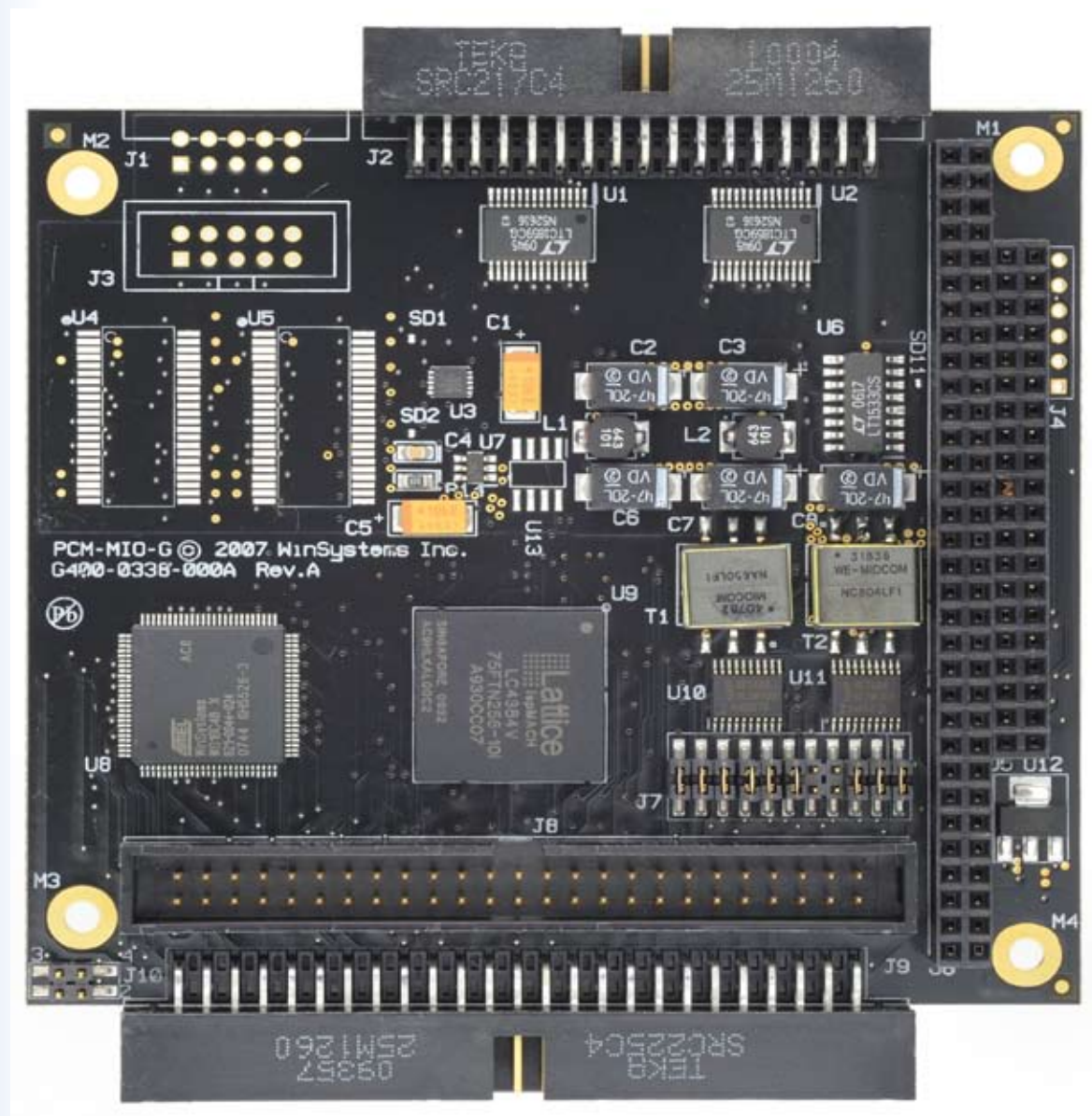


PCM-MIO-G-DA-1

8 Channel, 12-bit Digital Input and 48-bit Digital I/O

PRODUCT MANUAL



WinSystems, Inc.
715 Stadium Drive
Arlington, TX 76011

<http://www.winsystems.com>

REVISION HISTORY

P/N 400-0338-000

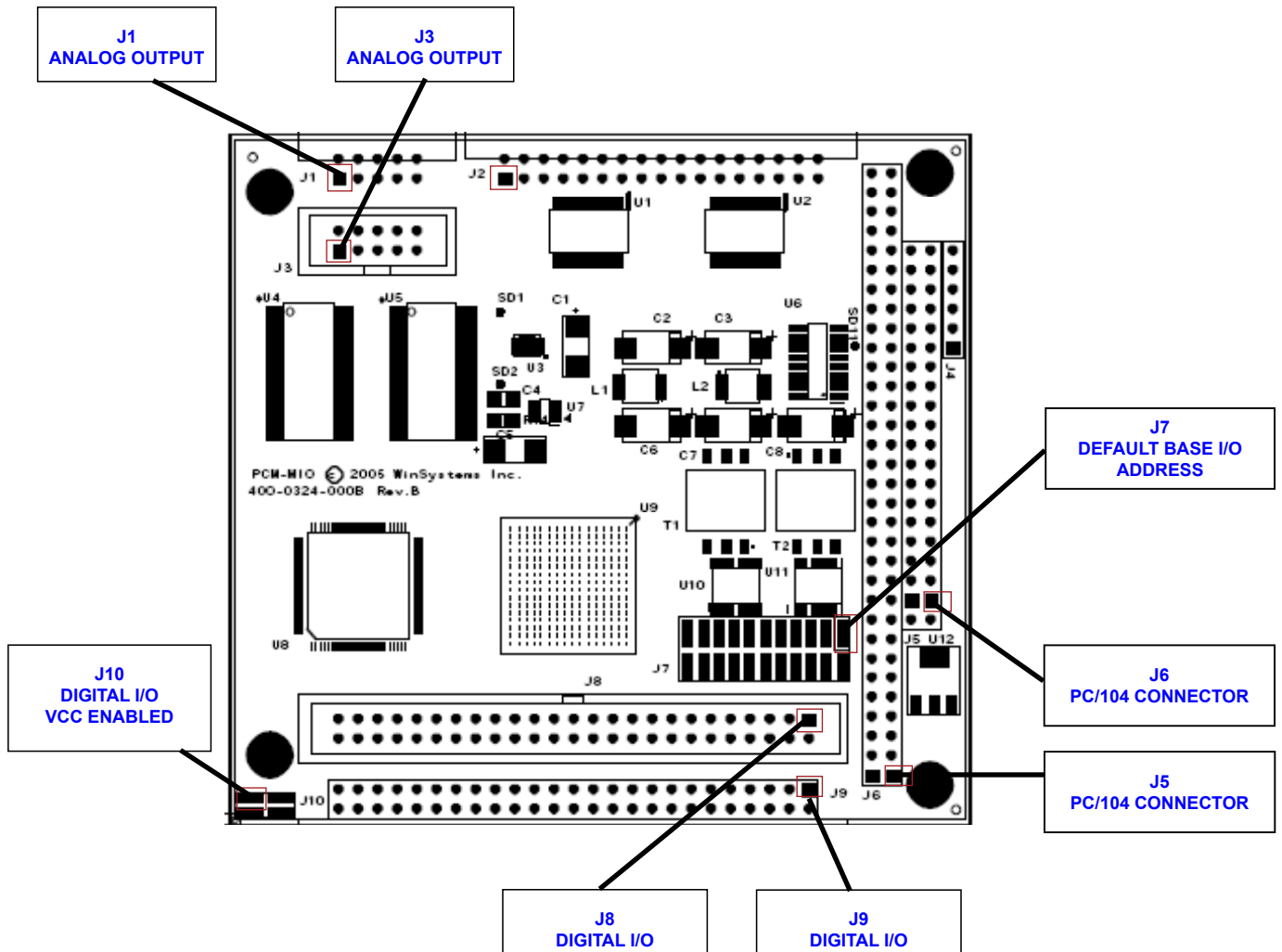
Date Code	REV Level	ECO Number
100609	Preliminary	

TABLE OF CONTENTS

Visual Index - Top View (Connectors)	5
JUMPER REFERENCE	6
BASE I/O SELECTION	6
J7 - Default Base I/O Address	6
DIGITAL I/O	6
J10 - VCC Enable	6
INTRODUCTION	7
FEATURES	7
FUNCTIONALITY	8
Digital-to-Analog	8
Parallel I/O	8
Interrupts	8
DMA Support	8
CONNECTOR REFERENCE	9
I/O ADDRESS	9
J7 - Default Base I/O Address	9
Analog Output (D/A)	10
J2 - D/A Analog Output Connections	10
DIGITAL I/O INTERFACE	11
J8, J9 - Digital I/O	11
J10 - VCC Enable	11
PC/104 BUS INTERFACE	12
J5, J6 - PC/104	12
SOFTWARE SUMMARY	13
REGISTERS	13
Register Definitions (WS16C48)	13
Register Details	13
D/A Converters	16
WS16C48 PROGRAMMING REFERENCE	21
Function Definitions	21
Sample Programs	24
Summary	24
CABLES	25
SOFTWARE DRIVERS & EXAMPLES	26
MECHANICAL DRAWING	27
APPENDIX	28
C Source Code Listings	29
WARRANTY INFORMATION	45

This page has been left intentionally blank.

