USER MANUAL

Accessory 59E

8-Channel 12-Bit ADC/DAC Board

3Ax-603494-xUxx

May 26, 2011

DELTA TAU
Data Systems, Inc.
NEW IDEAS IN MOTION ...

Single Source Machine Control *Power // Flexibility // Ease of Use*
21314 Lassen Street Chatsworth, CA 91311 // Tel. (818) 998-2095 Fax. (818) 998-7807 // www.deltatau.com

Copyright Information

To report errors or inconsistencies, call or email:

Delta Tau Data Systems, Inc. Technical Support
Phone: (818) 717-5656
Fax: (818) 998-7807
Email: support@deltatau.com
Website: http://www.deltatau.com

Operating Conditions

All Delta Tau Data Systems, Inc. motion controller products, accessories, and amplifiers contain static sensitive components that can be damaged by incorrect handling. When installing or handling Delta Tau Data Systems, Inc. products, avoid contact with highly insulated materials. Only qualified personnel should be allowed to handle this equipment.

In the case of industrial applications, we expect our products to be protected from hazardous or conductive materials and/or environments that could cause harm to the controller by damaging components or causing electrical shorts. When our products are used in an industrial environment, install them into an industrial electrical cabinet or industrial PC to protect them from excessive or corrosive moisture, abnormal ambient temperatures, and conductive materials. If Delta Tau Data Systems, Inc. products are directly exposed to hazardous or conductive materials and/or environments, we cannot guarantee their operation.

WARNING

A Warning identifies hazards that could result in personal injury or death. It precedes the discussion of interest.

Caution

A Caution identifies hazards that could result in equipment damage. It precedes the discussion of interest.

Note

A Note identifies information critical to the user's understanding or use of the equipment. It follows the discussion of interest.

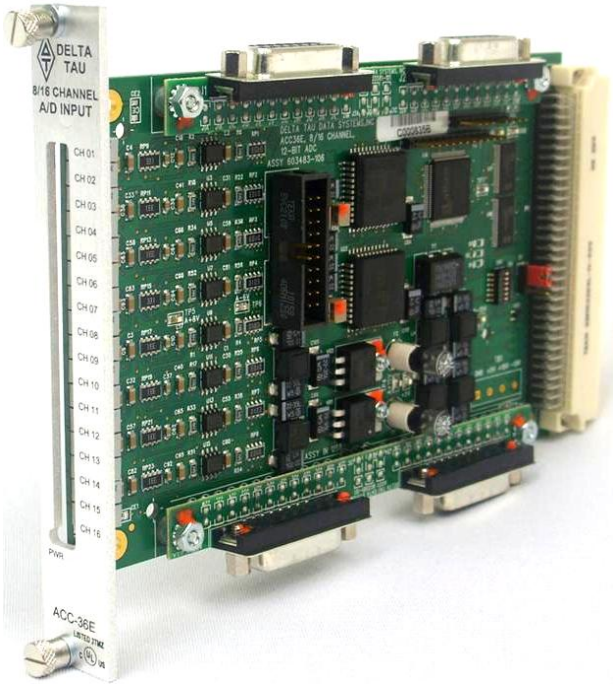
REVISION HISTORY				
REV.	**DESCRIPTION**	**DATE**	**CHG**	**APPVD**
1	CORR. TB3 HEADER P. 30, NEW SCHEMATIC	01/11/08	CP	S.MILICI
2	UPDATED J3 JUMPER DESCRIPTION	07/22/09	CP	S.SATTARI
3	ADDED UL SEAL TO MANUAL COVER	10/01/09	CP	S.FIERRO
4	CORRECTED VOLTAGE RANGES (P. 5, 10-11); CORRECTED M-VARIABLE NAMES (P. 11-12)	8/11/10	DCDP	R.N
5	MANUAL REFORMATTING; ADDED POWER PMAC	10/15/10	DCDP	R.N
6	REFURBISHED UMAC MACRO SECTION	4/27/11	DCDP	R.N

