A value from 0-20 for the sample rate in KHz. 0=OFF.

Return value(s):
0 = Command Completed (without error)
1 = Command Timeout Error

| **CLR_BUFF_ALLOCATION** – Clear all buffer allocations |
| --- |
| Syntax<br><br>**int adio_clr_buff_allocation(int boardnum)**<br><br>Description<br><br>*This command releases any and all buffer memory allocated to the A2D and DAC channels. It is basically a reset for all memory allocation and allows a fresh start.*<br><br>Argument(s):<br>boardnum (0-3)– The board number for this command.<br><br>Return value(s):<br>0 = Command Completed (without error)<br>1 = Command Timeout Error |

| **SET_A2D_BUFF_ALLOCATION** – Set A2D Buffer Allocation |
| --- |
| Syntax<br><br>**int adio_set_a2d_buff_allocation(int boardnum, int channel_ num, int block_count)**<br><br>Description<br><br>*This command allows the limited onboard buffer memory to be allocated on a custom basis channel by channel. Requesting more blocks than what remains will result in a command error. Use the function get_free_allocation_blocks to determine how many free blocks are left. Use the command clr_buff_allocation to free all buffer memory.*<br><br>Argument(s):<br>boardnum (0-3)– The board number for this command.<br>channel_num - The a2d Channel number (0-31).<br>block_count - The number of 256 byte blocks requested.<br><br>Return value(s):<br>0 = Command Completed (without error)<br>1 = Command Timeout Error |

| **SET_DAC_BUFF_ALLOCATION** – Set DAC buffer allocation |
|---|
| Syntax<br><br>**int adio_set_dac_buff_allocation(int boardnum, int channel_num, int block_count)**<br><br>Description<br><br>*This command allows for the allocation of buffer memory to the DAC channels. The same buffer memory is used for A2D and DAC functions. Requesting more blocks than are available will result in a command error. Use the get_free_allocation_blocks to determine how many free blocks are available. Use the command clr_buff_allocation to free all buffer memory.*<br><br>Argument(s):<br>boardnum (0-3)– The board number for this command.<br>channel_num - The DAC Channel number (0-3).<br>block_count - The number of 256 byte blocks requested.<br><br>Return value(s):<br>0 = Command Completed (without error)<br>1 = Command Timeout Error |

| **GET_FREE_ALLOCATION_BLOCKS** – Get the number of unallocated blocks |
|---|
| Syntax<br><br>**int adio_get_free_allocation_blocks(int boardnum)**<br><br>Description<br><br>*This command returns the number of free allocation blocks currently available. The value returned by this function will decrease as blocks are allocated to A2D or DAC channels. Calling clr_buff_allocation will free ALL available blocks.*<br><br>Argument(s):<br>boardnum (0-3)– The board number for this command.<br><br>Return value(s):<br>0 = Error, or no blocks available. Check *adio_error_code* to verify.<br>>0 = Number of 256 byte blocks available for allocation |

**WRITE_EEPROM** – Write word to EEPROM

Syntax

**int adio_write_eeprom(int boardnum, int address, unsigned value)**

Description

*This function writes the value into the address on the EEPROM. This function is primarily used for factory testing and storage of initial calibration values. This function should be used with care as it's possible to overwrite factory calibration data resulting in unpredictable measurements.*

Argument(s):
boardnum (0-3)– The board number for this command.
address (0-63)- The EEPROM address.
value - The 16-bit data value to write.

Return value(s):
0 = Command Completed (without error)
1 = Command Timeout Error

**READ_EEPROM** – Read word from the EEPROM

Syntax

**unsigned adio_read_eeprom(int boardnum, int address)**

Description

*This function returns a 16-bit value representing the contents of the EEPROM cell at the specified address. The value is only valid if the global variable adio_error_code is equal to zero following the function return.*

Argument(s):
boardnum (0-3)– The board number for this command.
address (0-63)- The EEPROM address from which to retrieve data.

Return value(s):
16-bit unsigned EEPROM value

**SET_EXTERNAL_RATE** – Set and enable external sampling clock

Syntax

**int adio_set_external_rate(int boardnum, unsigned rate)**

Description

*This function enables an external sampling clock to be applied at GPIO PORT A, Bit 0. This clock is prescaled by the specified value such that a prescale value of 65535 = Clock/1, 65534 = Clock/2….etc. 0 = OFF. This clock input is mutually exclusive and cannot be used in addition to an internal sample clock. This function also changes the port A direction to input. The maximum useable input frequency after pre-scaling should not exceed 20 KHz.*

Argument(s):
boardnum (0-3)– The board number for this command.
rate - Prescaler value from 0 (off) to 65535 (divide by 1)

Return value(s):
0 = Command Completed (without error)
1 = Command Timeout Error

## DAC Commands

| **READ_GPIO** – Read a byte from GPIO port |
| --- |
| <u>Syntax</u><br><br>**unsigned char adio_read_gpio(int boardnum, int channel)**<br><br><u>Description</u><br><br>*This function returns the byte value read from the specified GPIO port. The value is only valid if the global adio_error_code is zero following this call.*<br><br><u>Argument(s)</u>:<br>boardnum (0-3)– The board number for this command.<br>channel (0=A, 1=B)- The GPIO port number.<br><br><u>Return value(s)</u>:<br>Byte value read from the specified port. |

| **WRITE_GPIO** – Write a byte from GPIO port |
| --- |
| <u>Syntax</u><br><br>**int adio_write_gpio(int boardnum, int channel, unsigned char value)**<br><br><u>Description</u><br><br>*This function writes the specified value to the GPIO channel specified in the command.*<br><br><u>Argument(s)</u>:<br>boardnum (0-3)– The board number for this command.<br>channel (0=A, 1=B) - The GPIO port number.<br>value - The 8-bit value to be written to the port.<br><br><u>Return value(s)</u>:<br>0 = Command Completed (without error)<br>1 = Command Timeout Error |

**SET_GPIO_DESCRIPTION** – Set a GPIO Channel for input or output

Syntax

**int adio_set_gpio(int boardnum, int channel, int direction)**

Description

*This function sets the I/O directin for the specified GPIO port.  All bits in a port are either inputs or outputs.*

Argument(s):
boardnum (0-3)– The board number for this command.
channel (0=A, 1=B) - The GPIO port number.
direction- The port direction.

Return value(s):
0 = Command Completed (without error)
1 = Command Timeout Error

## A2D Commands

| **CONVERT_SINGLE** – Start a single A2D Conversion |
|---|
| <u>Syntax</u><br><br>**int adio_convert_single(int boardnum, int channel)**<br><br><u>Description</u><br><br>*This function schedules a conversion on the specified board and channel. This function returns immediately after the command is accepted by the onboard controller. If there is not an external or internal sample clock enabled, the conversion will be started immediately otherwise the conversion will start on the next sample clock. Use the function wait_convert_complete to see if the conversion if finished and read_single_data to read out the conversion data. The conversion mode must have been previously set either by default at power-up or by a call to set_channel_mode.*<br><br><u>Argument(s)</u>:<br>boardnum (0-3)– The board number for this command.<br>channel (0-31)- The A2D channel number for the conversion.<br><br><u>Return value(s)</u>:<br>0 = Command Completed (without error)<br>1 = Command Timeout Error |

| **WAIT_CONVERT_COMPLETE** – Wait for a single conversion to complete |
|---|
| <u>Syntax</u><br><br>**int adio_wait_convert_complete(int boardnum)**<br><br><u>Description</u><br><br>*This function is to be used ONLY with convert_single. It is used to wait for the conversion to complete so that the data can be read. It is also desirable to use the convert_single function only when there is no sample rate enabled. This function can easily result in a timeout while waiting for a conversion to be scheduled at some future clock tick.*<br><br><u>Argument(s)</u>:<br>boardnum (0-3)– The board number for this command.<br><br><u>Return value(s)</u>:<br>0 = Conversion Completed (without error)<br>1 = Conversion in progress or an argument error. Check *adio_error_code* to verify. |

**CONVERT_SINGLE_INT** – Schedule an A2D Conversion, Interrupt when complete

Syntax

**int adio_convert_single_int(int boardnum, int channel)**

Description

*This function schedules a conversion on the specified board and channel. This function returns immediately after the command is accepted by the onboard controller. If there is not an external or internal sample clock enabled, the conversion will be started immediately, otherwise the conversion will start on the next sample clock. An interrupt will be generated when the conversion is completed. Use the function read_single_data to read out the conversion data. The conversion mode must have been previously set either by default at power-up or by a call to set_channel_mode.*

Argument(s):
boardnum (0-3)– The board number for this command.
channel (0-31)- The channel number for the conversion.

Return value(s):
0 = Command Completed (without error)
1 = Command Timeout Error

**READ_SINGLE_DATA** – Read the conversion data for a single conversion

Syntax

**unsigned adio_read_single_data(int boardnum)**

Description

*This function returns the conversion data resulting from a convert_single or a convert_single_int call. The data is invalid if adio_ error_code is nonzero.*

Argument(s):
boardnum (0-3)– The board number for this command.

Return value(s):
16-bit conversion data. *adio_error_code* will be nonzero if an error has occurred.

**START_BLOCK_CONVERT** – Start a series of timed conversions

Syntax

**int adio_start_block_convert(int boardnum, int channel, unsigned rate, int block_count)**

Description

*This function schedules a number of block (128 samples) conversions at a specified sample rate and buffers up the results until all of the blocks have been completed or until the allocated buffer memory is full. The conversion mode and range must have been previously specified with a set_channel_mode call. This particular function has no real practical use as there is no way to tell when the series of block conversions have been completed. It is included for use in an interactive mode only. Use the start_block_convert_int function which signals that the conversions are complete with an interrupt. The counter value is incremented by one on each tick of the sample rate clock (internal or external) when the count overflows from 65535 to 0 it indicates that a conversion is to be performed. To convert once per tick use 65535, once every two ticks use 65534, once every ten ticks use 65526 etc.*

Argument(s):
boardnum (0-3)– The board number for this command.
channel (0-31)- The A2D channel number for the conversion.
rate (0-65535) - A counter value.
block_count - The number of blocks to convert.

Return value(s):
0 = Command Completed (without error)
1 = Command Timeout Error

**START_BLOCK_CONVERT_INT** – Start a series of timed conversions, Interrupt when complete

Syntax

**int adio_start_block_convert(int boardnum, int channel, unsigned rate, int block_count)**

Description

*This function schedules a number of block (128 samples) conversions at a specified sample rate and buffers up the results until all of the blocks have been completed or until the allocated buffer memory is full. The conversion mode and range must have been previously specified with a set_channel_mode call. This function signals that the conversions are complete with an interrupt. The counter value is incremented by one on each tick of the sample rate clock (internal or external) when the count overflows from 65535 to 0, it indicates that a conversion is to be performed. To convert once per tick use 65535, once every two ticks use 65534, once every ten ticks use 65526 etc.*

Argument(s):
boardnum (0-3)– The board number for this command.
channel (0-31)- The A2D channel number for the conversion.
rate (0-65535) - A counter value.
block_count - The number of blocks to convert.