Visual Index - Top View (Connectors)



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

JUMPER REFERENCE

NOTE: Jumper Part numbers W/S# G201-0002-005 and SAMTEC 2SN-BK-G are applicable to all jumpers. These are available in a ten piece kit from WinSystems (Part# KIT-JMP-G-200).

BASE I/O SELECTION

J7 - Default Base I/O Address

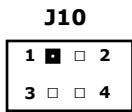


J7

A5	1 <input checked="" type="checkbox"/> 2
A6	3 <input type="checkbox"/> 4
A7	5 <input type="checkbox"/> 6
A8	7 <input type="checkbox"/> 8
A9	9 <input type="checkbox"/> 10
A10	11 <input type="checkbox"/> 12
A11	13 <input type="checkbox"/> 14
A12	15 <input type="checkbox"/> 16
A13	17 <input type="checkbox"/> 18
A14	19 <input type="checkbox"/> 20
A15	21 <input type="checkbox"/> 22

DIGITAL I/O

J10 - VCC Enable



+5V is provided at pin 49 of J9	3-4
+5V is provided at pin 49 of J8	1-2
No Power at Pin 49 of J8/J9 (default)	1 2, 3 4

INTRODUCTION

This manual is intended to provide the necessary information regarding configuration and usage of the PCM-MIO-G-DA-1 digital I/O module. WinSystems maintains a Technical Support Group to help answer 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).

FEATURES

Analog Output

- Two 4-channel, 12-bit Digital-to-Analog (D/A) (LTC-2704CGW-12)
- Output ranges: 0-5V, 0-10V, $\pm 5V$ or $\pm 10V$
- Each channel independently software programmable for output type and range
- Output channels can be updated and cleared individually or simultaneously
- Interrupt I/O supported
- Supports industry standard signal conditioners

Digital I/O

- 48 bidirectional lines with Input, Output, or Output with Readback (WS16C48)
- 12 mA sink current per line
- Ability to generate an interrupt on signal change-of-state (24 bits)
- Write-protection mask register for each port

Power

- +5V required

Industrial Operating Temperature

- -40°C to 85°C

Form Factor

- 3.6" x 3.8" (90 mm x 96 mm)

Standard

- RoHS compliant
- LPC subtractive decode requires no BIOS changes

FUNCTIONALITY

Digital-to-Analog

The PCM-MIO-G-DA-1 provides digital-to-analog conversion input using two of the 12-bit Linear Technologies LTC-2704 devices. These SoftSpan™ quad Digital-to-Analog converters (DACs) are software programmable for either unipolar or bipolar mode with specific voltage ranges on a per channel basis. Each of the 8 channels can be programmed to any one of the six output ranges (0V to 5V, 0V to 10V, $\pm 2.5V$, $\pm 5V$, $\pm 10V$ and $-2.5V$ to $7.5V$).

Parallel I/O

The PCM-MIO-G-DA-1 utilizes the WinSystems WS16C48 ASIC high-density I/O chip. The 48 lines are each individually programmable as input or output and the first 24 lines are capable of fully latched event sensing with sense polarity being software programmable.

Interrupts

The PCM-MIO-G-DA-1 provides flexible interrupt configuration options. Each D/A converter and 24 Digital I/O are capable of generating an interrupt. They can be setup to use individual interrupts, a single shared interrupt, or any combination of the two. The interrupts are completely software configurable and require no jumpers or other configuration. The individual registers and configuration for each device are discussed in the [Software Summary](#) section and under each device.

The PCM-MIO-G-DA-1 can be configured to use IRQ's 3, 4, 5, 6, 7, 9, 10, 11, 12, 14, or 15 depending on availability in the system. IRQ's 0, 1, 2, 8, and 13 are not supported.

DMA Support

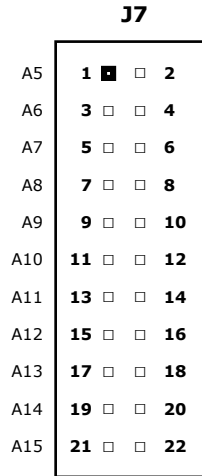
The PCM-MIO-G-DA-1 supports DMA on the D/A device. Commands can be configured on DMA channels 0, 1, 2, and 3, which are 8-bit channels. Data can be configured on DMA channels 5, 6, and 7, which are 16-bit channels.

DMA Channel 4 is not available.

CONNECTOR REFERENCE

I/O ADDRESS

J7 - Default Base I/O Address



Example - 320HEX

0	0	0	0	0	0	1	1	0	0	1	X	X	X	X	X
A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0

Jumper OPEN = 1

Jumper INSTALLED = 0

The PCM-MIO-G-DA-1 is I/O mapped and requires 32 sequential port addresses. The base address is jumper selectable at **J7**. Care should be taken to choose an I/O area that does not conflict with other resources in the system. The specific device locations and register offsets are discussed in more detail in the [Software Summary](#) section.

ANALOG OUTPUT (D/A)

J1, J3 - D/A Analog Output Connections



PCB Connector: G650-2034-1HA, TEKA SRC217C425M126-0

Mating Connector: ITW-PANCON 050-010-455A



The PCM-MIO-G-DA-1 provides digital-to-analog conversion input using two of the 12-bit Linear Technologies LTC-2704 devices. These SoftSpan™ quad Digital-to-Analog converters (DACs) are software programmable for either unipolar or bipolar mode with specific voltage ranges on a per channel basis. Each of the 8 channels can be programmed to any one of the six output ranges (0V to 5V, 0V to 10V, $\pm 2.5V, \pm 5V, \pm 10V$ and $-2.5V$ to $7.5V$).

Programming information for the D/A controller is provided in the [Software Summary](#) section of this manual.

The Analog Output channels are terminated at **J1** and **J3**. The pin definitions are shown in the illustration above.

NOTE: When the board is powered off, the D/A input has a 20K input impedance and is protected to $\pm 20V$. When the board is powered on, the D/A differential channel input has a 31K input impedance and the single-ended channel input has a 42K input impedance. Power on inputs are protected to $\pm 25V$.

DIGITAL I/O INTERFACE

J8, J9 - Digital I/O

Visual Index

PCB Connector: G650-2050-1HA, TEKA SRC225C425M126-0

Mating Connector: ITW-PANCON 050-050-455A

J9			J8		
GND	50 □ □ 49	VCC	GND	50 □ □ 49	VCC
GND	48 □ □ 47	Port 0 Bit 0	GND	48 □ □ 47	Port 3 Bit 0
GND	46 □ □ 45	Port 0 Bit 1	GND	46 □ □ 45	Port 3 Bit 1
GND	44 □ □ 43	Port 0 Bit 2	GND	44 □ □ 43	Port 3 Bit 2
GND	42 □ □ 41	Port 0 Bit 3	GND	42 □ □ 41	Port 3 Bit 3
GND	40 □ □ 39	Port 0 Bit 4	GND	40 □ □ 39	Port 3 Bit 4
GND	38 □ □ 37	Port 0 Bit 5	GND	38 □ □ 37	Port 3 Bit 5
GND	36 □ □ 35	Port 0 Bit 6	GND	36 □ □ 35	Port 3 Bit 6
GND	34 □ □ 33	Port 0 Bit 7	GND	34 □ □ 33	Port 3 Bit 7
GND	32 □ □ 31	Port 1 Bit 0	GND	32 □ □ 31	Port 4 Bit 0
GND	30 □ □ 29	Port 1 Bit 1	GND	30 □ □ 29	Port 4 Bit 1
GND	28 □ □ 27	Port 1 Bit 2	GND	28 □ □ 27	Port 4 Bit 2
GND	26 □ □ 25	Port 1 Bit 3	GND	26 □ □ 25	Port 4 Bit 3
GND	24 □ □ 23	Port 1 Bit 4	GND	24 □ □ 23	Port 4 Bit 4
GND	22 □ □ 21	Port 1 Bit 5	GND	22 □ □ 21	Port 4 Bit 5
GND	20 □ □ 19	Port 1 Bit 6	GND	20 □ □ 19	Port 4 Bit 6
GND	18 □ □ 17	Port 1 Bit 7	GND	18 □ □ 17	Port 4 Bit 7
GND	16 □ □ 15	Port 2 Bit 0	GND	16 □ □ 15	Port 5 Bit 0
GND	14 □ □ 13	Port 2 Bit 1	GND	14 □ □ 13	Port 5 Bit 1
GND	12 □ □ 11	Port 2 Bit 2	GND	12 □ □ 11	Port 5 Bit 2
GND	10 □ □ 9	Port 2 Bit 3	GND	10 □ □ 9	Port 5 Bit 3
GND	8 □ □ 7	Port 2 Bit 4	GND	8 □ □ 7	Port 5 Bit 4
GND	6 □ □ 5	Port 2 Bit 5	GND	6 □ □ 5	Port 5 Bit 5
GND	4 □ □ 3	Port 2 Bit 6	GND	4 □ □ 3	Port 5 Bit 6
GND	2 □ □ 1	Port 2 Bit 7	GND	2 □ □ 1	Port 5 Bit 7

Parallel I/O

The PCM-MIO-G-DA-1 utilizes the WinSystems WS16C48 ASIC high-density I/O chip. The 48 lines are each individually programmable as input or output and the first 24 lines are capable of fully latched event sensing with sense polarity being software programmable.