Table of Contents

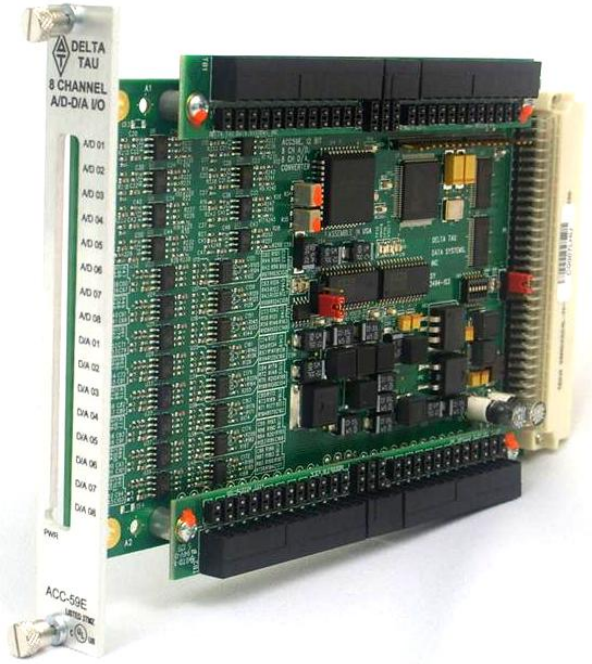
INTRODUCTION

The Accessory 59E (ACC-59E) is a 3U Euro bus compatible card for the UMAC rack family (Turbo, Power, and Macro). It has eight 12-bit Analog Inputs and eight 12-bit Analog Outputs.

ACC-59E:
D-Sub Option



ACC-59E:
Terminal Block Option



SPECIFICATIONS

Environmental Specifications

Description	Specification	Notes
Operating Temperature	0 °C to 45 °C	
Storage Temperature	-25 °C to 70 °C	
Humidity	10% to 95%	Non-Condensing

Electrical Specifications

Power Requirements

Whether providing the ACC-59E with power from the 3U backplane bus through P1 or externally (standalone mode) through TB1, the power requirements (± 10%) are: +5 V @ 110 mA
+15 V @ 75 mA
-15 V @ 95 mA

Note

The DAC outputs are optically isolated and should not be sharing the same ±15 V power supply with the non-optically isolated ADCs.

ACC-59E Fuse

Manufacturer	Specification	Delta Tau Part Number
Little Fuse	125 V @ 2.0 A	273.500

ADC Bandwidth

The ADC input cutoff frequency is around 300 Hz.

DAC Bandwidth

The DAC output cutoff frequency is around 1.5 kHz.

Adjustment Potentiometers (Pots)

Channel	ADC Pots	DAC Pots
1	R8	R36
2	R16	R37
3	R24	R50
4	R32	R51
5	R7	R64
6	R15	R65
7	R23	R78
8	R31	R79

Note

Pots R34 and R35 (for Delta Tau's internal use) are used to adjust the ADCs' voltage references.

ADC and DAC Chips

The Analog-to-Digital Converter (ADC) chip used in ACC-59E is the MAX180 monolithic device manufactured by Maxim Integrated Products. These devices have a 12-bit resolution with ±1 LSB linearity specification. For more details on the ADC chips, please refer to the data sheet published by the manufacturer:

Document 19-3950; Rev 0, 6/91
Complete, 8-Channel, 12-Bit Data Acquisition Systems
Maxim Integrated Products

The Digital-to-Analog Converter (DAC) chip used in ACC-59E is the DAC7625 manufactured by Burr-Brown Corporation. This device is a 12-bit quad voltage output digital-to-analog converter with guaranteed 12-bit monotonic performance from –40 °C to +85 °C. For more details about the DAC chips, please refer to the data sheet published by the manufacturer:

DAC7624 and DAC7625
12-Bit Quad Voltage Output Digital-To-Analog Converter
Burr-Brown Corporation

Physical Specifications

Description	Specification	Notes
Terminal Block Connectors	FRONT-MC1,5/12-ST3,81 FRONT-MC1,5/5-ST3,81 FRONT-MC1,5/3-ST3,81	Terminal Blocks from Phoenix Contact (UL-94V0)
DB Option Connectors	DB15 Female	UL-94V0

ADDRESSING ACC-59E

Dip switch SW1 specifies the base address of the ACC-59E in a 3U TURBO / POWER UMAC, or MACRO Station rack.