Return value(s):
0 = Command Completed (without error)
1 = Command Timeout Error

**READ_BLOCK_DATA** – Read out a series of block conversion data

Syntax

**int adio_read_block_data(int boardnum, int channel, int block_count, unsigned *buff)**

Description

*This function loads the caller's data buffer 'buff' with the contents of the onboard channel buffer. This data would ordinarily be the result of the completion of a block conversion command. The data is loaded into the buffer in a low-byte, high-byte order such that the user memory area can be accessed as an array of unsigned values.*

Argument(s):
boardnum (0-3)– The board number for this command.
channel (0-31)- The A2D channel number for the conversion.
block_count - The number of blocks to convert.
*buff - A pointer to a memory area large enough to hold the number of blocks specified.

Return value(s):
0 = Command Completed (without error)
1 = Command Timeout Error

**SET_CHANNEL_MODE** – Set the Mode for an A2D Channel

Syntax

**int adio_set_channel_mode(int boardnum, int channel, int mode)**

Description

*This command sets the mode for the specified A2D channel. Differential modes can only be set for even number channels (0,2,4,6 etc). Setting a channel to differential mode disables the next channel. The mode setting is NOT automatically saved by this command. Calling save_channel_modes will save the modes into the onboard EEPROM and the modes will be set automatically at power-up to the saved values.*

Argument(s):
boardnum (0-3)– The board number for this command.
channel (0-31)- The A2D channel number for the conversion.
block_count - The number of blocks to convert.
mode - an 8-bit mode value according to the following:

| | | |
|---|---|---|
| Bits 7-5: | Reserved (0) | |
| Bits 4 : | 0=Single-ended, 1=Differential | |
| Bits 3 : | 0=Normal Polarity, 1=Reverse Polarity | |
| Bits 2-0: | Range | 000 = 0-1.25V |
| | | 001 = 0-2.50V |
| | | 010 = 0-5.00V |
| | | 011 = 0-10.0V |
| | | 100 = ± 1.25V |
| | | 101 = ± 2.50V |
| | | 110 = ± 5.00V |
| | | 111 = ± 10.0V |

Return value(s):
0 = Command Completed (without error)
1 = Command Timeout Error

**READ_CHANNEL_MODE** – Read an A2D channel's operating mode

Syntax

**unsigned char adio_read_channel_mode(int boardnum, int channel)**

Description

*This function returns the current mode value (See set_channel_mode for bit definitions) for the specified channel. This value is either the power-up default or a value set by a previous call to set_channel_mode. It does NOT read the default value from the EEPROM.*

Argument(s):
boardnum (0-3)– The board number for this command.
channel (0-31)- The A2D channel number for the conversion.

Return value(s):
8-bit mode value. Value is invalid if *adio_error_code* is nonzero.

**STOP_BLOCK_CONVERSION** – Abort block conversions on a channel

Syntax

**int adio_stop_block_conversion(int boardnum, int channel)**

Description

*This function is used to terminate or abort a scheduled periodic conversion. The channel is returned to the idle state.*

Argument(s):
boardnum (0-3)– The board number for this command.
channel (0-31)- The A2D channel number.

Return value(s):
0 = Command Completed (without error)
1 = Command Timeout Error

**SAVE_CHANNEL_MODES** – Save the channel mode values to EEPROM

Syntax

**int adio_save_channel_modes(int boardnum)**

Description

*This function saves all of the current mode bytes (See set_channel_mode) to the onboard EEPROM. These values will be restored automatically at power-up.*

Argument(s):
boardnum (0-3)– The board number for this command.

Return value(s):
0 = Command Completed (without error)
1 = Command Timeout Error

**READ_CONVERSION_DATA** – Read the A2D conversion data

Syntax

**unsigned adio_read_conversion_data(int boardnum, int channel)**

Description

*This function reads a conversion value from the onboard buffer. It is used primarily with the convert_single_int command to read out the data after the interrupt occurs. It can also be used with block mode conversions but it will only return the first value in the buffer and the buffer pointers are not updated.*

Argument(s):
boardnum (0-3)– The board number for this command.
channel (0-31)- The A2D channel number for the conversion.

Return value(s):
16-bit conversion data.  Value is valid only if *adio_error_code* is zero.

**FAST_CONVERT** – Perform repeated conversions at an accelerated rate

Syntax

**int adio_fast_convert(int boardnum, int channel)**

Description

*This command is identical in function to convert_single with one exception. The internal conversion process bypasses the normal setup and hold times in the hardware, resulting in a significantly faster conversion completion. This function should only be used when a single channel is being repetitively converted and then only after following an initial convert_single command.*

Argument(s):
boardnum (0-3)– The board number for this command.
channel (0-31)- The A2D channel number for the conversion.

Return value(s):
0 = Command Completed (without error)
1 = Command Timeout Error

**FAST_CONVERT_INT** – Perform a repeated conversion at an accelerated rate.

Syntax

**int adio_fast_convert(int boardnum, int channel)**

Description

*This command is identical in function to convert_single_int except that it bypasses the normal setup and hold times in the hardware resulting in a significantly faster conversion time. This function should only be used when a single channel is being used for repetitive conversions and then only following at least one call to convert_single_int.*

Argument(s):
boardnum (0-3)– The board number for this command.
channel (0-31)- The A2D channel number for the conversion.

Return value(s):
0 = Command Completed (without error)
1 = Command Timeout Error

**STOP_BLOCK_CONVERSIONS** – Aborts all block conversions on a board

Syntax

**int adio_stop_block_conversions(int boardnum, int channel)**

Description

*This command is used to stop all scheduled block conversions on a board. It avoids repeated calls to stop_block_conversion for each channel.*

Argument(s):
boardnum (0-3)– The board number for this command.
channel (0-31)- The A2D channel number.

Return value(s):
0 = Command Completed (without error)
1 = Command Timeout Error

**GET_MODE_CAL_VALUES** – Retrieve calibration values for specified mode.

Syntax

**int adio_get_mode_cal_values(int boardnum, int mode, unsigned *trim, unsigned *zero)**

Description

*This function returns the zero offset and trim gain values for the specified mode. Board calibration is performed by the factory and is done on a mode by mode basis. This function is primarily used by factory calibration routines.*

Argument(s):
boardnum (0-3)– The board number for this command.
channel (0-31)- The A2D channel number for the conversion.
*trim - A pointer to an unsigned integer to receive the 12-bit trim DAC value.
*zero - A pointer to an unsigned integer to receive the 12-bit zero offset DAC value.

Return value(s):
0 = Command Completed (without error)
1 = Command Timeout Error

**SET_MODE_CAL_VALUES** – Retrieve calibration values for specified mode.

Syntax

**int adio_set_mode_cal_values(int boardnum, int mode, unsigned trim, unsigned zero)**

Description

*This function is used to calibrate the desired mode. Factory software adjusts the trim and zero values using a precision source so that the zero and full-scale values are accurate to published specifications for each supported mode. User code should never have a reason to change these values. Changes to the calibration values are temporary and are not restored at power-up unless the save_cal_values function is called to save all of the calibration values to the onboard EEPROM.*

Argument(s):
boardnum (0-3)– The board number for this command.
mode (0-31)- The mode number.  See *set_channel_mode* for bit definitions.
trim - The 12-bit full scale trim DAC value.
zero - The 12-bit zero offset DAC value.

Return value(s):
0 = Command Completed (without error)
1 = Command Timeout Error

**SET_CAL_VALUES** – Save all of the calibration values to EEPROM

Syntax

**int adio_set_cal_values(int boardnum)**

Description

*This function saves all of the calibration values for all of the operating modes to the onboard EERPROM. These saved values are automatically restored at power-up. A user program should NEVER alter the factory defaults unless so instructed by WinSystems' technical support. It is possible to make the board practically unusable if bad values are saved into the calibration EEPROM.*

Argument(s):
boardnum (0-3)– The board number for this command.

Return value(s):
0 = Command Completed (without error)
1 = Command Timeout Error

**START_SEQUENCE** – Start a sequence of A2D Conversions

Syntax

**int adio_start_sequence(int boardnum, int start_channel, int end_ channel)**

Description

*This function schedules a sequence of conversions beginning with start_ channel up to and including end_channel. The mode for each channel must have been set by a previous set_channel_mode command. The data is buffered until the sequence is complete. An interrupt is generated at the completion of the sequence. The channel number returned for a sequence intterupt is 26h. Use the adio_read_sequence command to retrieve the data from the buffer.*

Argument(s):
boardnum (0-3)– The board number for this command.
start_channel (0-30)- The first channel in sequence.
end_channel (1-31)- The last channel in sequence.

Return value(s):
0 = Command Completed (without error)
1 = Command Timeout Error

**READ_SEQUENCE** – Read the data from a sequence of A2D Conversions

Syntax

**int adio_read_sequence(int boardnum, unsigned *value)**

Description

*This function retrieves the data buffered up by the adio_start_sequence command. The data is loaded into the supplied buffer. The data is raw 16-bit converter results from each of the sequenced channels.*

Argument(s):
boardnum (0-3)– The board number for this command.
value - A pointer to an array of 16-bit values large enough for the number of channels converted.

Return value(s):
0 = Command Completed (without error)
1 = Command Timeout Error

## DAC Commands

**DIRECT_DAC_OUTPUT** – Set DAC Output Level

Syntax

**int adio_direct_dac_output(int boardnum, int channel, int mode, unsigned data)**

Description

*This function sets the desired DAC output channel to the range and value specified. Note that the output value is specified as 16-bits. This is for compatibility with boards with either 12-bit or 16-bit DACs installed. With 12-Bit DACs the lower nibble of the output value will be ignored.*

Argument(s):
boardnum (0-3)– The board number for this command.
channel (0-3)- The DAC channel number.
mode - The DAC output mode:
08H = 0-5V
09H = 0-10.0V
0AH = ±5.0V
0BH = ±10.0V
0CH = ±2.5V
0DH = -2.5V - +7.5V
value - The 16-bit output value.

Return value(s):
0 = Command Completed (without error)
1 = Command Timeout Error

**CLR_DAC_OUTPUT** – Clear one or more DAC channel outputs

Syntax

**int adio_clr_dac_output(int boardnum, int channel_mask)**

Description

*This function clears (sets to 0) any or all DAC channels as specified by the channel_mask argument.*

Argument(s):
boardnum (0-3)– The board number for this command.
channel_mask (Bit 0 = Channel 0, Bit 1 = Channel 1, etc.)- A 4-bit value with each bit corresponding to a DAC channel.