Digital I/O Connectors

The 48 lines of parallel I/O are terminated through two 50-pin connectors at **J8** and **J9**. The **J9** connector handles I/O ports 0-2 while **J8** handles ports 3-5. The pin definitions for **J8** and **J9** are shown above.

Note: Pin 49 on each connector can supply VCC to the I/O rack.

J10 - VCC Enable

Visual Index

The digital I/O connector can provide +5V to an I/O rack, when required. +5V is provided at pin 49 on connector **J9** when **J10** is jumpered, pin 3-4. +5V is provided at pin 49 of **J8** when **J10** is jumpered, pin 1-2. It is the user's responsibility to limit current to a safe value (less than 400 mA) to avoid damaging the board.

J10		
1 ■ □ 2		+5V is provided at pin 49 of J9
3 □ □ 4		+5V is provided at pin 49 of J8
		No Power at Pin 49 of J8/J9 (default)

PC/104 BUS INTERFACE

J5, J6 - PC/104



PCB Connector: **TEKA PC232-A-1BD-M (J6)**
TEKA PC220-A-1BD-M (J5)

The PC/104 bus is electrically equivalent to the 16-bit ISA bus. Standard PC/104 I/O cards can be populated on PCM-MIO-G-DA-1's connectors, located at **J5** and **J6**. The interface does not support hot swap capability. The PC/104 bus connector pin definitions are provided here for reference. Refer to the [PC/104 Bus Specification](#) for specific signal and mechanical specifications.

J5			J6		
GND	D0 ■ □ C0	GND	IOCHK#	A1 ■ □ B1	GND
MEMCS16#	D1 □ □ C1	SBHE#	SD7	A2 □ □ B2	RESET
IOCS16#	D2 □ □ C2	LA23	SD6	A3 □ □ B3	+5V
IRQ10	D3 □ □ C3	LA22	SD5	A4 □ □ B4	IRQ
IRQ11	D4 □ □ C4	LA21	SD4	A5 □ □ B5	-5V
IRQ12	D5 □ □ C5	LA20	SD3	A6 □ □ B6	DRQ2
IRQ15	D6 □ □ C6	LA19	SD2	A7 □ □ B7	-12V
IRQ14	D7 □ □ C7	LA18	SD1	A8 □ □ B8	SRDY#
DACK0#	D8 □ □ C8	LA17	SD0	A9 □ □ B9	+12V
DRQ0	D9 □ □ C9	MEMR#	IOCHRDY	A10 □ □ B10	KEY
DACK5#	D10 □ □ C10	MEMW#	AEN	A11 □ □ B11	SMEMW#
DRQ5	D11 □ □ C11	SD8	SA19	A12 □ □ B12	SMEMR#
DACK6#	D12 □ □ C12	SB9	SA18	A13 □ □ B13	IOW#
DRQ6	D13 □ □ C13	SD10	SA17	A14 □ □ B14	IOR#
DACK7#	D14 □ □ C14	SD11	SA16	A15 □ □ B15	DACK3#
DRQ7	D15 □ □ C15	SD12	SA15	A16 □ □ B16	DRQ3
+5V	D16 □ □ C16	SD13	SA14	A17 □ □ B17	DACK1#
MASTER#	D17 □ □ C17	SD14	SA13	A18 □ □ B18	DRQ1
GND	D18 □ □ C18	SD15	SA12	A19 □ □ B19	REFRESH#
GND	D19 □ □ C19	KEY	SA11	A20 □ □ B20	BCLK
			SA10	A21 □ □ B21	IRQ7
			SA9	A22 □ □ B22	IRQ6
			SA8	A23 □ □ B23	IRQ5
			SA7	A24 □ □ B24	IRQ4
			SA6	A25 □ □ B25	IRQ3
			SA5	A26 □ □ B26	DACK2#
			SA4	A27 □ □ B27	TC
			SA3	A28 □ □ B28	BALE
			SA2	A29 □ □ B29	+5V
			SA1	A30 □ □ B30	OSC
			SA0	A31 □ □ B31	GND
			GND	A32 □ □ 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.

SOFTWARE SUMMARY

REGISTERS

Register Definitions (WS16C48)

Visual
Index

The PCM-MIO-G-DA-1 uses the WinSystems exclusive ASIC device, the WS16C48. This device provides 48 lines of digital I/O. There are 17 unique registers within the WS16C48. The following table summarizes the registers, and the text that follows provides details on each of the internal registers.

BASE+	I/O Address Offset	Page 0	Page 1	Page 2	Page 3
16	00H	Port 0 I/O	Port 0 I/O	Port 0 I/O	Port 0 I/O
17	01H	Port 1 I/O	Port 1 I/O	Port 1 I/O	Port 1 I/O
18	02H	Port 2 I/O	Port 2 I/O	Port 2 I/O	Port 2 I/O
19	03H	Port 3 I/O	Port 3 I/O	Port 3 I/O	Port 3 I/O
20	04H	Port 4 I/O	Port 4 I/O	Port 4 I/O	Port 4 I/O
21	05H	Port 5 I/O	Port 5 I/O	Port 5 I/O	Port 5 I/O
22	06H	Int_Pending	Int_Pending	Int_Pending	Int_Pending
23	07H	Page/Lock	Page/Lock	Page/Lock	Page/Lock
24	08H	IRQ_REG	Pol_0	Enab_0	Int_ID0
25	09H	REV_LO	Pol_1	Enab_1	Int_ID1
26	0AH	REV_HI	Pol_2	Enab_2	Int_ID2

Register Details

Port 0 through 5 I/O

Each I/O bit in each of the six ports can be individually programmed for input or output. Writing a **0** to a bit position causes the corresponding output pin to go to a high-impedance state (pulled high by external 10 K Ω resistors). This allows it to be used as an input. When used in the input mode, a read reflects the inverted state of the I/O pin, such that a high on the pin will read as a **0** in the register. Writing a **1** to a bit position causes that output pin to sink current (up to 12 mA), effectively pulling it low.

INT_PENDING

This read-only register reflects the combined state of the INT_ID0 through INT_ID2 registers. When any of the lower three bits are set, it indicates that an interrupt is pending on the I/O port corresponding to the bit position(s) that are set. Reading this register allows an Interrupt Service Routine to quickly determine if any interrupts are pending and which I/O port has a pending interrupt.

PAGE/LOCK

This register serves two purposes. The upper two bits select the register page in use as shown here:

D7	D6	Page
0	0	Page 0
0	1	Page 1
1	0	Page 2
1	1	Page 3

Bits 5-0 allow for locking the I/O ports. A **1** written to the I/O port position will prohibit further writes to the corresponding I/O port.

IRQ_REG

This register is accessible when Page 0 is selected. It is used to select the bus routine for the generated interrupt.

REV_LO

This register is accessible when Page 0 is selected. This register returns the low byte of the current revision level for the FPGA firmware.

REV_HIGH

This register is accessible when Page 0 is selected. This register returns the high byte of the current revision level for the FPGA firmware.

POL0 - POL2

These registers are accessible when Page 1 is selected. They allow interrupt polarity selection on a port-by-port and bit-by-bit basis. Writing a **1** to a bit position selects the rising edge detection interrupts while writing a **0** to a bit position selects falling edge detection interrupts.

ENAB0 - ENAB2

These registers are accessible when Page 2 is selected. They allow for port-by-port and bit-by-bit enabling of the edge detection interrupts. When set to a **1**, the edge detection interrupt is enabled for the corresponding port and bit. When cleared to **0**, the bit's edge detection interrupt is disabled. Note that this register can be used to individually clear a pending interrupt by disabling and re-enabling the pending interrupt.

INT_ID0 – INT_ID2

These registers are accessible when Page 3 is selected. They are used to identify currently pending edge interrupts. A bit when read as a **1** indicates that an edge of the polarity programmed into the corresponding polarity register has been recognized. Note that a write to this register (value ignored) clears ALL of the pending interrupts in this register.

Digital I/O Interrupts

The interrupt registers described above are specific to the internal operation of the DIO device. It is also necessary to configure the interrupt routing to the system processor. This is accomplished with the resource and resource enable register as shown in the table below.

			7	6	5	4	3	2	1	0
RESOURCE*	2	R/W	X	X	X	X	INT(3)	INT(2)	INT(1)	INT(0)
RESOURCE ENABLE	3	W	X	X	X	DIO ACCESS	X	X	X	X

* Interrupt Routing Assignment (IRQ15-3, 1, 2, 8, and 13 not available)

A sample configuration:

1. Write **xxx1xxxx** to bit 4 of BASE +3 (select access to DIO IRQ configuration register).
2. Write IRQ selection (0-15 hex) to bits 3-0 of BASE +2.
3. Configure DIO internal IRQ as discussed above.

Master Interrupt Status Register

Although each device contains an interrupt pending status bit, a single read only register is also available to provide the status of all devices in one location. With the limited number of interrupts available, this register is very helpful by allowing your application to share a single interrupt amongst all the on-board devices.

D/A Converters

The PCM-MIO-G-DA-1 contains two Linear Tech LTC-2704 Digital-to-Analog Converter (DAC) devices. Each device is a 4-channel converter with software selectable output span.