Turbo/Power UMAC, Macro Station Dip Switch Settings

Chip Select	Base Address (TURBO/MACRO) and Base Offset (POWER)				SW1 Positions					
	TURBO	MACRO	POWER		6	5	4	3	2	1
			Offset	Index (n)						
CS10	Y:$78C00	Y:$8800	$A00000	0	ON	ON	ON	ON	ON	ON
	Y:$79C00	Y:$9800	$A08000	4	ON	ON	ON	**OFF**	ON	ON
	Y:$7AC00	Y:$A800	$A10000	8	ON	ON	**OFF**	ON	ON	ON
	Y:$7BC00	Y:$B800	$A18000	12	ON	ON	**OFF**	**OFF**	ON	ON
CS12	Y:$78D00	Y:$8840	$B00000	1	ON	ON	ON	ON	ON	**OFF**
	Y:$79D00	Y:$9840	$B08000	5	ON	ON	ON	**OFF**	ON	**OFF**
	Y:$7AD00	Y:$A840	$B10000	9	ON	ON	**OFF**	ON	ON	**OFF**
	Y:$7BD00	Y:$B840	$B18000	13	ON	ON	**OFF**	**OFF**	ON	**OFF**
CS14	Y:$78E00	Y:$8880	$C00000	2	ON	ON	ON	ON	**OFF**	ON
	Y:$79E00	Y:$9880	$C08000	6	ON	ON	ON	**OFF**	**OFF**	ON
	Y:$7AE00	Y:$A880	$C10000	10	ON	ON	**OFF**	ON	**OFF**	ON
	Y:$7BE00	Y:$B880	$C18000	14	ON	ON	**OFF**	**OFF**	**OFF**	ON
CS16	Y:$78F00	Y:$88C0	$D00000	3	NA					
	Y:$79F00	Y:$98C0	$D08000	7	NA					
	Y:$7AF00	Y:$A8C0	$D10000	11	NA					
	Y:$7BF00	Y:$B8C0	$D18000	15	NA					

Note

- The ACC-59E can only be assigned to Chip Select 10, 12, or 14.
- ON designates Closed. OFF designates Open.
- Factory default is all ON.
- The maximum addressable number of ACC-59Es (or similar type accessories) in a single rack is 12.

Legacy Macro Dip Switch Settings

Chip Select	Base Address (Alternate)	SW1 Positions					
		6	5	4	3	2	1
10	Y:$B800 (Y:$FFE0)	ON (**OFF**)	ON (**OFF**)	**OFF**	**OFF**	ON	ON
12	Y:$B840 (Y:$FFE8)	ON (**OFF**)	ON (**OFF**)	**OFF**	**OFF**	ON	**OFF**
14	Y:$B880 (Y:$FFF0)	ON (**OFF**)	ON (**OFF**)	**OFF**	**OFF**	**OFF**	ON
16	Y:$B8C0 (Y:$FFF8)	NA					

Note

The Legacy Macro base addresses are double mapped. Set SW1 positions 5 & 6 to OFF if the alternate addressing is desired.