Return value(s):
0 = Command Completed (without error)
1 = Command Timeout Error

**ENABLE_EXT_DAC_CLR** – Enables the external DAC clear pin

Syntax

**int adio_enab_ext_dac_clr(int boardnum)**

Description

*This function sets the GPIO channel A to input and enables GPIO Channel A bit 1 to act as a master DAC clear. All DAC outputs return to 0 when this input pin is set high.*

Argument(s):
boardnum (0-3)– The board number for this command.

Return value(s):
0 = Command Completed (without error)
1 = Command Timeout Error

**DISABLE_EXT_DAC_CLR** – Disables the external DAC clear pin

Syntax

**int adio_disab_ext_dac_clr(int boardnum)**

Description

*This function disables GPIO Port A bit 1 from serving as a master DAC clear. Port A is left in the input mode following the command, but Bit 1 may now serve as general I/O.*

Argument(s):
boardnum (0-3)– The board number for this command.

Return value(s):
0 = Command Completed (without error)
1 = Command Timeout Error

## DAC Commands

| **WAIT_READY** – Waits for the board to be ready |
| --- |
| <u>Syntax</u><br><br>**int adio_wait_ready(int boardnum)**<br><br><u>Description</u><br><br>*This function is used internally by all of the other functions. A zero return indicates that the board is ready for a command or has completed the previous command. A return of 1 indicates that a timeout error has occurred and the board is not responding to commands.*<br><br><u>Argument(s)</u>:<br>boardnum (0-3)– The board number for this command.<br><br><u>Return value(s)</u>:<br>0 = Command Completed (without error)<br>1 = Command Timeout Error |

| **WAIT_DRQ** – Waits for Data Request |
| --- |
| <u>Syntax</u><br><br>**int adio_wait_drq(int boardnum)**<br><br><u>Description</u><br><br>*This function is used internally by the commands that require block transfers of data. A return of 0 indicates that the data transfer may proceed. A return of 1 indicates a timeout error has occurred.*<br><br><u>Argument(s)</u>:<br>boardnum (0-3)– The board number for this command.<br><br><u>Return value(s)</u>:<br>0 = Command Completed (without error)<br>1 = Command Timeout Error |

## Sample Programs

There were several sample programs planned for inclusion on the Driver/Utility diskette accompanying the PCM-ADIO. These sample can serve as working examples of using the board and the library functions. Refer to the documentation in the samples directory and the comments within the source code for further information.

## Summary

Links to the source code for all three programs as well as the I/O routines can be found in the Software Drivers & Examples Section of this manual. It is also provided in the C Source Code Listings Section of this manual.