D/A1 - Starting at BASE +8

The COMMAND register, RESOURCE register and RESOURCE ENABLE registers are used to configure the D/A device operation. The specific options of each register are detailed here.

Register	Address (Base+)	Read/Write	7	6	5	4	3	2	1	0
DATA_LO	8	R/W	LOW ORDER DATA BYTE							
			DATA-BIT 7	DATA-BIT 6	DATA-BIT 5	DATA-BIT 4	DATA-BIT 3	DATA-BIT 2	DATA-BIT 1	DATA-BIT 0
READBACK (1)	8	R	LOW ORDER DATA BYTE							
			DATA-BIT 7	DATA-BIT 6	DATA-BIT 5	DATA-BIT 4	DATA-BIT 3	DATA-BIT 2	DATA-BIT 1	DATA-BIT 0
DATA_HI	9	R/W	HIGH ORDER DATA BYTE							
			DATA-BIT 15	DATA-BIT 14	DATA-BIT 13	DATA-BIT 12	DATA-BIT 11	DATA-BIT 10	DATA-BIT 9	DATA-BIT 8
READBACK (1)	9	R	HIGH ORDER DATA BYTE							
			DATA-BIT 15	DATA-BIT 14	DATA-BIT 13	DATA-BIT 12	DATA-BIT 11	DATA-BIT 10	DATA-BIT 9	DATA-BIT 8
COMMAND (2)	10	R/W	COMMAND							
			CMD-BIT 7	CMD-BIT 6	CMD-BIT 5	CMD-BIT 4	CMD-BIT 3	CMD-BIT 2	CMD-BIT 1	CMD-BIT 0
RESOURCE (3)	10	R/W	DMA CHANNEL ASSIGNMENT				INTERRUPT ROUTING ASSIGNMENT IRQ[15-3] 0, 1, 2, 8 AND 13 NOT AVAILABLE			
			DATA REGISTER BIT 1	DATA REGISTER BIT 0	DATA REGISTER BIT 1	DATA REGISTER BIT 0	BIT 3	BIT 2	BIT 1	BIT 0
RESOURCE ENABLE	11	W	X	X	X	READBACK ENABLE	REGISTER SELECT	DATA DRQ ENABLE (4)	CMD DRQ ENABLE (4)	INTERRUPT ENABLE (4)
STATUS	11	R	DATA READY	DATA DMA REQUEST PENDING (4)	CMD DMA REQUEST PENDING (4)	INTERRUPT REQUEST PENDING (4)	REGISTER SELECT STATUS	DATA DRQ ENABLE STATUS (4)	CMD DRQ ENABLE STATUS (4)	INTERRUPT EANBLE BIT STATUS (4)

- Notes:**
- (1) Accessed when READBACK ENABLE (BASE +11 bit 4) = 1
 - (2) Accessed when REGISTER SELECT (BASE +11 bit 3) = 0
 - (3) Accessed when REGISTER SELECT (BASE +11 bit 3) = 1
 - (4) 0=Disabled, 1=Enable

D/A2 - Starting at BASE +12

As shown in the table below, interface to the 2nd device is almost identical to the first with a change in the base address. The RESOURCE ENABLE register does contain an additional register select bit, MASTER IRQ / DA2 SELECT. If this bit is set to 1, reading BASE +15 will result in the status of the Master Interrupt Status Register.

Register	Address (Base+)	Read/Write	7	6	5	4	3	2	1	0
DATA_LO	12	R/W	LOW ORDER DATA BYTE							
			DATA-BIT 7	DATA-BIT 6	DATA-BIT 5	DATA-BIT 4	DATA-BIT 3	DATA-BIT 2	DATA-BIT 1	DATA-BIT 0
READBACK (1)	12	R	LOW ORDER DATA BYTE							
			DATA-BIT 7	DATA-BIT 6	DATA-BIT 5	DATA-BIT 4	DATA-BIT 3	DATA-BIT 2	DATA-BIT 1	DATA-BIT 0
DATA_HI	13	R/W	HIGH ORDER DATA BYTE							
			DATA-BIT 15	DATA-BIT 14	DATA-BIT 13	DATA-BIT 12	DATA-BIT 11	DATA-BIT 10	DATA-BIT 9	DATA-BIT 8
READBACK (1)	13	R	HIGH ORDER DATA BYTE							
			DATA-BIT 15	DATA-BIT 14	DATA-BIT 13	DATA-BIT 12	DATA-BIT 11	DATA-BIT 10	DATA-BIT 9	DATA-BIT 8
COMMAND (2)	14	R/W	COMMAND							
			CMD-BIT 7	CMD-BIT 6	CMD-BIT 5	CMD-BIT 4	CMD-BIT 3	CMD-BIT 2	CMD-BIT 1	CMD-BIT 0
RESOURCE (3)	14	R/W	DMA CHANNEL ASSIGNMENT				INTERRUPT ROUTING ASSIGNMENT IRQ[15-3] 0, 1, 2, 8 AND 13 NOT AVAILABLE			
			DATA REGISTER BIT 1	DATA REGISTER BIT 0	DATA REGISTER BIT 1	DATA REGISTER BIT 0	BIT 3	BIT 2	BIT 1	BIT 0
RESOURCE ENABLE	15	W	X	X	D/A2 SELECT	READBACK ENABLE	REGISTER SELECT	DATA DRQ ENABLE (4)	CMD DRQ ENABLE (4)	INTERRUPT ENABLE (4)
STATUS (5)	15	R	DATA READY	DATA DMA REQUEST PENDING (4)	CMD DMA REQUEST PENDING (4)	INTERRUPT REQUEST PENDING (4)	REGISTER SELECT STATUS	DATA DRQ ENABLE STATUS (4)	CMD DRQ ENABLE STATUS (4)	INTERRUPT EABLE BIT STATUS (4)
IRQ REGISTER (6)	15	R	DATA READY	DATA DMA REQUEST PENDING (4)	CMD DMA REQUEST PENDING (4)	DA2 IRQ PENDING	DIO IRQ PENDING	DA/1 IRQ PENDING	AD/2 IRQ PENDING	AD/1 IRQ PENDING

- Notes:**
- (1) Accessed when READBACK ENABLE (BASE +15 bit 4) = 1
 - (2) Accessed when REGISTER SELECT (BASE +15 bit 3) = 0
 - (3) Accessed when REGISTER SELECT (BASE +15 bit 3) = 1
 - (4) 0=Disabled, 1=Enable
 - (5) Accessed when MASTER IRQ/DA2 SELECT (BASE +15 bit 5) = 0
 - (6) Accessed when MASTER IRQ/DA2 SELECT (BASE +15 bit 5) = 1

The Linear Tech LTC-2704 devices are unique in that each channel consists of a double-buffered data register (B1 Code and B2 Code) and a double-buffered span register (B1 Span and B2 Span). B1 buffers are the holding buffers and data is loaded into each one using a write operation, the DAC outputs are not affected. The contents of the B2 buffers can only be updated by copying the contents of B1 into B2 via an update operation initiated by the Command Code. The contents of the B2 buffers (both DAC Span and DAC Code) directly control the DAC output voltage or the DAC output range. Configuration, programming and writing of the D/A data is achieved through a series of control registers listed below for each DAC.

Command Register

Each DAC contains a command register used to configure the span and load the data. The command word consists of a 4-bit command and a 4-bit address, as shown. Each DAC contains a command register used to configure the span and load the data. The command word consists of a 4 bit command and a 4-bit address, as shown.

7	6	5	4	3	2	1	0
C3	C2	C1	C0	A3	A2	A1	A0

Command Register

C3	C2	C1	C0	COMMAND	Readback Point Current Input Word	Readback Pointer Next Input Word
0	0	1	0	Write to B1 Span DAC n	Set by Previous Command	B1 Span DAC n
0	0	1	1	Write to B1 Code DAC n	Set by Previous Command	B1 Code DAC n
0	1	0	0	Update B1 → B2 DAC n	Set by Previous Command	B2 Span DAC n
0	1	0	1	Update B1 → B2 All DAC n	Set by Previous Command	B2 Code DAC n
0	1	1	0	Write to B1 Span DAC n Update B1 → B2 DAC n	Set by Previous Command	B1 Span DAC n
0	1	1	1	Write to B1 Code DAC n Update B1 → B2 DAC n	Set by Previous Command	B2 Code DAC n
1	0	0	0	Write to B1 Span DAC n Update B1 → B2 All DACs	Set by Previous Command	B2 Span DAC n
1	0	0	1	Write to B1 Code DAC n Update B1 → B2 All DACs	Set by Previous Command	B2 Code DAC n
1	0	1	0	Read B1 Span DAC n	B1 Span DAC n	
1	0	1	1	Read B1 Code DAC n	B1 Code DAC n	
1	1	0	0	Read B2 Span DAC n	B2 Span DAC n	
1	1	0	1	Read B2 Code DAC n	B2 Code DAC n	
1	1	1	1	No Operation	Set by Previous Command	B2 Span DAC n

Codes not shown are reserved and should not be used

Address Codes

A3	A2	A1	A0	n	Readback Pointer n
0	0	0	0	DAC A	DAC A
0	0	1	0	DAC B	DAC B
0	1	0	0	DAC C	DAC C
0	1	1	0	DAC D	DAC D
1	1	1	1	All DACs	DAC A

Codes not shown are reserved and should not be used

Span Codes

The span for each channel is set by loading the desired value into the data registers, then issuing one of the span commands. The last 4 bits set the span as shown below. The rest of the data should be set to **0**.