Hardware Address Limitations

Historically, two types of accessory cards have been designed for the UMAC 3U bus type rack; type A and type B cards. They can be sorted out as follows:

Name	Type	Category	Possible Number of Addresses	Maximum Number of cards in 1 rack
ACC-9E	A	General I/O	4	12
ACC-10E	A	General I/O	4	
ACC-11E	A	General I/O	4	
ACC-12E	A	General I/O	4	
ACC-14E	B	General I/O	16	16
ACC-28E	B	Analog I/O	16	
ACC-36E	B	Analog I/O	16	
ACC-53E	B	Feedback	16	
ACC-57E	B	Feedback	16	
ACC-58E	B	Feedback	16	
ACC-59E	B	Analog I/O	12	12
ACC-65E	B	General I/O	16	16
ACC-66E	B	General I/O	16	
ACC-67E	B	General I/O	16	
ACC-68E	B	General I/O	16	
ACC-84E	B	Feedback	12	12

Addressing Type A and Type B accessory cards in a UMAC or MACRO station rack requires paying attention to the following set of rules:

Populating Rack with Type A Cards Only (No Conflicts)

In this mode, the card(s) can potentially use any available Address/Chip Select.

Note

Type A cards can have up to 4 different base addresses. Knowing that each card can be configured (jumper settings) to use the lower, middle or higher byte of a specific base address, it is possible to populate a single rack with a maximum of 12 Type A accessory cards.

Populating Rack with Type B Cards Only (No Conflicts)

In this mode, the card(s) can potentially use any available Address/Chip Select.

Populating Rack with Type A & Type B Cards (Possible Conflicts)

- Typically, Type A and Type B cards should not share the same Chip Select. If this configuration is possible, then the next couple of rules does not apply, and can be disregarded.
- Type A cards cannot share the same Chip Select as Type B cards.
- Type A cards can share the same Chip Select as Type B general I/O cards. However, in this mode, Type B cards naturally use the lower byte (default), and Type A cards must be set up (jumper settings) to the middle/high byte of the selected base address.

USING ACC-59E WITH TURBO UMAC

Setting Up the Analog Inputs (ADCs)

The A/D converter chips used on the ACC-59E multiplex the resulting data, and therefore it is mandatory to read each input one at a time. With Turbo UMAC, this can be done in two ways:

- Automatic Read
- Manual (semi-automatic) Read

Note

The automatic read feature is available with Turbo firmware version 1.936 and newer. It supports up to two ACC-59E cards in one rack.

One should use the automatic read method when one wants to use the ADCs for servo feedback. One should use the manual read method when one needs to read ADCs from more than two ACC-36E or ACC-59E cards or when one wants the timing of the ADC conversion to occur exactly when specified, rather than at an automatic time that the user does not control.

Automatic ADC Read

The automatic read function demultiplexes the data into individual registers and stores them once every I5060 phase cycles in accessible memory locations. It can handle up to a total of 32 ADC channels (16 pairs). There are 16 ADCs (8 pairs) per base address (or ACC-59E card) for a maximum of 2 base addresses (or two ACC-59E cards) as shown below:

Pair #	1st ACC-59E		
1	ADC#1	&	ADC#9
2	ADC#2	&	ADC#10
3	ADC#3	&	ADC#11
4	ADC#4	&	ADC#12
5	ADC#5	&	ADC#13
6	ADC#6	&	ADC#14
7	ADC#7	&	ADC#15
8	ADC#8	&	ADC#16

Pair #	2nd ACC-59E		
9	ADC#1	&	ADC#9
10	ADC#2	&	ADC#10
11	ADC#3	&	ADC#11
12	ADC#4	&	ADC#12
13	ADC#5	&	ADC#13
14	ADC#6	&	ADC#14
15	ADC#7	&	ADC#15
16	ADC#8	&	ADC#16

The ACC-59E has only 8 ADC channels, which is less than what the automatic read function can handle. Thus, one configures the software to explicitly sample actually 16 ADCs (8 pairs) per card. However, in reality, the upper 8 ADCs (4 pairs) are ignored (i.e., ADC#9 thru ADC#16), and only the lower 8 are used.

These are the necessary steps for setting up the automatic read function:

1. **Configure A/D Processing Ring Size**: I5060. This is the number of ADC pairs to be demuxed.
   I5060 = (Number of ADC Pairs to Sample)

Note

- Saving I5060 to a value greater than zero, the A/D Convert Enable I5080 is automatically set to 1 on power-up or reset. Subsequently setting I5080 to 0, the user can suspend the de-multiplexing, to be resumed by setting I5080=1.
- Each ADC is automatically updated every I5060 phase cycles.