# C Source Code Listings

```
/* ADIO_LIB.H Copyright 2004 WinSystems Inc. All Rights Reserved.
**********************************************************************
*
*   Name    : adio_lib.h
*
*   Project : PCM-ADIO
*
*   Purpose : User functions header file
*
*   Date    : March 30, 2006
*
*   Revision: 1.01
*
*   Author  : Steve Mottin
*
**********************************************************************
*
*   Revision History
*
*   Date        Revision    Description
*   -------     --------    ------------------------------------------
*   03/01/04    1.00        Intial
*   03/30/06    1.01         Added Sequence Commands
*
*   $Log: E:\source\pcm_adio\Library\adio_lib.i $
//
//    First Checkin for user programs.
//
//  Revision 1.1  by: Steve  Rev date: 3/30/2006 3:32:36 PM
//    Develpoment Alpha 03/30/2006
//
//  Revision 1.0  by: Steve  Rev date: 12/17/2003 9:07:08 AM
//    Initial revision.
//
//  $Endlog$
**********************************************************************
*/

int adio_wait_ready(int boardnum);
int adio_wait_drq(int boardnum);
int adio_wait_convert_complete(int boardnum);
int adio_clear_interrupt(int boardnum);
int adio_set_irq(int boardnum, int irqnum);
int adio_test_interrupt(int boardnum);
int adio_param_test(int boardnum, unsigned char *inregs, unsigned char *outregs);
int adio_block_read_test(int boardnum, unsigned char *buff);
int adio_block_write_test(int boardnum, unsigned char *buff);
int adio_abort_block(int boardnum);
int adio_get_pending_int(int boardnum);
int adio_clear_pending_ints(int boardnum);
int adio_set_sample_rate(int boardnum, int sample_rate);
int adio_clr_buff_allocation(int boardnum);
int adio_set_a2d_buff_allocation(int boardnum, int channel_num, int block_count);
int adio_set_dac_buff_allocation(int boardnum, int channel_num, int block_count);
int adio_get_free_allocation_blocks(int boardnum);
int adio_write_eeprom(int boardnum, int address, unsigned value);
unsigned adio_read_eeprom(int boardnum, int address);
int adio_set_external_rate(int boardnum, unsigned rate);
int adio_sys_reset(int boardnum);
int adio_get_firmware_version(int boardnum,unsigned *major, unsigned *minor, unsigned *sub);
unsigned char adio_read_gpio(int boardnum, int channel);
```

```
unsigned char adio_write_gpio(int boardnum, int channel, unsigned char value);
int adio_set_gpio_direction(int boardnum, int channel, int direction);
int adio_convert_single(int boardnum, int channel_num);
int adio_convert_single_int(int boardnum, int channel_num);
int adio_start_block_convert(int boardnum,int channel, unsigned rate, int block_count);
int adio_start_block_convert_int(int boardnum, int channel, unsigned rate, int block_count);
int adio_read_block_data(int boardnum, int channel_num,int block_count,char *buff);
int adio_set_channel_mode(int boardnum,int channel_num, int mode);
unsigned char adio_read_channel_mode(int boardnum, int channel_num);
int adio_stop_block_conversion(int boardnum, int channel_num);
int adio_save_channel_modes(int boardnum);
unsigned adio_read_conversion_data(int boardnum, int channel_num);
int adio_fast_convert(int boardnum, int channel_num);
int adio_fast_convert_int(int boardnum, int channel_num);
int adio_stop_block_conversions(int boardnum);
int adio_get_mode_cal_values(int boardnum,int mode_val, unsigned *trim_val, unsigned *gain_val);
int adio_set_mode_cal_values(int boardnum, int mode_val, unsigned trim_val, unsigned gain_val);
int adio_save_cal_values(int boardnum);
int adio_direct_dac_output(int boardnum,int dac_channel, int dac_mode, unsigned dac_data);
int adio_set_dac_mode(int boardnum, int dac_channel, int dac_mode);
int adio_start_dac_block(int boardnum, int dac_channel, unsigned dac_rate, unsigned block_count);
int adio_start_dac_block_int(int boardnum, int dac_channel, unsigned dac_rate, unsigned block_count);
int adio_load_dac_data(int boardnum, int dac_channel, unsigned block_count);
int adio_get_dac_mode(int boardnum, int dac_channel);
unsigned adio_read_single_data(int boardnum);
int adio_start_sequence(int boardnum, int start_channel, int end_channel);
int adio_read_sequence(int boardnum,int count,unsigned *value);

extern unsigned adio_base_address[4];
extern char adio_error_string[];
extern int adio_error_code;


/* ADIO_LIB.C Copyright 2004-2006 by WinSystems Inc. All rights Reserved
*********************************************************************
*
*   Name    : adio_lib.c
*
*   Project : PCM-ADIO
*
*   Purpose : User Library Functions
*
*   Date    : March 16, 2006
*
*   Revision: 1.01
*
*   Author  : Steve Mottin
*
*********************************************************************
*
*   Revision History
*
*   Date        Revision    Description
*   -------     --------    ------------------------------------------
*   03/01/04    1.00        Initial
*   03/16/06    1.01        Added new sequence functions
*
*   $Log: E:\source\pcm_adio\Library\adio_lib.d $
//
//    First Checkin for user programs.
//
//  Revision 1.2  by: Steve  Rev date: 3/30/2006 3:32:36 PM
//    Develpoment Alpha 03/30/2006
```

```
//
//  Revision 1.1  by: Steve  Rev date: 12/17/2003 9:20:24 AM
//    Added Parameter checks for clear_interrupt function.
//
//  Revision 1.0  by: Steve  Rev date: 12/17/2003 9:07:08 AM
//    Initial revision.
//
//  $Endlog$
*********************************************************************
*/

#include <stdio.h>
#include <dos.h>
#include "adio_cmd.h"

#define MAX_BOARD 3

#define ADIO_NO_ERROR 0
#define ADIO_NO_BASE_ADDRESS 1
#define ADIO_CMD_TIMEOUT_ERROR 2
#define ADIO_BAD_BOARD_NUM  3
#define ADIO_BAD_IRQ_NUM 4
#define ADIO_DRQ_TIMEOUT_ERROR 5
#define ADIO_BAD_SAMPLE_RATE 6
#define ADIO_BAD_CHANNEL_NUM 7
#define ADIO_BAD_EEPROM_ADDRESS 8
#define ADIO_BAD_GPIO_CHANNEL_NUM 9
#define ADIO_BAD_GPIO_DIRECTION 10
#define ADIO_BAD_A2D_CHANNEL_NUM 11
#define ADIO_BAD_A2D_MODE 12
#define ADIO_BAD_DAC_MODE 13
#define ADIO_BAD_DAC_CHANNEL 14
#define ADIO_CIP_TIMEOUT_ERROR 15
#define ADIO_BAD_DAC_MASK 16

#define ADIO_CMD_ERROR 255

int adio_wait_ready(int boardnum);

int adio_error_code;
char adio_error_string[80];

unsigned adio_base_address[4] = {0,0,0,0};


int adio_wait_ready(int boardnum)
{
unsigned long retry;
unsigned port;


   adio_error_code = ADIO_NO_ERROR;

   if(boardnum < 0 || boardnum > MAX_BOARD)
   {
      adio_error_code = ADIO_BAD_BOARD_NUM;
      sprintf(adio_error_string,"adio_wait_ready : Bad Board Number");
      return(1);
   }


   port = adio_base_address[boardnum];
```

```
    if(port == 0)
    {
        adio_error_code = ADIO_NO_BASE_ADDRESS;
        sprintf(adio_error_string,"wait_ready : no base address");
        return(1);
    }

    retry = 0xffffl;
    while(retry--)
    {
        if((inportb(port) & 1) == 0)
            return 0;

    }
    adio_error_code = ADIO_CMD_TIMEOUT_ERROR;
    sprintf(adio_error_string,"wait_ready : Command Timeout Error");
    return(1);
}

int adio_wait_drq(int boardnum)
{
unsigned long retry;
unsigned port;


    adio_error_code = ADIO_NO_ERROR;

    port = adio_base_address[boardnum];

    retry = 0xffffl;
    while(retry--)
    {
        if((inportb(port) & 0x20) != 0)
            return 0;

    }
    adio_error_code = ADIO_DRQ_TIMEOUT_ERROR;
    sprintf(adio_error_string,"wait_drq : DRQ Timeout error");
    return(1);
}


int adio_wait_convert_complete(int boardnum)
{
unsigned long retry;
unsigned port;


    adio_error_code = ADIO_NO_ERROR;

    port = adio_base_address[boardnum];

    retry = 0xffffl;
    while(retry--)
    {
        if((inportb(port) & 0x3) == 0)
            return 0;

    }
    adio_error_code = ADIO_CIP_TIMEOUT_ERROR;
    sprintf(adio_error_string,"wait_convert_complete : Conversion Timeout error");
    return(1);
}
```

```
int adio_clear_interrupt(int boardnum)
{
unsigned port;


   adio_error_code = ADIO_NO_ERROR;

   if(adio_wait_ready(boardnum))
      return(1);

   port = adio_base_address[boardnum];

   outportb(port,ADIO_CLR_INT);

   if(adio_wait_ready(boardnum))
      return(1);


   if(inportb(port) & 0x80)
   {
      adio_error_code = ADIO_CMD_ERROR;
      sprintf(adio_error_string,"adio_clear_interrupt : Command/Parameter error");
      return(1);
   }

   return 0;

}

int adio_set_irq(int boardnum, int irqnum)
{
unsigned port;

   adio_error_code = ADIO_NO_ERROR;

   if(irqnum < 0 || irqnum > 15)
   {
      adio_error_code = ADIO_BAD_IRQ_NUM;
      sprintf(adio_error_string,"adio_set_irq : Bad IRQ Number");
      return(1);
   }

   if(adio_wait_ready(boardnum))
      return(1);

   port = adio_base_address[boardnum];

   outportb(port+1,irqnum);

   outportb(port,ADIO_SET_IRQ);

   if(adio_wait_ready(boardnum))
      return(1);

   if(inportb(port) & 0x80)
   {
      adio_error_code = ADIO_CMD_ERROR;
      sprintf(adio_error_string,"adio_set_irq : Command/Parameter error");
      return(1);
   }
```

```
    return(0);
}

int adio_test_interrupt(int boardnum)
{
unsigned port;

    adio_error_code = ADIO_NO_ERROR;

    if(adio_wait_ready(boardnum))
        return(1);

    port = adio_base_address[boardnum];

    outportb(port,ADIO_TEST_IRQ);

    if(adio_wait_ready(boardnum))
        return(1);

    if(inportb(port) & 0x80)
    {
        adio_error_code = ADIO_CMD_ERROR;
        sprintf(adio_error_string,"adio_test_interrupt : Command/Parameter error");
        return(1);
    }

    return(0);
}


int adio_param_test(int boardnum, unsigned char *inregs, unsigned char *outregs)
{
unsigned port;
int x;

    adio_error_code = ADIO_NO_ERROR;

    if(adio_wait_ready(boardnum))
        return(1);

    port = adio_base_address[boardnum];

    for(x=0; x<6; x++)
        outportb(port+x+1,*inregs++);

    outportb(port,ADIO_PARAM_TEST);

    if(adio_wait_ready(boardnum))
        return(1);

    for(x=0; x<6; x++)
        *outregs++ = inportb(port+x+1);

    if(inportb(port) & 0x80)
    {
        adio_error_code = ADIO_CMD_ERROR;
        sprintf(adio_error_string,"adio_param_test : Command/Parameter error");
        return(1);
    }

    return(0);
}
```

```
int adio_block_read_test(int boardnum, unsigned char *buff)
{
unsigned port;
int x;

    adio_error_code = ADIO_NO_ERROR;

    if(adio_wait_ready(boardnum))
        return(1);

    port = adio_base_address[boardnum];

    outportb(port,ADIO_BLOCK_READ_TEST);

    if(adio_wait_ready(boardnum))
        return(1);

    if(adio_wait_drq(boardnum))
        return(1);

    for(x=0; x<256; x++)
        *buff++ = inportb(port+7);

    if(inportb(port) & 0x80)
    {
        adio_error_code = ADIO_CMD_ERROR;
        sprintf(adio_error_string,"adio_block_read_test : Command/Parameter error");
        return(1);
    }

    return(0);
}

int adio_block_write_test(int boardnum, unsigned char *buff)
{
unsigned port;
int x;

    adio_error_code = ADIO_NO_ERROR;

    if(adio_wait_ready(boardnum))
        return(1);

    port = adio_base_address[boardnum];

    outportb(port,ADIO_BLOCK_WRITE_TEST);

    if(adio_wait_ready(boardnum))
        return(1);

    if(adio_wait_drq(boardnum))
        return(1);

    for(x=0; x<256; x++)
        outportb(port+7,*buff++);

    if(inportb(port) & 0x80)
    {
        adio_error_code = ADIO_CMD_ERROR;
        sprintf(adio_error_string,"adio_block_write_test : Command/Parameter error");
        return(1);
    }
```

```
    return(0);
}


int adio_abort_block(int boardnum)
{
unsigned port;

    adio_error_code = ADIO_NO_ERROR;

    if(adio_wait_ready(boardnum))
        return(1);

    port = adio_base_address[boardnum];

    outportb(port,ADIO_ABORT_BLOCK);

    if(adio_wait_ready(boardnum))
        return(1);

    if(inportb(port) & 0x80)
    {
        adio_error_code = ADIO_CMD_ERROR;
        sprintf(adio_error_string,"adio_abort_block : Command/Parameter error");
        return(1);
    }

    return(0);
}

int adio_get_pending_int(int boardnum)
{
unsigned port;
int value;

    adio_error_code = ADIO_NO_ERROR;

    if(adio_wait_ready(boardnum))
        return(-1);

    port = adio_base_address[boardnum];

    outportb(port,ADIO_GET_PENDING_INT);

    if(adio_wait_ready(boardnum))
        return(-1);

    if(inportb(port) & 0x80)
    {
        adio_error_code = ADIO_CMD_ERROR;
        sprintf(adio_error_string,"adio_get_pending_int : Command/Parameter error");
        return(-1);
    }

    value = inportb(port + 1);

    return(value & 0xff);   // Make sure return is not negative
}

int adio_clear_pending_ints(int boardnum)
{
unsigned port;
```

```
    adio_error_code = ADIO_NO_ERROR;

    if(adio_wait_ready(boardnum))
        return(1);

    port = adio_base_address[boardnum];

    outportb(port,ADIO_CLR_PENDING_INTS);

    if(adio_wait_ready(boardnum))
        return(1);

    if(inportb(port) & 0x80)
    {
        adio_error_code = ADIO_CMD_ERROR;
        sprintf(adio_error_string,"adio_clear_pending_ints : Command/Parameter error");
        return(1);
    }

    return(0);
}

int adio_set_sample_rate(int boardnum, int sample_rate)
{
unsigned port;

    adio_error_code = ADIO_NO_ERROR;

    if(adio_wait_ready(boardnum))
        return(1);

    if(sample_rate < 0 || sample_rate > 20)
    {
        adio_error_code = ADIO_BAD_SAMPLE_RATE;
        sprintf(adio_error_string,"adio_set_sample_rate : Bad Sample Rate");
        return(1);
    }

    port = adio_base_address[boardnum];

    outportb(port + 1, sample_rate);

    outportb(port, ADIO_SET_SAMPLE_RATE);

    if(adio_wait_ready(boardnum))
        return(1);

    if(inportb(port) & 0x80)
    {
        adio_error_code = ADIO_CMD_ERROR;
        sprintf(adio_error_string,"adio_set_sample_rate : Command/Parameter error");
        return(1);
    }


    return(0);
}

int adio_clr_buff_allocation(int boardnum)
{
unsigned  port;
```

```
    adio_error_code = ADIO_NO_ERROR;

    if(adio_wait_ready(boardnum))
        return(1);

    port = adio_base_address[boardnum];

    outportb(port,ADIO_CLR_BUFF_ALLOCATION);

    if(adio_wait_ready(boardnum))
        return(1);

    if(inportb(port) & 0x80)
    {
        adio_error_code = ADIO_CMD_ERROR;
        sprintf(adio_error_string,"adio_clr_buff_allocation : Command/Parameter error");
        return(1);
    }

    return(0);
}

int adio_set_a2d_buff_allocation(int boardnum, int channel_num, int block_count)
{
unsigned port;

    adio_error_code = ADIO_NO_ERROR;


    if(channel_num < 0 || channel_num > 31)
    {
        adio_error_code = ADIO_BAD_CHANNEL_NUM;
        sprintf(adio_error_string,"adio_set_a2d_buff_allocation : Bad Channel Number");
        return(1);
    }


    if(adio_wait_ready(boardnum))
        return(1);

    port = adio_base_address[boardnum];