S3	S2	S1	S0	Span
0	0	0	0	Unipolar 0V to 5V
0	0	0	1	Unipolar 0V to 10V
0	0	1	0	Bipolar -5V to 5V
0	0	1	1	Bipolar -10V to 10V
0	1	0	0	Bipolar -2.5V to 2.5V
0	1	0	1	Bipolar -2.5V to 7.5V

Codes not shown are reserved and should not be used

Readback Enable

Each time a command is issued to one of Linear Tech LTC-2704 devices, the value of one of the buffers is simultaneously shifted out of the device. Except when issuing one of the specific readback commands (Ax, Bx, Cx, Dx), the data returned corresponds to the Readback Pointer from the previous command as shown in the [Command Codes Table](#). The Readback Enable bit must be set to **1** to read this data.

Note: If the Readback Enable bit is set to **0**, a read of the DAC data registers will return the last value written to that register not the Readback value of the actual buffers.

D/A Interrupts

To operate using interrupt mode, IRQ routing must be configured and interrupts enabled for each device. This is achieved with the Resource and Resource Enable registers. The following would apply to D/A1:

1. Write **xxxx1xxx** to bit 3 of BASE +11 (select access to Resource Register).
2. Write IRQ selection (0-15 hex) to bits 3-0 of BASE +10 (**xF Hex** = IRQ 15).
3. Write **xxxxxx1** BASE +11 to enable the IRQ.

Enabling an interrupt for D/A2 can be achieved in the same manner with the appropriate offset.

It is possible for both devices to share an interrupt or use individual interrupts. When sharing interrupts, the most efficient method to determine which device generated an interrupt request is to utilize the Master Interrupt Status Register.

DMA Support

DMA operation is available for this device. A sample of these operations under DOS is provided in the attached software section. These operations under other operating systems can be quite complex and are beyond the scope of this manual.

D/A Examples

The most basic method is to first set the output span for a channel and then write the output value for that channel. Notice that the configuration and data write operations can each be performed with either a single or double instruction sequence. Each channel can be updated individually using command values 6x & 7x (for configuration and output data, respectively) which will pre-load the value and present it to the DAC with a single instruction sequence. The second option is to pre-load the configuration and output data using command values 2x & 3x and then present the values to the DAC either individually (with command value 4x) or simultaneously (with command value 5x).

Example 1 - Single Instruction Sequence

To configure and write data to a DAC channel, each with a single command sequence, is very simple. The configuration must be set first and then the data output is written. Of course the span configuration is only required to be set once unless changes are required during the application.

1. Write xxxx0xxx to bit 3 of BASE +11.	Select access to CMD
2. Write Span data 0000xxxx to BASE +8.	Where xxxx = Span
3. Write 00000000 (zero) BASE +9.	High order byte for Span
4. Write CMD 01100xxx to BASE +10.	Where xxx = DAC channel
5. Write Low Byte data to BASE +8.	
6. Write High Byte data to BASE +9.	
7. Write CMD 01110xxx to BASE +10.	Where xxx = DAC channel
8. Additional channels are then programmed by repeating steps 2 through 7.	

Example 2 - Double Instruction Sequence

The second option is to pre-load the configuration and output data using command values 2x and 3x and then present the values to the DAC either individually using command value 4x simultaneously with command value 5x.

This example will demonstrate pre-loading the span configuration and data output values for each DAC channel and then presenting the information simultaneously to all DAC channels.

1. Write xxxx0xxx to bit 3 of BASE +11.	Select access to CMD
2. To set Span Configuration for Channel 0:	
Write Span Configuration data 00000000 (zero) to BASE +8.	Set Span = 0V to 5V
Write 00000000 (zero) to BASE +9.	High Order Data Byte
Write CMD 00100000 to BASE +10.	Move data to B1 Span
3. To set Span Configuration for Channel 1:	
Write Span Configuration data 00000001 to BASE +8.	Set Span = 0V to 10V
Write 00000000 (zero) to BASE +9.	High Order Data Byte
Write CMD 00100010 to BASE +10.	Move data to B1 Span
4. To set Span Configuration for Channel 2:	
Write Span Configuration data 00000010 to BASE +8.	Set Span = -5V to 5V
Write 00000000 (zero) to BASE +9.	High Order Data Byte
Write CMD 00100100 to BASE +10.	Move data to B1 Span
5. To set Span Configuration for Channel 3:	
Write Span Configuration data 00000011 to BASE +8.	Set Span = -10V to 10V
Write 00000000 (zero) to BASE +9.	High Order Data Byte
Write CMD 00100110 to BASE +10.	Move data to B1 Span
6. To pre-load Data Output for Channel 0:	
Write Low Byte data to BASE +8.	
Write High Byte data to BASE +9.	
Write CMD 00110000 to BASE +10.	Move data to B1 Code
7. To pre-load Data Output for Channel 1:	
Write Low Byte data to BASE +8.	
Write High Byte data to BASE +9.	
Write CMD 00110010 to BASE +10.	Move data to B1 Code
8. To pre-load Data Output for Channel 2:	
Write Low Byte data to BASE +8.	
Write High Byte data to BASE +9.	
Write CMD 00110100 to BASE +10.	Move data to B1 Code
9. To pre-load Data Output for Channel 3:	
Write Low Byte data to BASE +8.	
Write High Byte data to BASE +9.	
Write CMD 00110110 to BASE +10.	Move data to B1 Code
10. To simultaneously update all DAC channels:	

If the application requires all DAC channels to be configured to the same output span, command value 8x supports this action with a single instruction sequence. Likewise if all DAC channels will be written with the same data output then command value 9x both pre-loads and presents the value to all DAC channels with a single instruction sequence.

WS16C48 PROGRAMMING REFERENCE

This section provides basic documentation for the included I/O routines. It is intended that the accompanying source code equip the programmer with a basic library of I/O functions for the WS16C48 or can serve as the basis from which application-specific code can be derived.

Function Definitions

This section describes each of the functions contained in the driver. Where necessary, short examples will be provided to illustrate usage. Any application making use of any of the driver functions should include the header file UIO48.H, which includes the function prototypes and the needed constant definitions.

Note that all of the functions utilize the concept of *bit_number*. The *bit_number* is a value from 1 to 48 (1 to 24 for interrupt related functions) that correlates to a specific I/O pin. Bit *_number* 1 is Port 0 Bit 0 and continues through to bit *_number* 48 at Port 5 Bit 7.

INIT_IO - Initializes I/O, set all ports to input
Syntax
<code>void init_io(unsigned io_address);</code>
Description
This function takes a single argument: io_address - The I/O address of the WS16C48 chip.
There is no return value. This function initializes all I/O pins for input (sets them high), disables all interrupt settings, and sets the image values.

READ_BIT – Reads an I/O port Pin
Syntax
<code>int read_bit(int bit_number);</code>
Description
This function takes a single argument: bit_number – A value from 1 to 48 specifying the I/O pin to read from.
This function returns the state of the I/O pin. A 1 is returned if the I/O pin is low and a 0 is returned if the pin is high.

WRITE_BIT - Writes a 1 or 0 to an I/O Pin
Syntax
<code>void write_bit(int bit_number, int value);</code>
Description
This function takes two arguments: bit_number – A value from 1 to 48 specifying the I/O pin to be acted upon. value - Is either 1 or 0.
This function allows for writing of a single bit to either a 0 or a 1 as specified by the second argument. There is no return value and other bits in the I/O port are not affected.

SET_BIT - Sets the specified I/O Pin
<p>Syntax</p> <pre>void set_bit(int bit_number);</pre>
<p>Description</p> <p>This function takes a single argument: bit_number – A value from 1 to 48 specifying the I/O pin to be set.</p>
<p>This function sets the specified I/O port bit. Note that setting a bit results in the I/O pin actually going low. There is no return value and other bits in the same I/O port are unaffected.</p>

CLR_BIT – Clears the specified I/O Pin
<p>Syntax</p> <pre>int clr_bit(int bit_number);</pre>
<p>Description</p> <p>This function takes a single argument: bit_number – A value from 1 to 48 specifying the I/O pin to clear.</p>
<p>This function clears the specified I/O bit. Note that clearing the I/O bit results in the actual I/O pin going high. This function does not affect any bits other than the one specified.</p>

ENAB_INT - Enables Edge Interrupt, Select Polarity
<p>Syntax</p> <pre>void enab_int(int bit_number, int polarity);</pre>
<p>Description</p> <p>This function takes a single argument: bit_number – A value from 1 to 24, specifying the appropriate bit.</p> <p>polarity - Specifies rising or falling edge polarity detect. The constraints RISING and FALLING are defined in UIO48.H.</p>
<p>This function enables the edge detection circuitry for the specified bit at the specified polarity. It does not unmask the interrupt controller, install vectors, or handle interrupts when they occur. There is no return value and only the specified bit is affected.</p>

DISAB_INT - Disables Edge Detect Interrupt Detection**Syntax**

```
void disab_int(int bit_number);
```

Description

This function takes a single argument:
bit_number - A value from 1 to 24 specifying the appropriate bit.

This function shuts down the edge detection interrupts for the specified bit. There is no return value and no harm is done by calling this function for a bit which did not have edge detection interrupts enabled. There is no effect on any other bits.

CLR_INT – Clears the specified pending interrupt**Syntax**

```
void clr_int(int bit_number);
```

Description

This function takes a single argument:
bit_number – A value from 1 to 24 specifying the bit number to reset the interrupt.

This function clears a pending interrupt on the specified bit. It does this by disabling and reenabling the interrupt. The net result after the call is that the interrupt is not longer pending and is renamed for the next transition of the same polarity. Calling this function on a bit that has not been enabled for interrupts will result in its interrupt being enabled with an undefined polarity. Calling this function with no interrupt pending will have no adverse effect. Only the specified bit is affected.

GET_INT - Retrieves bit number of pending interrupt**Syntax**

```
void get_int(void);
```

Description

This function requires no arguments.

This function returns either a 0 for no bit interrupts pending or a value between 1 and 24 representing a bit number that has a pending edge detect interrupt. The function returns with the first interrupt found and begins its search at Port 0 Bit 0 proceeding through to Port 2 Bit 7. It is necessary to use either clr_int() or disab_int() to avoid returning the same bit continuously. This function may be used in an application's ISE or can be used in the foreground to poll for bit transitions.

Sample Programs

There are three sample programs in source code form included on the PCM-MIO-G-DA-1 file set. These programs are not useful by themselves but are provided to illustrate the usage of the I/O functions provided in UIO48.C.

FLASH.C

This program was compiled with Borland C/C++ version 3.1 on the command line with: **bcc flash.c uio48.c**

This program illustrates the most basic usage of the WS16C48. It uses three functions from the driver code. The `io_init()` function is used to initialize the I/O functions and the `set_bit()` and `clr_bit()` functions are used to sequence through all 48 bits turning each on and then off in turn.

POLL.C

This program was compiled with Borland C/C++ version 3.1 on the command line with: **bcc poll.c uio48.c**

This program illustrates additional features of the WS16C48 and the I/O library functions. It programs the first 24 bits for input, arms them for falling edge detection, and then polls using the library routine `get_init()` to determine if any transitions have taken place.

INTS.C

This program was compiled with Borland C/C++ version 3.1 on the command line with: **bcc int.c uio48.c**

This program is identical in function to the POLL.C program, except that interrupts are active and all updating of the transition counters is completed in the background during the interrupt service routine.

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](#) in the Appendix of this manual. These I/O routines along with the sample program should provide for a good basis on which to build an application using the features of the WS16C48.

CABLES

Part Number	Description
Cables	
CBL-115-4	4ft., Opto rack interface

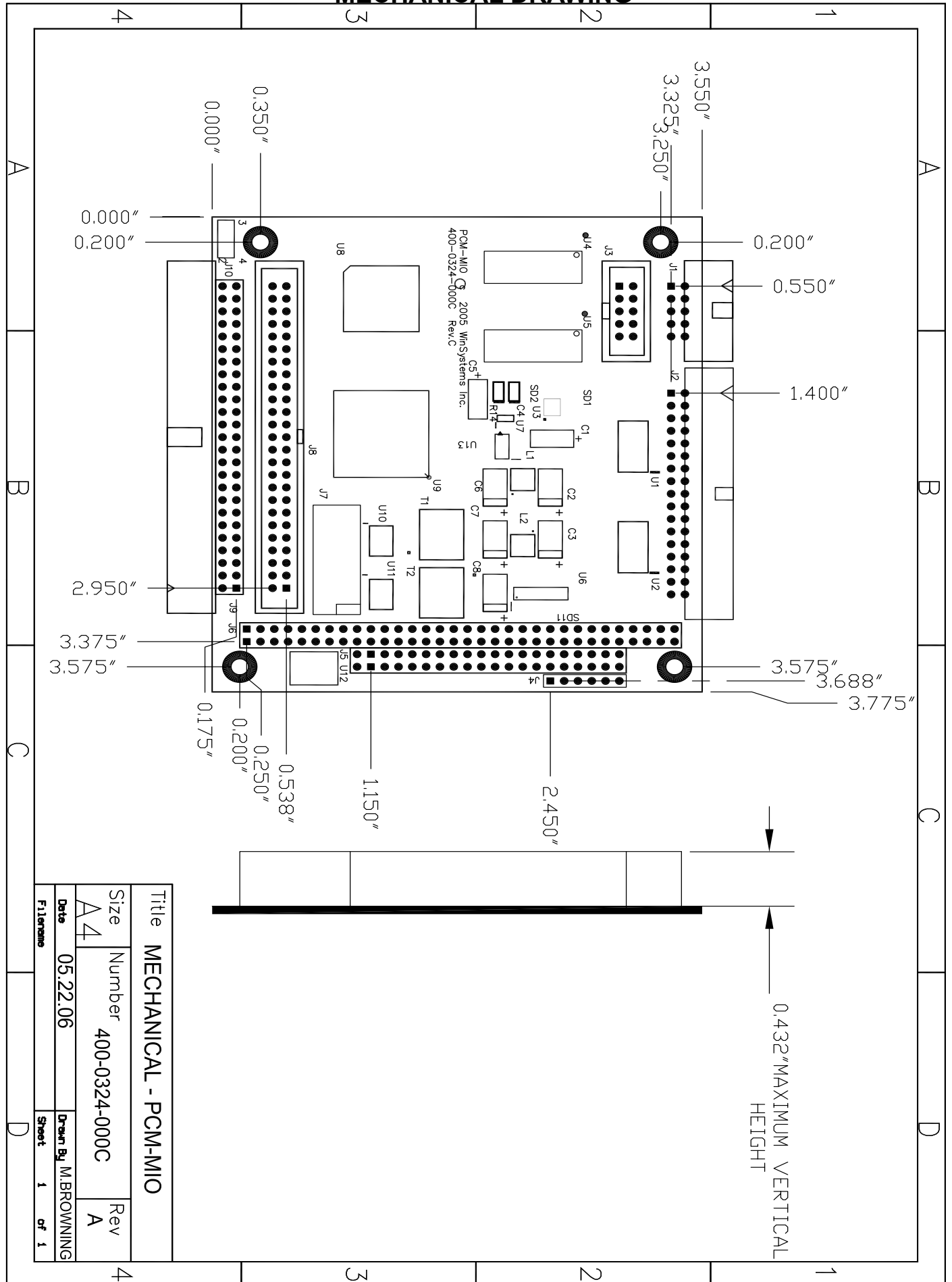
SOFTWARE DRIVERS & EXAMPLES

Examples

(For WS16C48 Digital I/O Chip)

Linux driver - Kernel 2.6	pcmmio_linux.zip
DOS Drivers and Examples	mio_dos.zip
Windows XP Drivers and Example	mio_xp.zip

MECHANICAL DRAWING



Title		MECHANICAL - PCM-MIO	
Size	Number	Rev	
A4	400-0324-000C	A	
Date	05.22.06	Drawn By	M. BROWNING
Filename		Sheet	1 of 1

APPENDIX

C Source Code Listings

C Source Code Listings

/* UIO48.H

Copyright 1996 by WinSystems Inc.

Permission is hereby granted to the purchaser of the WinSystems UIO cards and CPU products incorporating the UIO device, to distribute any binary file or files compiled using this source code directly or in any work derived by the user from this file. In no case may the source code, original or derived from this file, be distributed to any third party except by explicit permission of WinSystems. This file is distributed on an "As-is" basis and no warranty as to performance, fitness of purposes, or any other warranty is expressed or implied. In no case shall WinSystems be liable for any direct or indirect loss or damage, real or consequential resulting from the usage of this source code. It is the user's sole responsibility to determine fitness for any considered purpose.

*/

/******

* Name : uio48.h

*

* Project : PCM-UIO48 Software Samples/Examples

*

* Date : October 30, 1996

*

* Revision: 1.00

*

* Author : Steve Mottin

*

*

* Changes :