    outportb(port+1, channel_num);
    outportb(port+2, block_count);

    outportb(port, ADIO_SET_A2D_BUFF_ALLOCATION);

    if(adio_wait_ready(boardnum))
        return(1);

    if(inportb(port) & 0x80)
    {
        adio_error_code = ADIO_CMD_ERROR;
        sprintf(adio_error_string,"adio_set_a2d_buff_allocation : Command/Parameter error");
        return(1);
    }


    return(0);
}


int adio_set_dac_buff_allocation(int boardnum, int channel_num, int block_count)
```

```
{
unsigned port;

   adio_error_code = ADIO_NO_ERROR;


   if(channel_num < 0 || channel_num > 3)
   {
      adio_error_code = ADIO_BAD_CHANNEL_NUM;
      sprintf(adio_error_string,"adio_set_dac_buff_allocation : Bad Channel Number");
      return(1);
   }


   if(adio_wait_ready(boardnum))
      return(1);

   port = adio_base_address[boardnum];

   outportb(port+1, channel_num);
   outportb(port+2, block_count);

   outportb(port, ADIO_SET_DAC_BUFF_ALLOCATION);

   if(adio_wait_ready(boardnum))
      return(1);

   if(inportb(port) & 0x80)
   {
      adio_error_code = ADIO_CMD_ERROR;
      sprintf(adio_error_string,"adio_set_dac_buff_allocation : Command/Parameter error");
      return(1);
   }

   return(0);
}

int adio_get_free_allocation_blocks(int boardnum)
{
unsigned port;
int value;

   adio_error_code = ADIO_NO_ERROR;

   if(adio_wait_ready(boardnum))
      return(0);

   port = adio_base_address[boardnum];

   outportb(port, ADIO_GET_FREE_BUFF_BLOCKS);

   if(adio_wait_ready(boardnum))
      return(0);

   if(inportb(port) & 0x80)
   {
      adio_error_code = ADIO_CMD_ERROR;
      sprintf(adio_error_string,"adio_get_free_allocation_blocks : Command/Parameter error");
      return(0);
   }

   value = inportb(port + 1);
```

```
    return(value & 0xff);
}

int adio_write_eeprom(int boardnum, int address, unsigned value)
{
unsigned port;

    adio_error_code = ADIO_NO_ERROR;

    if(adio_wait_ready(boardnum))
        return(1);

    if(address < 0 || address > 63)
    {
        adio_error_code = ADIO_BAD_EEPROM_ADDRESS;
        sprintf(adio_error_string,"adio_write_eeprom : Bad EEPROM Address");
        return(1);
    }

    port = adio_base_address[boardnum];

    outportb(port+1, address);
    outportb(port+2, value >> 8);   // MSB of value
    outportb(port+3, value & 0xff); // LSB of value
    outportb(port, ADIO_WRITE_EEPROM_WORD);

    if(adio_wait_ready(boardnum))
        return(1);

    if(inportb(port) & 0x80)
    {
        adio_error_code = ADIO_CMD_ERROR;
        sprintf(adio_error_string,"adio_write_eeprom : Command/Parameter error");
        return(1);
    }

    return(0);

}

unsigned adio_read_eeprom(int boardnum, int address)
{
unsigned port;
unsigned value;

    adio_error_code = ADIO_NO_ERROR;

    if(adio_wait_ready(boardnum))
        return(0);

    if(address < 0 || address > 63)
    {
        adio_error_code = ADIO_BAD_EEPROM_ADDRESS;
        sprintf(adio_error_string,"adio_read_eeprom : Bad EEPROM Address");
        return(0);
    }

    port = adio_base_address[boardnum];

    outportb(port+1, address);
    outportb(port, ADIO_READ_EEPROM_WORD);

    if(adio_wait_ready(boardnum))
```

```
        return(0);

    if(inportb(port) & 0x80)
    {
        adio_error_code = ADIO_CMD_ERROR;
        sprintf(adio_error_string,"adio_read_eeprom : Command/Parameter error");
        return(0);
    }

    value = inportb(port + 2);
    value = value << 8;
    value = value | (inportb(port+3) & 0xff);

    return(value);
}

int adio_set_external_rate(int boardnum, unsigned rate)
{
unsigned port;

    adio_error_code = ADIO_NO_ERROR;

    if(adio_wait_ready(boardnum))
        return(1);


    port = adio_base_address[boardnum];

    outportb(port + 1, rate >> 8);
    outportb(port + 2, rate & 0xff);

    outportb(port, ADIO_SET_EXTERNAL_RATE);

    if(adio_wait_ready(boardnum))
        return(1);

    if(inportb(port) & 0x80)
    {
        adio_error_code = ADIO_CMD_ERROR;
        sprintf(adio_error_string,"adio_set_external_rate : Command/Parameter error");
        return(1);
    }

    return(0);
}

int adio_sys_reset(int boardnum)
{
unsigned port;

    adio_error_code = ADIO_NO_ERROR;

    if(adio_wait_ready(boardnum))
        return(1);


    port = adio_base_address[boardnum];

    outportb(port, ADIO_SYS_RESET);

    return(0);
}
```

```
int adio_get_firmware_version(int boardnum,unsigned *major,unsigned *minor,unsigned *sub)
{
unsigned port;

   adio_error_code = ADIO_NO_ERROR;

   if(adio_wait_ready(boardnum))
      return(0);

   port = adio_base_address[boardnum];

   outportb(port, ADIO_GET_FIRMWARE_REV);

   if(adio_wait_ready(boardnum))
      return(0);

   if(inportb(port) & 0x80)
   {
      adio_error_code = ADIO_CMD_ERROR;
      sprintf(adio_error_string,"adio_get_firmware_version : Command/Parameter error");
      return(0);
   }

   *major = inportb(port+1);
   *minor = inportb(port+2);
   *sub = inportb(port+3);

   return(0);
}

unsigned char adio_read_gpio(int boardnum, int channel)
{
unsigned port;
unsigned char value;

   adio_error_code = ADIO_NO_ERROR;

   if(adio_wait_ready(boardnum))
      return(0);

   if((channel < 0) || (channel > 1))
   {
      adio_error_code = ADIO_BAD_GPIO_CHANNEL_NUM;
      sprintf(adio_error_string,"adio_read_gpio : Bad GPIO Channel Number");
      return(0);
   }

   port = adio_base_address[boardnum];

   if(channel)
      outportb(port, ADIO_READ_GPIO_B);
   else
      outportb(port, ADIO_READ_GPIO_A);

   if(adio_wait_ready(boardnum))
      return(0);

   if(inportb(port) & 0x80)
   {
      adio_error_code = ADIO_CMD_ERROR;
      sprintf(adio_error_string,"adio_read_gpio : Command/Parameter error");
      return(0);
```

```
    }

    value = inportb(port+1);

    return(value);
}

unsigned char adio_write_gpio(int boardnum, int channel, unsigned char value)
{
unsigned port;

    adio_error_code = ADIO_NO_ERROR;

    if(adio_wait_ready(boardnum))
        return(0);

    if((channel < 0) || (channel > 1))
    {
        adio_error_code = ADIO_BAD_GPIO_CHANNEL_NUM;
        sprintf(adio_error_string,"adio_write_gpio : Bad GPIO Channel Number");
        return(0);
    }

    port = adio_base_address[boardnum];

    outportb(port+1,value);

    if(channel)
        outportb(port, ADIO_WRITE_GPIO_B);
    else
        outportb(port, ADIO_WRITE_GPIO_A);

    if(adio_wait_ready(boardnum))
        return(0);

    if(inportb(port) & 0x80)
    {
        adio_error_code = ADIO_CMD_ERROR;
        sprintf(adio_error_string,"adio_write_gpio : Command/Parameter error");
        return(0);
    }

    value = inportb(port+1);

    return(value);
}


int adio_set_gpio_direction(int boardnum, int channel, int direction)
{
unsigned port;

    adio_error_code = ADIO_NO_ERROR;

    if(adio_wait_ready(boardnum))
        return(1);

    if((channel < 0) || (channel > 1))
    {
        adio_error_code = ADIO_BAD_GPIO_CHANNEL_NUM;
        sprintf(adio_error_string,"adio_set_gpio_direction : Bad GPIO Channel Number");
        return(1);
    }
```

```
    if((direction < 0) || (direction > 1))
    {
       adio_error_code = ADIO_BAD_GPIO_DIRECTION;
       sprintf(adio_error_string,"adio_set_gpio_direction : Bad GPIO Direction");
       return(1);

    }

    port = adio_base_address[boardnum];

    outportb(port+1, channel);
    outportb(port+2, direction);
    outportb(port, ADIO_SET_GPIO_DIRECTION);

    if(adio_wait_ready(boardnum))
       return(1);

    if(inportb(port) & 0x80)
    {
       adio_error_code = ADIO_CMD_ERROR;
       sprintf(adio_error_string,"adio_set_gpio_direction : Command/Parameter error");
       return(1);
    }

    return(0);
}

int adio_convert_single(int boardnum, int channel_num)
{
unsigned port;

    adio_error_code = ADIO_NO_ERROR;

    if(adio_wait_ready(boardnum))
       return(1);

    if((channel_num < 0) || (channel_num > 31))
    {
       adio_error_code = ADIO_BAD_A2D_CHANNEL_NUM;
       sprintf(adio_error_string,"adio_convert_single : Bad A2D Channel Number");
       return(1);
    }

    port = adio_base_address[boardnum];

    outportb(port+1, channel_num);
    outportb(port, ADIO_CONVERT_SINGLE);

    if(adio_wait_ready(boardnum))
       return(1);

    if(inportb(port) & 0x80)
    {
       adio_error_code = ADIO_CMD_ERROR;
       sprintf(adio_error_string,"adio_convert_single : Command/Parameter error");
       return(1);
    }

    return(0);
}
```

```
int adio_convert_single_int(int boardnum, int channel_num)
{
unsigned port;

   adio_error_code = ADIO_NO_ERROR;

   if(adio_wait_ready(boardnum))
      return(1);

   if((channel_num < 0) || (channel_num > 31))
   {
      adio_error_code = ADIO_BAD_A2D_CHANNEL_NUM;
      sprintf(adio_error_string,”adio_convert_single : Bad A2D Channel Number”);
      return(1);
   }

   port = adio_base_address[boardnum];

   outportb(port+1, channel_num);
   outportb(port, ADIO_CONVERT_SINGLE_INT);

   if(adio_wait_ready(boardnum))
      return(1);

   if(inportb(port) & 0x80)
   {
      adio_error_code = ADIO_CMD_ERROR;
      sprintf(adio_error_string,”adio_convert_single_int : Command/Parameter error”);
      return(1);
   }

   return(0);
}

int adio_start_block_convert(int boardnum, int channel_num, unsigned rate, int block_count)
{
unsigned port;

   adio_error_code = ADIO_NO_ERROR;

   if(adio_wait_ready(boardnum))
      return(1);

   if((channel_num < 0) || (channel_num > 31))
   {
      adio_error_code = ADIO_BAD_A2D_CHANNEL_NUM;
      sprintf(adio_error_string,”adio_start_block_convert : Bad A2D Channel Number”);
      return(1);
   }


   port = adio_base_address[boardnum];

   outportb(port+1, channel_num);
   outportb(port+2, rate >> 8);
   outportb(port+3, rate & 0xff);
   outportb(port+4, block_count);

   outportb(port, ADIO_CONVERT_BLOCK);

   if(adio_wait_ready(boardnum))
      return(1);
```

```
    if(inportb(port) & 0x80)
    {
       adio_error_code = ADIO_CMD_ERROR;
       sprintf(adio_error_string,"adio_start_block_convert : Command/Parameter error");
       return(1);
    }

    return(0);
}

int adio_start_block_convert_int(int boardnum, int channel_num, unsigned rate, int block_count)
{
unsigned port;

    adio_error_code = ADIO_NO_ERROR;

    if(adio_wait_ready(boardnum))
       return(1);

    if((channel_num < 0) || (channel_num > 31))
    {
       adio_error_code = ADIO_BAD_A2D_CHANNEL_NUM;
       sprintf(adio_error_string,"adio_start_block_convert : Bad A2D Channel Number");
       return(1);
    }


    port = adio_base_address[boardnum];

    outportb(port+1, channel_num);
    outportb(port+2, rate >> 8);
    outportb(port+3, rate & 0xff);
    outportb(port+4, block_count);

    outportb(port, ADIO_CONVERT_BLOCK_INT);

    if(adio_wait_ready(boardnum))
       return(1);

    if(inportb(port) & 0x80)
    {
       adio_error_code = ADIO_CMD_ERROR;
       sprintf(adio_error_string,"adio_start_block_convert_int : Command/Parameter error");
       return(1);
    }

    return(0);
}

int adio_read_block_data(int boardnum, int channel_num,int block_count,char *buff)
{
unsigned port;
int x;

    adio_error_code = ADIO_NO_ERROR;

    if(adio_wait_ready(boardnum))
       return(1);

    if((channel_num < 0) || (channel_num > 31))
    {
       adio_error_code = ADIO_BAD_A2D_CHANNEL_NUM;
       sprintf(adio_error_string,"adio_read_block_data : Bad A2D Channel Number");
```

```
		return(1);
	}

	port = adio_base_address[boardnum];

	outportb(port + 1, channel_num);
	outportb(port + 2, block_count);
	outportb(port, ADIO_READ_BLOCK_DATA);

	if(adio_wait_ready(boardnum))
		return(1);

	if(inportb(port) & 0x80)
	{
		adio_error_code = ADIO_CMD_ERROR;
		sprintf(adio_error_string,"adio_read_block_data : Command/Parameter error");
		return(1);
	}

	for(x=0; x < (block_count * 256); x++)
		*buff++ = inportb(port+7);

	return(0);
}

int adio_set_channel_mode(int boardnum,int channel_num, int mode)
{
unsigned port;

	adio_error_code = ADIO_NO_ERROR;

	if(adio_wait_ready(boardnum))
		return(1);

	if((channel_num < 0 ) || (channel_num > 31))
	{
		adio_error_code = ADIO_BAD_A2D_CHANNEL_NUM;
		sprintf(adio_error_string,"adio_set_channel_mode : Bad A2D Channel Number");
		return(1);
	}


	if((mode < 0 ) || (mode > 31))
	{
		adio_error_code = ADIO_BAD_A2D_MODE;
		sprintf(adio_error_string,"adio_set_channel_mode : Bad A2D Mode Number");
		return(1);
	}


	port = adio_base_address[boardnum];

	outportb(port+1, channel_num);
	outportb(port+2, mode);
	outportb(port, ADIO_SET_CHANNEL_MODE);

	if(adio_wait_ready(boardnum))
		return(1);

	if(inportb(port) & 0x80)
	{
		adio_error_code = ADIO_CMD_ERROR;
		sprintf(adio_error_string,"adio_set_channel_mode : Command/Parameter error");
```

```
        return(1);
    }

    return(0);
}

unsigned char adio_read_channel_mode(int boardnum, int channel_num)
{
unsigned port;
unsigned char value;

    adio_error_code = ADIO_NO_ERROR;

    if(adio_wait_ready(boardnum))
        return(0);

    if((channel_num < 0 ) || (channel_num > 31))
    {
        adio_error_code = ADIO_BAD_A2D_CHANNEL_NUM;
        sprintf(adio_error_string,"adio_read_channel_mode : Bad A2D Channel Number");
        return(0);
    }

    port = adio_base_address[boardnum];

    outportb(port+1,channel_num);
    outportb(port,ADIO_READ_CHANNEL_MODE);

    if(adio_wait_ready(boardnum))
        return(0);

    if(inportb(port) & 0x80)
    {
        adio_error_code = ADIO_CMD_ERROR;
        sprintf(adio_error_string,"adio_read_channel_mode : Command/Parameter error");
        return(0);
    }

    value = inportb(port+2) & 0xff;

    return(value);
}

int adio_stop_block_conversion(int boardnum, int channel_num)
{
unsigned port;

    adio_error_code = ADIO_NO_ERROR;

    if(adio_wait_ready(boardnum))
        return(1);

    if((channel_num < 0 ) || (channel_num > 31))
    {
        adio_error_code = ADIO_BAD_A2D_CHANNEL_NUM;
        sprintf(adio_error_string,"adio_stop_block_conversion : Bad A2D Channel Number");
        return(1);
    }

    port = adio_base_address[boardnum];

    outportb(port+1, channel_num);
    outportb(port, ADIO_STOP_BLOCK_CONVERT);
```

```
    if(adio_wait_ready(boardnum))
        return(1);

    if(inportb(port) & 0x80)
    {
        adio_error_code = ADIO_CMD_ERROR;
        sprintf(adio_error_string,"adio_stop_block_conversion : Command/Parameter error");
        return(1);
    }

     return(0);
}

int adio_save_channel_modes(int boardnum)
{
unsigned port;

    adio_error_code = ADIO_NO_ERROR;

    if(adio_wait_ready(boardnum))
        return(1);

    port = adio_base_address[boardnum];

    outportb(port, ADIO_SAVE_CHANNEL_MODES);

    if(adio_wait_ready(boardnum))