*

* Date Revision Description

*

* 10/30/96 1.00 Created

*

*/

```
#define RISING 1
#define FALLING 0
```

```
void init_io(unsigned io_address);
int read_bit(int bit_number);
void write_bit(int bit_number);
void set_bit(int bit_number);
void clr_bit(int bit_number);
void enab_int(int bit_number, int polarity);
void disab_int(int bit_number);
void clr_int(int bit_number);
int get_int(void);
```

```
/* UIO48.C
```

```
Copyright 1996 by WinSystems Inc.
```

```
Permission is hereby granted to the purchaser of the WinSystems
UIO cards and CPU products incorporating the UIO device, to distribute
any binary file or files compiled using this source code directly or
in any work derived by the user from this file. In no case may the
source code, original or derived from this file, be distributed to any
third party except by explicit permission of WinSystems. This file is
distributed on an "As-is" basis and no warranty as to performance,
fitness of purposes, or any other warranty is expressed or implied.
In no case shall WinSystems be liable for any direct or indirect loss
or damage, real or consequential resulting from the usage of this
source code. It is the user's sole responsibility to determine
fitness for any considered purpose.
```

```
*/
/*****
```

```
* Name : uio48.c
*
* Project : PCM-UIO48 Software Samples/Examples
*
* Date : October 30, 1996
*
* Revision: 1.00
*
* Author : Steve Mottin
*
```

```
*****
```

```
* Changes :
```

```
* Date Revision Description
* -----
* 10/30/96 1.00 Created
```

```
*****
```

```
*/
```

```
#include <dos.h>
```

```
/* This global holds the base address of the UIO chip */
```

```
unsigned base_port;
```

```
/* This global array holds the image values of the last write to each I/O
ports. This allows bit manipulation routines to work without having to
actually do a read-modify-write to the I/O port.
```

```
*/
```

```
unsigned port_images[6];
```

```
/*=====
```

```
* INIT_IO
```

```
* This function take a single argument :
```

```
*
```

```
* io_address : This is the base I/O address of the 16C48 UIO Chip
* on the board.
```

```
*
```

```
* This function initializes all I/O pins for input, disables all interrupt
```

```

* sensing, and sets the image values.
*
*=====*/

void init_io(unsigned io_address)
{
int x;

    /* Save the specified address for later use */

    base_port = io_address;

    /* Clear all of the I/O ports. This also makes them inputs */

    for(x=0; x < 7; x++)
        outportb(base_port+x, 0);

    /* Clear our image values as well */

    for(x=0; x < 6; x++)
        port_images[x] = 0;

    /* Set page 2 access, for interrupt enables */

    outportb(base_port+7,0x80);

    /* Clear all interrupt enables */

    outportb(base_port+8,0);
    outportb(base_port+9,0);
    outportb(base_port+0x0a,0);

    /* Restore normal page 0 register access */
    outportb(base_port+7,0);

}

/*=====
*
*          READ_BIT
*
* This function takes a single argument :
*
* bit_number  : The integer argument specifies the bit number to read.
*               Valid arguments are from 1 to 48.
*
* return value : The current state of the specified bit, 1 or 0.
*
* This function returns the state of the current I/O pin specified by
* the argument bit_number.
*
*=====*/

int read_bit(int bit_number)
{
    unsigned port;
    int val;

    /* Adjust the bit_number to 0 to 47 numbering */

    --bit_number;

```

```

    /* Calculate the I/O port address based on the updated bit_number */
    port = (bit_number / 8) + base_port;

    /* Get the current contents of the port */
    val = inportb(port);

    /* Get just the bit we specified */
    val = val & (1 << (bit_number % 8));

    /* Adjust the return for a 0 or 1 value */

    if(val)
        return 1;

    return 0;
}

/*=====
*
*          WRITE_BIT
*
* This function takes two arguments :
*
*
* bit_number : The I/O pin to access is specified by bit_number 1 to 48.
*
* val : The setting for the specified bit, either 1 or 0.
*
* This function sets the specified I/O pin to either high or low as dictated
* by the val argument. A non zero value for val sets the bit.
*
*=====*/

void write_bit(int bit_number, int val)
{
    unsigned port;
    unsigned temp;
    unsigned mask;

    /* Adjust bit_number for 0 based numbering */

    --bit_number;

    /* Calculate the I/O address of the port based on the bit number */
    port = (bit_number / 8) + base_port;

    /* Use the image value to avoid having to read the port first. */
    temp = port_images[bit_number / 8];          /* Get current value */

    /* Calculate a bit mask for the specified bit */
    mask = (1 << (bit_number % 8));

    /* Check whether the request was to set or clear and mask accordingly */

    if(val)          /* If the bit is to be set */

```

```

        temp = temp | mask;
    else
        temp = temp & ~mask;

    /* Update the image value with the value we're about to write */
    port_images[bit_number / 8] = temp;

    /* Now actually update the port. Only the specified bit is affected */
    outportb(port,temp);
}

/*=====
 *
 *          SET_BIT
 *
 *
 * This function takes a single argument :
 *
 * bit_number : The bit number to set.
 *
 * This function sets the specified bit.
 *
 *=====*/

void set_bit(int bit_number)
{
    write_bit(bit_number,1);
}

/*=====
 *
 *          CLR_BIT
 *
 *
 * This function takes a single argument :
 *
 * bit_number : The bit number to clear.
 *
 * This function clears the specified bit.
 *
 *=====*/

void clr_bit(int bit_number)
{
    write_bit(bit_number,0);
}

/*=====
 *
 *          ENAB_INT
 *
 *
 * This function takes two arguments :
 *
 * bit_number : The bit number to enable interups for. Range from 1 to 48.
 *
 * polarity : This specifies the polarity of the interrupt. A non-zero
 *            argument enables rising-edge interrupt. A zero argument
 *            enables the interrupt on the falling edge.
 *
 * This function enables within the 16C48 an interrupt for the specified bit
 * at the specified polarity. This function does not setup the interrupt
 * controller, nor does it supply an interrupr handler.
 *
 *=====*/

```

```

*
*=====*/

void enab_int(int bit_number, int polarity)
{
unsigned port;
unsigned temp;
unsigned mask;

    /* Adjust for 0 based numbering */
    --bit_number;

    /* Calculate the I/O address based upon the bit number */
    port = (bit_number / 8) + base_port + 8;

    /* Calculate a bit mask based on the specified bit number */
    mask = (1 << (bit_number % 8));

    /* Turn on page 2 access */
    outportb(base_port+7,0x80);

    /* Get the current state of the interrupt enable register */
    temp = inportb(port);

    /* Set the enable bit for our bit number */
    temp = temp | mask;

    /* Now update the interrupt enable register */
    outportb(port,temp);

    /* Turn on access to page 1 for polarity control */
    outportb(base_port+7,0x40);

    /* Get the current state of the polarity register */
    temp = inportb(port);          /* Get current polarity settings */

    /* Set the polarity according to the argument in the image value */
    if(polarity)                  /* If the bit is to be set */
        temp = temp | mask;
    else
        temp = temp & ~mask;

    /* Write out the new polarity value */
    outportb(port,temp);

    /* Set access back to Page 0 */
    outportb(base_port+7,0x0);

}
*=====

```

```

*
*           DISAB_INT
*
* This function takes a single argument :
*
* bit_number : Specifies the bit number to act upon. Range is from 1 to 48.
*
* This function shuts off the interrupt enabled for the specified bit.
*
*=====*/

void disab_int(int bit_number)
{
unsigned port;
unsigned temp;
unsigned mask;

    /* Adjust the bit_number for 0 based numbering */
    --bit_number;

    /* Calculate the I/O Address for the enable port */
    port = (bit_number / 8) + base_port + 8;

    /* Calculate the proper bit mask for this bit number */
    mask = (1 << (bit_number % 8));

    /* Turn on access to page 2 registers */
    outportb(base_port+7,0x80);

    /* Get the current state of the enable register */
    temp = inportb(port);

    /* Clear the enable bit int the image for our bit number */
    temp = temp & ~mask;

    /* Update the enable register with the new information */
    outportb(port,temp);

    /* Set access back to page 0 */
    outportb(base_port+7,0x0);
}

/*=====*/
*
*           CLR_INT
*
* This function takes a single argument :
*
* bit_number : This argument specifies the bit interrupt to clear. Range
*             is 1 to 24.
*
*
* This function is use to clear a bit interrupt once it has been recognized.
* The interrupt left enabled.

```

```

*
*=====*/

void clr_int(int bit_number)
{
unsigned port;
unsigned temp;
unsigned mask;

    /* Adjust for 0 based numbering */
    --bit_number;

    /* Calculate the correct I/O address for our enable register */
    port = (bit_number / 8) + base_port + 8;

    /* Calculate a bit mask for this bit number */
    mask = (1 << (bit_number % 8));

    /* Set access to page 2 for the enable register */
    outportb(base_port+7,0x80);

    /* Get current state of the enable register */
    temp = inportb(port);

    /* Temporarily clear only OUR enable. This clears the interrupt */
    temp = temp & ~mask;          /* clear the enable for this bit */

    /* Write out the temporary value */
    outportb(port,temp);

    /* Re-enable our interrupt bit */
    temp = temp | mask;

    /* Write it out */
    outportb(port,temp);

    /* Set access back to page 0 */
    outportb(base_port+7,0x0);
}

/*=====
*
*          GET_INT
*
* This function take no arguments.
*
* return value : The value returned is the highest level bit interrupt
*                currently pending. Range is 1 to 24.
*
* This function returns the highest level interrupt pending. If no interrupt

```

```

* is pending, a zero is returned. This function does NOT clear the interrupt.
*
*=====*/

int get_int(void)
{
int temp;
int x;

    /* read the master interrupt pending register, mask off undefined bits */

    temp = inportb(base_port+6) & 0x07;

    /* If there are no interrupts pending, return a 0 */

    if((temp & 7) == 0)
        return(0);

    /* There is something pending, now we need to identify what it is */

    /* Set access to page 3 for interrupt id registers */

    outportb(base_port+7,0xc0);

    /* Read interrupt ID register for port 0 */

    temp = inportb(base_port+8);

    /* See if any bit set, if so return the bit number */

    if(temp !=0)
    {
        for(x=0; x <=7; x++)
        {
            if(temp & (1 << x))
            {
                outportb(base_port+7,0); /* Turn off access */
                return(x+1);           /* Return bitnumber with active int */
            }
        }
    }

    /* None in Port 0, read port 1 interrupt ID register */

    temp = inportb(base_port+9);

    /* See if any bit set, if so return the bit number */

    if(temp !=0)
    {
        for(x=0; x <=7; x++)
        {
            if(temp & (1 << x))
            {
                outportb(base_port+7,0); /* Turn off access */
                return(x+9);           /* Return bitnumber with active int */
            }
        }
    }

    /* Lastly, read status of port 2 int id */

```

```

temp = inportb(base_port+0x0a);    /* Read port 2 status */

/* If any pending, return the appropriate bit number */

if(temp !=0)
{
    for(x=0; x <=7; x++)
    {
        if(temp & (1 << x))
        {
            outportb(base_port+7,0); /* Turn off access */
            return(x+17);           /* Return bitnumber with active int */
        }
    }
}

/* We should never get here unless the hardware is misbehaving but just
to be sure. We'll turn the page access back to 0 and return a 0 for
no interrupt found.
*/

outportb(base_port+7,0);
return 0;
}

```

```
/* FLASH.C
```

```
Copyright 1996-2001 by WinSystems Inc.
```

```
Permission is hereby granted to the purchaser of the WinSystems
UIO cards and CPU products incorporating the UIO device, to distribute
any binary file or files compiled using this source code directly or
in any work derived by the user from this file. In no case may the
source code, original or derived from this file, be distributed to any
third party except by explicit permission of WinSystems. This file is
distributed on an "As-is" basis and no warranty as to performance,
fitness of purposes, or any other warranty is expressed or implied.
In no case shall WinSystems be liable for any direct or indirect loss
or damage, real or consequential resulting from the usage of this
source code. It is the user's sole responsibility to determine
fitness for any considered purpose.
```

```
*/
```

```
#include <stdio.h>
#include <conio.h>
#include <dos.h>
#include "uio48.h"
```

```
/* This is where we have our board jumpered to */
```

```
#define BASE_PORT 0x120
```

```
/* This is an ultra-simple demonstration program of some of the functions
available in the UIO48 source code library. This program simply sets and
clears each I/O line in succession. It was tested by hooking LEDs to all
of the I/O lines and watching the lit one race through the bits.
```

```
*/
```

```
void main()
```

```
{
int x;
```

```
/* Initialize all I/O bits, and set then for input */
```

```
init_io(BASE_PORT);
```

```
/* We'll repeat our sequencing until a key is pressed */
```

```
while(!kbhit())
```

```
{
```

```
/* We will light the LED attached to each of the 48 lines */
```

```
for(x=1; x <=48; x++)
```

```
{
```

```
/* Setting the bit lights the LED */
```

```
set_bit(x);
```

```
/* The wait time is subjective. We liked 100mS */
```

```
delay(100);
```

```
/* Now turn off the LED */
```

```
clr_bit(x);
```

```
}
```

```
}
```

```
getch();
```

```
}
```

```
/* POLL.C
```

```
Copyright 1996-2001 by WinSystems Inc.
```

```
Permission is hereby granted to the purchaser of the WinSystems
UIO cards and CPU products incorporating the UIO device, to distribute
any binary file or files compiled using this source code directly or
in any work derived by the user from this file. In no case may the
source code, original or derived from this file, be distributed to any
third party except by explicit permission of WinSystems. This file is
distributed on an "As-is" basis and no warranty as to performance,
fitness of purposes, or any other warranty is expressed or implied.
In no case shall WinSystems be liable for any direct or indirect loss
or damage, real or consequential resulting from the usage of this
source code. It is the user's sole responsibility to determine
fitness for any considered purpose.
```

```
*/
```

```
#include <stdio.h>
#include <conio.h>
#include "uio48.h"
```

```
#define BASE_PORT 0x120
```

```
/* This program uses the edge detection interrupt capability of the
WS16C48 to count transitions on the first 24 lines. It does this
however, no by using true interrupts but by polling for transitions
using the get_int() function.
```

```
*/
```

```
/* Our transition totals are stored in this array */
```

```
unsigned int_counts[25];
```

```
/* Definitions for local functions */
```

```
void check_ints(void);
```

```
void main()
```

```
{
int x;
```

```
/* Initialize the I/O ports. Set all I/O pins to input */
```

```
init_io(BASE_PORT);
```

```
/* Initialize our transition counts, and enable falling edge
transition interrupts.
```

```
*/
```

```
for(x=1; x<25; x++)
```

```
{
```

```
int_counts[x] = 0; /* Clear the counts */
```

```
enab_int(x,FALLING); /* Enable the falling edge interrupts */
```

```

}

/* Clean up the screen for our display. Nothing fancy */
clrscr();

for(x=1; x<25; x++)
{
    gotoxy(1,x);
    printf("Bit number %02d ",x);
}

/* We will continue to display until any key is pressed */

while(!kbhit())
{
    /* Retrieve any pending transitions and update the counts */

    check_ints();

    /* Display the current count values */

    for(x=1; x < 25; x++)
    {
        gotoxy(16,x);
        printf("%05u",int_counts[x]);
    }
    getch();
}

void check_ints()
{
    int current;

    /* Get the bit number of a pending transition interrupt */

    current = get_int();

    /* If it's 0 there are none pending */

    if(current == 0)
        return;

    /* Clear and rearm this one so we can get it again */

    clr_int(current);

    /* Tally a transition for this bit */

    ++int_counts[current];
}

```

```
/* INTS.C
```

```
Copyright 1996-2001 by WinSystems Inc.
```

```
Permission is hereby granted to the purchaser of the WinSystems
UIO cards and CPU products incorporating the UIO device, to distribute
any binary file or files compiled using this source code directly or
in any work derived by the user from this file. In no case may the
source code, original or derived from this file, be distributed to any
third party except by explicit permission of WinSystems. This file is
distributed on an "As-is" basis and no warranty as to performance,
fitness of purposes, or any other warranty is expressed or implied.
In no case shall WinSystems be liable for any direct or indirect loss
or damage, real or consequential resulting from the usage of this
source code. It is the user's sole responsibility to determine
fitness for any considered purpose.
```

```
*/
```

```
#include <stdio.h>
#include <dos.h>
#include <conio.h>
#include "uio48.h"
```

```
#define BASE_PORT 0x120
```

```
/* This program like the poll.c sample uses the edge detection interrupt
capability of the WS16C48 to count edge transitions. Unlike poll.c,
however this program actually uses interrupts and update all of the
transition counters in the background.
```

```
*/
```

```
/* Our transition totals are stored in this global array */
```

```
unsigned int_counts[25];
```

```
/* Function declarations for local functions */
```

```
void check_ints(void);
void interrupt int_handler(void);
void interrupt (*old_handler)(void);
```

```
void main()
```

```
{
int x;
```

```
    /* Initialize the I/O ports. Set all I/O pins to input */
```

```
    init_io(BASE_PORT);
```

```
    /* Install an interrupt handler for the board */
```

```
    /* We disable interrupts whenever we're changing the environment */
```

```
    disable();          /* Disable interrupts during initialization */
```

```
    /* Get the old handler and save it for later resoration */
```

```
    old_handler = getvect(0x72); /* Hardwired for IRQ10 */
```

```

/* Install out new interrupt handler */

setvect(0x72,int_handler);

/* Clear the transition count values and enable the falling edge
interrupts.
*/

for(x=1; x<25; x++)
{
    int_counts[x] = 0; /* Clear the counts */
    enab_int(x,FALLING); /* Enable the falling edge interrupts */
}

/* Unmask the interrupt controller */

outportb(0xa1,(inportb(0xa1) & 0xfb)); /* Unmask IRQ 10 */

/* Reenable interrupts */
enable();

/* Set up the display */

clrscr(); /* Clear the Text Screen */

for(x=1; x<25; x++)
{
    gotoxy(1,x);
    printf("Bit Number %02d ",x);
}

/* We will continuously print the transition totals until a
key is pressed */

/* All of the processing of the transition interrupts, including
updating the counts is done in the background when an interrupt
occurs.
*/

while(!kbhit())
{
    for(x=1; x < 25; x++)
    {
        gotoxy(16,x);
        printf("%05u",int_counts[x]);
    }
}

getch();

/* Disable interrupts while we restore things */

disable();

/* Mask off the interrupt at the interrupt controller */

outportb(0xa1,inportb(0xa1) | 0x02); /* Mask IRQ 10 */

/* Restore the old handler */

```

```

    setvect(0x72,old_handler); /* Put back the old interrupt handler */

    /* Reenable interrupts. Things are back they way they were before we
       started.
    */

    enable();
}

/* This function is executed when an edge detection interrupt occurs */

void interrupt int_handler(void)
{
    int current;

    /* Get the current interrupt pending. There really should be one
       here or we shouldn't even be executing this function.
    */

    current = get_int();

    /* We will continue processing pending edge detect interrupts until
       there are no more present. In which case current == 0
    */

    while(current)
    {
        /* Clear the current one so that it's ready for the next edge */

        clr_int(current);

        /* Tally up one for the current bit number */

        ++int_counts[current];

        /* Get the next one, if any others pending */

        current = get_int();
    }

    /* Issue a non-specific end of interrupt command (EOI) to the
       interrupt controller. This rearms it for the next shot.
    */

    outportb(0xa0,0x20); /* Do non-specific EOI */
    outportb(0x20,0x20);
}

```

WARRANTY INFORMATION

(<http://www.winsystems.com/company/warranty.cfm>)

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, INCIDENTAL 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.