        return(1);

    if(inportb(port) & 0x80)
    {
        adio_error_code = ADIO_CMD_ERROR;
        sprintf(adio_error_string,"adio_save_channel_modes : Command/Parameter error");
        return(1);
    }

    return(0);
}

unsigned adio_read_conversion_data(int boardnum, int channel_num)
{
unsigned port;
unsigned value;

    adio_error_code = ADIO_NO_ERROR;

    if(adio_wait_ready(boardnum))
        return(0);

    if((channel_num < 0 ) || (channel_num > 31))
    {
        adio_error_code = ADIO_BAD_A2D_CHANNEL_NUM;
        sprintf(adio_error_string,"adio_read_conversion_data : Bad A2D Channel Number");
        return(0);
    }

    port = adio_base_address[boardnum];

    outportb(port+1, channel_num);
    outportb(port, ADIO_READ_CONVERSION_DATA);

    if(adio_wait_ready(boardnum))
```

```
        return(0);

    if(inportb(port) & 0x80)
    {
        adio_error_code = ADIO_CMD_ERROR;
        sprintf(adio_error_string,"adio_read_conversion_data : Command/Parameter error");
        return(1);
    }

    value = inportb(port+2) << 8;
    value = value | (inportb(port+3) & 0xff);

    return(value);
}


int adio_fast_convert(int boardnum, int channel_num)
{
unsigned port;

    adio_error_code = ADIO_NO_ERROR;

    if(adio_wait_ready(boardnum))
        return(1);

    if((channel_num < 0) || (channel_num > 31))
    {
        adio_error_code = ADIO_BAD_A2D_CHANNEL_NUM;
        sprintf(adio_error_string,"adio_fast_convert : Bad A2D Channel Number");
        return(1);
    }

    port = adio_base_address[boardnum];

    outportb(port+1, channel_num);
    outportb(port, ADIO_FAST_CONVERT);

    if(adio_wait_ready(boardnum))
        return(1);

    if(inportb(port) & 0x80)
    {
        adio_error_code = ADIO_CMD_ERROR;
        sprintf(adio_error_string,"adio_fast_convert : Command/Parameter error");
        return(1);
    }

    return(0);
}

int adio_fast_convert_int(int boardnum, int channel_num)
{
unsigned port;

    adio_error_code = ADIO_NO_ERROR;

    if(adio_wait_ready(boardnum))
        return(1);

    if((channel_num < 0) || (channel_num > 31))
    {
        adio_error_code = ADIO_BAD_A2D_CHANNEL_NUM;
        sprintf(adio_error_string,"adio_fast_convert_int : Bad A2D Channel Number");
```

```
      return(1);
   }

   port = adio_base_address[boardnum];

   outportb(port+1, channel_num);
   outportb(port, ADIO_FAST_CONVERT_INT);

   if(adio_wait_ready(boardnum))
      return(1);

   if(inportb(port) & 0x80)
   {
      adio_error_code = ADIO_CMD_ERROR;
      sprintf(adio_error_string,"adio_fast_convert_int : Command/Parameter error");
      return(1);
   }

   return(0);
}

int adio_stop_block_conversions(int boardnum)
{
unsigned port;

   adio_error_code = ADIO_NO_ERROR;

   if(adio_wait_ready(boardnum))
      return(1);

   port = adio_base_address[boardnum];

   outportb(port,ADIO_STOP_ALL_BLOCK_CONVERSIONS);

   if(adio_wait_ready(boardnum))
      return(1);

   if(inportb(port) & 0x80)
   {
      adio_error_code = ADIO_CMD_ERROR;
      sprintf(adio_error_string,"adio_stop_block_conversions : Command/Parameter error");
      return(1);
   }

   return(0);
}

int adio_get_mode_cal_values(int boardnum,int mode_val, unsigned *trim_val, unsigned *gain_val)
{
unsigned port;
unsigned value;

   adio_error_code = ADIO_NO_ERROR;

   if(adio_wait_ready(boardnum))
      return(1);

   if((mode_val < 0) || (mode_val > 31))
   {
      adio_error_code = ADIO_BAD_A2D_MODE;
      sprintf(adio_error_string,"adio_get_mode_cal_value : Bad A2D Mode Number");
      return(1);
   }
```

```
    port = adio_base_address[boardnum];

    outportb(port+1,mode_val);
    outportb(port,ADIO_GET_MODE_CAL_VALUES);

    if(adio_wait_ready(boardnum))
        return(1);

    if(inportb(port) & 0x80)
    {
        adio_error_code = ADIO_CMD_ERROR;
        sprintf(adio_error_string,"adio_get_mode_val_value : Command/Parameter error");
        return(1);
    }

    value = inportb(port+2);
    value = (value << 8) | inportb(port+3);
    *trim_val = value;

    value = inportb(port+4);
    value = (value << 8) | inportb(port+5);
    *gain_val = value;

    return(0);
}

int adio_set_mode_cal_values(int boardnum, int mode_val, unsigned trim_val, unsigned gain_val)
{
unsigned port;
unsigned value;

    adio_error_code = ADIO_NO_ERROR;

    if(adio_wait_ready(boardnum))
        return(1);

    if((mode_val < 0) || (mode_val > 31))
    {
        adio_error_code = ADIO_BAD_A2D_MODE;
        sprintf(adio_error_string,"adio_set_mode_cal_value : Bad A2D Mode Number");
        return(1);
    }

    port = adio_base_address[boardnum];

    outportb(port+1,mode_val);
    outportb(port+2,trim_val >> 8);
    outportb(port+3,trim_val & 0x0ff);
    outportb(port+4,gain_val >> 8);
    outportb(port+5,gain_val & 0xff);

    outportb(port,ADIO_SET_MODE_CAL_VALUES);

    if(adio_wait_ready(boardnum))
        return(1);

    if(inportb(port) & 0x80)
    {
        adio_error_code = ADIO_CMD_ERROR;
        sprintf(adio_error_string,"adio_set_mode_cal_value : Command/Parameter error");
        return(1);
    }
```

```
   return(0);
}

int adio_save_cal_values(int boardnum)
{
unsigned port;

   adio_error_code = ADIO_NO_ERROR;

   if(adio_wait_ready(boardnum))
      return(1);

   port = adio_base_address[boardnum];

   outportb(port,ADIO_SAVE_CAL_VALUES);

   if(adio_wait_ready(boardnum))
      return(1);

   if(inportb(port) & 0x80)
   {
      adio_error_code = ADIO_CMD_ERROR;
      sprintf(adio_error_string,"adio_save_cal_values : Command/Parameter error");
      return(1);
   }

   return(0);
}

int adio_direct_dac_output(int boardnum,int dac_channel, int dac_mode, unsigned dac_data)
{
unsigned port;

   adio_error_code = ADIO_NO_ERROR;

   if(adio_wait_ready(boardnum))
      return(1);

   if((dac_channel < 0) || (dac_channel > 3))
   {
      adio_error_code = ADIO_BAD_DAC_CHANNEL;
      sprintf(adio_error_string,"adio_direct_dac_output : Bad DAC Channel Number");
      return(1);
   }


   if((dac_mode < 8) || (dac_mode > 13))
   {
      adio_error_code = ADIO_BAD_DAC_MODE;
      sprintf(adio_error_string,"adio_direct_dac_output : Bad DAC Mode Number");
      return(1);
   }


   port = adio_base_address[boardnum];

   outportb(port + 1, dac_channel);
   outportb(port + 2, dac_mode);
   outportb(port + 3, dac_data >> 8);
   outportb(port + 4, dac_data & 0xff);

   outportb(port, ADIO_DIRECT_DAC_OUTPUT);
```

```
    if(adio_wait_ready(boardnum))
        return(1);

    if(inportb(port) & 0x80)
    {
        adio_error_code = ADIO_CMD_ERROR;
        sprintf(adio_error_string,"adio_direct_dac_output : Command/Parameter error");
        return(1);
    }

    return(0);
}

int adio_set_dac_mode(int boardnum, int dac_channel, int dac_mode)
{
unsigned port;

    adio_error_code = ADIO_NO_ERROR;

    if(adio_wait_ready(boardnum))
        return(1);

    if((dac_channel < 0) || (dac_channel > 3))
    {
        adio_error_code = ADIO_BAD_DAC_CHANNEL;
        sprintf(adio_error_string,"adio_set_dac_mode : Bad DAC Channel Number");
        return(1);
    }


    if((dac_mode < 8) || (dac_mode > 13))
    {
        adio_error_code = ADIO_BAD_DAC_MODE;
        sprintf(adio_error_string,"adio_set_dac_mode : Bad DAC Mode Number");
        return(1);
    }

    port = adio_base_address[boardnum];

    outportb(port + 1, dac_channel);
    outportb(port + 2, dac_mode);

    outportb(port, ADIO_SET_DAC_MODE);

    if(adio_wait_ready(boardnum))
        return(1);

    if(inportb(port) & 0x80)
    {
        adio_error_code = ADIO_CMD_ERROR;
        sprintf(adio_error_string,"adio_set_dac_mode : Command/Parameter error");
        return(1);
    }

    return(0);
}

int adio_start_dac_block(int boardnum, int dac_channel, unsigned dac_rate, unsigned block_count)
{
unsigned port;

    adio_error_code = ADIO_NO_ERROR;
```

```
    if(adio_wait_ready(boardnum))
        return(1);

    if((dac_channel < 0) || (dac_channel > 3))
    {
        adio_error_code = ADIO_BAD_DAC_CHANNEL;
        sprintf(adio_error_string,"adio_start_dac_block : Bad DAC Channel Number");
        return(1);
    }

    port = adio_base_address[boardnum];

    outportb(port + 1, dac_channel);
    outportb(port + 2, dac_rate >> 8);
    outportb(port + 3, dac_rate & 0xff);
    outportb(port + 4, block_count & 0xff);

    outportb(port, ADIO_START_DAC_BLOCK);

    if(adio_wait_ready(boardnum))
        return(1);

    if(inportb(port) & 0x80)
    {
        adio_error_code = ADIO_CMD_ERROR;
        sprintf(adio_error_string,"adio_start_dac_block : Command/Parameter error");
        return(1);
    }

    return(0);
}


int adio_start_dac_block_int(int boardnum, int dac_channel, unsigned dac_rate, unsigned block_count)
{
unsigned port;

    adio_error_code = ADIO_NO_ERROR;

    if(adio_wait_ready(boardnum))
        return(1);

    if((dac_channel < 0) || (dac_channel > 3))
    {
        adio_error_code = ADIO_BAD_DAC_CHANNEL;
        sprintf(adio_error_string,"adio_start_dac_block_int : Bad DAC Channel Number");
        return(1);
    }

    port = adio_base_address[boardnum];

    outportb(port + 1, dac_channel);
    outportb(port + 2, dac_rate >> 8);
    outportb(port + 3, dac_rate & 0xff);
    outportb(port + 4, block_count & 0xff);

    outportb(port, ADIO_START_DAC_BLOCK_INT);

    if(adio_wait_ready(boardnum))
        return(1);

    if(inportb(port) & 0x80)
```

```
    {
        adio_error_code = ADIO_CMD_ERROR;
        sprintf(adio_error_string,"adio_start_dac_block_int : Command/Parameter error");
        return(1);
    }

    return(0);
}


int adio_load_dac_data(int boardnum, int dac_channel, unsigned block_count)
{
unsigned port;

    adio_error_code = ADIO_NO_ERROR;

    if(adio_wait_ready(boardnum))
        return(1);

    if((dac_channel < 0) || (dac_channel > 3))
    {
        adio_error_code = ADIO_BAD_DAC_CHANNEL;
        sprintf(adio_error_string,"adio_load_dac_data : Bad DAC Channel Number");
        return(1);
    }

    port = adio_base_address[boardnum];

    outportb(port + 1, dac_channel);
    outportb(port + 2, block_count & 0xff);

    outportb(port , ADIO_LOAD_DAC_DATA);

    if(adio_wait_ready(boardnum))
        return(1);

    if(inportb(port) & 0x80)
    {
        adio_error_code = ADIO_CMD_ERROR;
        sprintf(adio_error_string,"adio_load_dac_data : Command/Parameter error");
        return(1);
    }

    return(0);
}

int adio_get_dac_mode(int boardnum, int dac_channel)
{
unsigned port;

    adio_error_code = ADIO_NO_ERROR;

    if(adio_wait_ready(boardnum))
        return(1);

    if((dac_channel < 0) || (dac_channel > 3))
    {
        adio_error_code = ADIO_BAD_DAC_CHANNEL;
        sprintf(adio_error_string,"adio_get_dac_mode : Bad DAC Channel Number");
        return(1);
    }

    port = adio_base_address[boardnum];
```

```
    outportb(port + 1, dac_channel);
    outportb(port, ADIO_READ_DAC_MODE);

    if(adio_wait_ready(boardnum))
        return(-1);

    if(inportb(port) & 0x80)
    {
        adio_error_code = ADIO_CMD_ERROR;
        sprintf(adio_error_string,"adio_get_dac_mode : Command/Parameter error");
        return(-1);
    }

    return(inportb(port+2) & 0xff);

}


int adio_clr_dac_output(int boardnum, int channel_mask)
{
unsigned port;

    adio_error_code = ADIO_NO_ERROR;

    if(adio_wait_ready(boardnum))
        return(1);

    if((channel_mask < 0) || (channel_mask > 15))
    {
        adio_error_code = ADIO_BAD_DAC_MASK;
        sprintf(adio_error_string,"adio_clr_dac_output : Bad DAC Channel Mask");
        return(1);
    }

    port = adio_base_address[boardnum];

    outportb(port + 1, channel_mask);
    outportb(port, ADIO_CLR_DAC_OUTPUT);

    if(adio_wait_ready(boardnum))
        return(1);

    if(inportb(port) & 0x80)
    {
        adio_error_code = ADIO_CMD_ERROR;
        sprintf(adio_error_string,"adio_clr_dac_output : Command/Parameter error");
        return(1);
    }

    return(0);
}


int adio_enable_ext_dac_clr(int boardnum)
{
unsigned port;

    adio_error_code = ADIO_NO_ERROR;

    if(adio_wait_ready(boardnum))
        return(1);
```

```
    port = adio_base_address[boardnum];

    outportb(port, ADIO_ENAB_EXT_DAC_CLR);

    if(adio_wait_ready(boardnum))
        return(1);

    if(inportb(port) & 0x80)
    {
        adio_error_code = ADIO_CMD_ERROR;
        sprintf(adio_error_string,"adio_enab_ext_dac_clr : Command/Parameter error");
        return(1);
    }

    return(0);
}


int adio_disable_ext_dac_clr(int boardnum)
{
unsigned port;

    adio_error_code = ADIO_NO_ERROR;

    if(adio_wait_ready(boardnum))
        return(1);


    port = adio_base_address[boardnum];

    outportb(port, ADIO_DISAB_EXT_DAC_CLR);

    if(adio_wait_ready(boardnum))
        return(1);

    if(inportb(port) & 0x80)
    {
        adio_error_code = ADIO_CMD_ERROR;
        sprintf(adio_error_string,"adio_disab_ext_dac_clr : Command/Parameter error");
        return(1);
    }

    return(0);
}



unsigned adio_read_single_data(int boardnum)
{
unsigned port;
unsigned value;

    adio_error_code = ADIO_NO_ERROR;

    if(adio_wait_ready(boardnum))
        return(1);


    port = adio_base_address[boardnum];

    value = inportb(port + 2);
    value = (value << 8) | inportb(port + 3);
```

```
   return(value);
}


int adio_start_sequence(int boardnum, int start_channel, int end_channel)
{
unsigned port;

        adio_error_code = ADIO_NO_ERROR;

        if((start_channel < 0) || (start_channel > 31) || (end_channel < 0) || (end_channel > 31) ||
(start_channel > end_channel))
        {
        adio_error_code = ADIO_BAD_A2D_CHANNEL_NUM;
        sprintf(adio_error_string,"Adio start sequence : Bad A2D Channel Number");
        return(1);

        }

        if(adio_wait_ready(boardnum))
        return(1);

        port = adio_base_address[boardnum];

        outportb(port + 1, start_channel);
        outportb(port + 2, end_channel);

        outportb(port,ADIO_SEQUENCE_START);

    if(adio_wait_ready(boardnum))
        return(1);

        if(inportb(port) & 0x80)
        {
        adio_error_code = ADIO_CMD_ERROR;
        sprintf(adio_error_string,"Adio start sequence : Command/Parameter error");
        return(1);
        }

        return(0);
}

int adio_read_sequence(int boardnum,int count,unsigned *value)
{
unsigned char *buff;
unsigned port;
int x;

        buff = (unsigned char *)value;

        if(adio_wait_ready(boardnum))
        return(1);

        port = adio_base_address[boardnum];

        outportb(port, ADIO_READ_SEQUENCE);

        if(adio_wait_ready(boardnum))
        return(1);

        if(adio_wait_drq(boardnum))
                return(1);
```

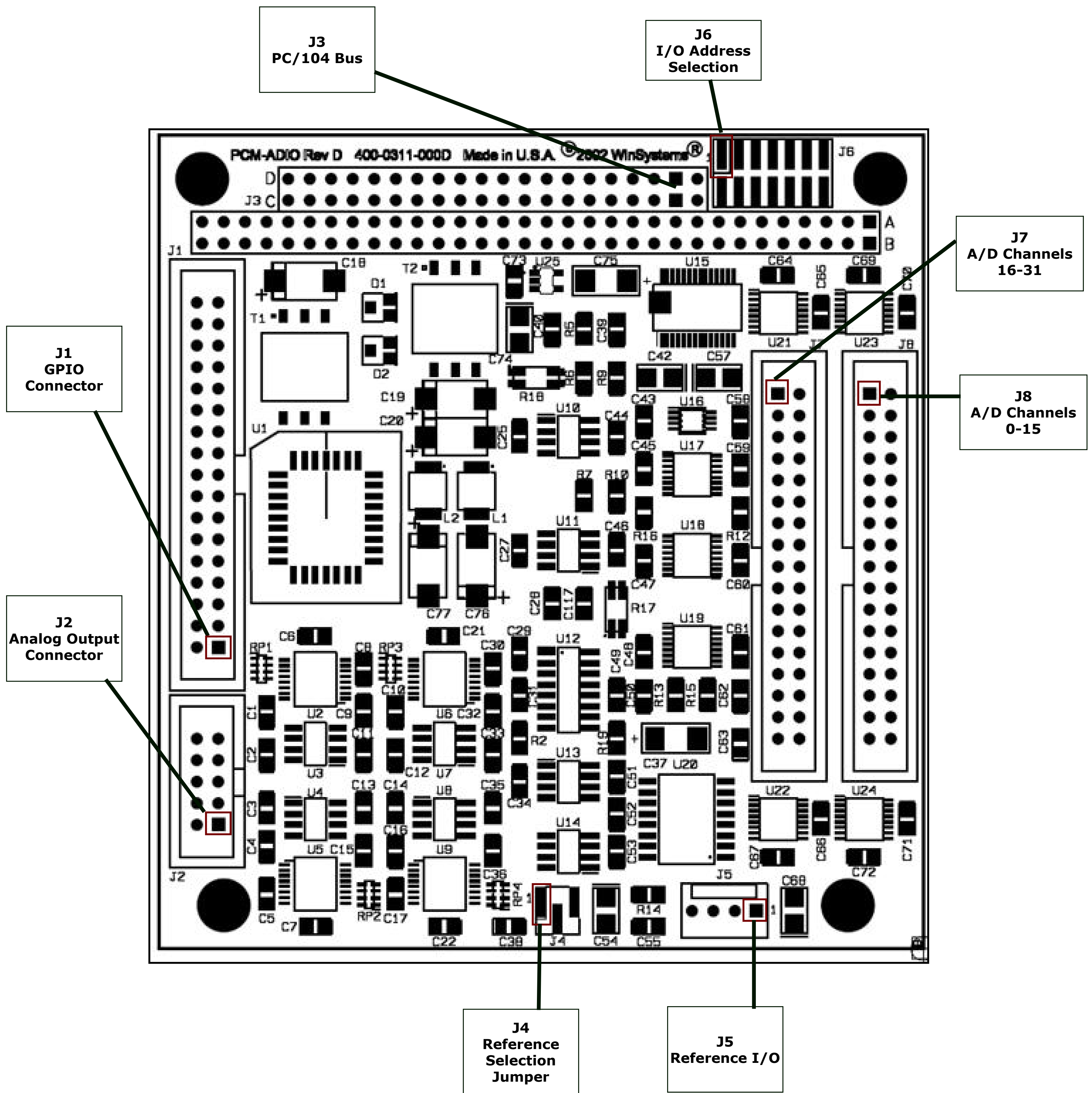
```
	for(x=0; x<count; x++)
		*buff++ = inportb(port+7);

	return 0;

}
```

## Software Drivers & Examples

| **Documentation** | |
|---|---|
| (Reference) | |
| Application Note(AN-22): Understanding Conversion Rate in Industrial Multiplexed Data Acquisition Systems | AN-22Speed.pdf |
| Application Note(AN-23): Understanding Analog to Digital Converter Transition Noise | AN-23Noise.pdf |
| Application Note(AN-24): Simple Resolution Enhancement for the PCM-ADIO Advanced Analog I/O | AN-24Resol.pdf |
| Application Note(AN-25): Understanding Auto-Calibration and Winsystems' Advanced Analog I/O | AN-25Auto.pdf |
| | |
| **Examples** | |
| (I/O Drivers) | |
| Windows 2000/XP Drivers | winadio.zip |
| Linux Kernel 2.2 | linux_pcmadio.zip |
| Linux Kernel 2.6 | linuxadio_2.6.zip |
| DOS Drivers and Utilities | dos_pcmadio.zip |
| | |

# Jumper Reference

***Drawings ONLY*** - for more detailed information on these parts, refer to the descriptions shown previously in this manual.




**NOTE:** The reference line to each component part has been drawn to Pin 1, where applicable. Pin 1 is also highlighted with a red square, where applicable.

# Specifications

**Electrical**

| | |
|---|---|
| **Bus Interface** | :PC/104 16-Bit |
| **VCC** | :+5V ±5% @ 1000 mA typical |
| **I/O Addressing** | :10-Bit user jumperable address. Each board uses 8 consecutive I/O addresses. |
| **A2D Converter** | :16-Bit resolution<br>15-bit no missing codes<br>±3 LSB INL maximum<br>±3 LSB DNL maximum<br>0.9 LSB typical transition noise. |
| **D2A Converter** | :Four (4) independently software configurable output channels<br>12-Bit resolution, no missing codes<br>±1 LSB INL maximum<br>±1 LSB DNL maximum, Monotonic<br>±0.5 LSB Gain error typical<br>±1 LSB Bipolar zero error maximum<br>3 PPM/degree C Gain temperature Coefficient<br>2 µS settling time to .1% full scale step<br>Output Current:<br>±10 mA per output<br>±30 mA absolute max all outputs combined |
| **Digital I/O** | :16 General purpose TTL digital I/O pins.<br>Byte programmable for input or output<br>:Sink and source : 12 mA each pin |
| **Reference** | :2ppm/±C typical temperature drift<br>:10mA buffered reference out |

**Mechanical**

| | |
|---|---|
| **Dimensions** | :3.6” X 3.8” (90 mm x 96 mm) |
| **PC Board** | :FR-4 Epoxy glass with 4 signal layers 2 power planes, screened component legend, and plated through holes. |
| **Jumpers** | :0.025” square posts on 0.10” centers |
| **Connectors** | :Analog Input (34 Pin 0.10” grid RN type IDH-34-LP)<br>:Analog Output (10 Pin 0.10” grid RN type IDH-10-LP)<br>:Digital I/O (34 Pin 0.10” grid RN type IDH-34-LP) |

**Environmental**

| | |
|---|---|
| **Operating Temperature** | :-40°C to +85° C |
| **Noncondensing humidity** | :5% to 95% |

# WARRANTY REPAIR INFORMATION

**WARRANTY**

WinSystems warrants to Customer that for a period of two (2) years from the date of shipment any Products and Software purchased or licensed hereunder which have been developed or manufactured by WinSystems shall be free of any material defects and shall perform substantially in accordance with WinSystems' specifications therefore. With respect to any Products or Software purchased or licensed hereunder which have been developed or manufactured by others, WinSystems shall transfer and assign to Customer any warranty of such manufacturer or developer held by WinSystems, provided that the warranty, if any, may be assigned. Notwithstanding anything herein to the contrary, this warranty granted by WinSystems to the Customer shall be for the sole benefit of the Customer, and may not be assigned, transferred or conveyed to any third party. The sole obligation of WinSystems for any breach of warranty contained herein shall be, at its option, either (i) to repair or replace at its expense any materially defective Products or Software, or (ii) to take back such Products and Software and refund the Customer the purchase price and any license fees paid for the same. Customer shall pay all freight, duty, broker's fees, insurance charges for the return of any Products or Software to WinSystems under this warranty. WinSystems shall pay freight and insurance charges for any repaired or replaced Products or Software thereafter delivered to Customer within the United States. All fees and costs for shipment outside of the United States shall be paid by Customer. The foregoing warranty shall not apply to any Products of Software which have been subject to abuse, misuse, vandalism, accidents, alteration, neglect, unauthorized repair or improper installations.

THERE ARE NO WARRANTIES BY WINSYSTEMS EXCEPT AS STATED HEREIN, THERE ARE NO OTHER WARRANTIES EXPRESS OR IMPLIED INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, IN NO EVENT SHALL WINSYSTEMS BE LIABLE FOR CONSEQUENTIAL, INCIDENTIAL OR SPECIAL DAMAGES INCLUDING, BUT NOT LIMITED TO, DAMAGES FOR LOSS OF DATA, PROFITS OR GOODWILL. WINSYSTEMS' MAXIMUM LIABILITY FOR ANY BREACH OF THIS AGREEMENT OR OTHER CLAIM RELATED TO ANY PRODUCTS, SOFTWARE, OR THE SUBJECT MATTER HEREOF, SHALL NOT EXCEED THE PURCHASE PRICE OR LICENSE FEE PAID BY CUSTOMER TO WINSYSTEMS FOR THE PRODUCTS OR SOFTWARE OR PORTION THEREOF TO WHICH SUCH BREACH OR CLAIM PERTAINS.

**WARRANTY SERVICE**

1. To obtain service under this warranty, obtain a return authorization number. In the United States, contact the WinSystems' Service Center for a return authorization number. Outside the United States, contact your local sales agent for a return authorization number.

2. You must send the product postage prepaid and insured. You must enclose the products in an anti-static bag to protect from damage by static electricity. WinSystems is not responsible for damage to the product due to static electricity.