2. **Configure A/D Ring Pointers**: I5061 thru I5076.
   I5061 through I5068 specify the offset widths (hex) of the first 8 pairs (1st ACC-59E).
   I5069 through I5076 specify the offset widths (hex) of the second 8 pairs (2nd ACC-59E).
   $(Offset Width) = $(Card Base Address) – $078800

3. **Configure A/D Ring Convert Codes**: I5081 thru I5096.
   I5081 through I5088 define the voltage modes for each of the first 8 pairs (1st ACC-59E).
   I5089 through I5096 define the voltage modes for each of the second 8 pairs (2nd ACC-59E).

Setup I-Variable	Definition of n	Mode	Voltages
I5081 through I5096 = n	n = ADC# - 1	Unipolar	Only positive
	n = ADC# + 7	Bipolar	Positive/Negative

4. Assign M-Variables to demuxed data registers, then **Save** and reset (**$$$**).

Automatic ADC Read Example 1

Setting up Turbo UMAC with an ACC-59E at base address $078C00 to automatically read all 8 ADCs, allowing the user to choose unipolar or bipolar mode:

1. A/D Processing Ring Size

```
I5060=8                    ; Demux 8 ADC pairs
```

2. A/D Ring Pointers

```
I5061,8=$400               ; ADC pairs 1 through 8 offset width: $400 = $078C00-$78800
```

3. A/D Convert Codes

Pair #	Unipolar	Bipolar
1	I5081=0 ; ADC#1 Unipolar	I5081=8 ; ADC#1 Bipolar
2	I5082=1 ; ADC#2 Unipolar	I5082=9 ; ADC#2 Bipolar
3	I5083=2 ; ADC#3 Unipolar	I5083=10 ; ADC#3 Bipolar
4	I5084=3 ; ADC#4 Unipolar	I5084=11 ; ADC#4 Bipolar
5	I5085=4 ; ADC#5 Unipolar	I5085=12 ; ADC#5 Bipolar
6	I5086=5 ; ADC#6 Unipolar	I5086=13 ; ADC#6 Bipolar
7	I5087=6 ; ADC#7 Unipolar	I5087=14 ; ADC#7 Bipolar
8	I5088=7 ; ADC#8 Unipolar	I5088=15 ; ADC#8 Bipolar

Note

These convert codes are applicable to ADCs #9 thru #16 as well by definition, but knowing that the ACC-59E carries only 8 channels, they can be ignored.

4. Assigning M-Variables to Demuxed Data Registers

Unipolar	Bipolar
M5061->Y:$003400,12,12,U ; ADC#1 Unipolar	M5061->Y:$003400,12,12,S ; ADC#1 Bipolar
M5062->Y:$003402,12,12,U ; ADC#2 Unipolar	M5062->Y:$003402,12,12,S ; ADC#2 Bipolar
M5063->Y:$003404,12,12,U ; ADC#3 Unipolar	M5063->Y:$003404,12,12,S ; ADC#3 Bipolar
M5064->Y:$003406,12,12,U ; ADC#4 Unipolar	M5064->Y:$003406,12,12,S ; ADC#4 Bipolar
M5065->Y:$003408,12,12,U ; ADC#5 Unipolar	M5065->Y:$003408,12,12,S ; ADC#5 Bipolar
M5066->Y:$00340A,12,12,U ; ADC#6 Unipolar	M5066->Y:$00340A,12,12,S ; ADC#6 Bipolar
M5067->Y:$00340C,12,12,U ; ADC#7 Unipolar	M5067->Y:$00340C,12,12,S ; ADC#7 Bipolar
M5068->Y:$00340E,12,12,U ; ADC#8 Unipolar	M5068->Y:$00340E,12,12,S ; ADC#8 Bipolar

Note

- Issue a **Save** and reset (**$$$**) after download to enable the A/D ring processing
- If the convert code is unipolar, the M-Variable assigned to read the ADC result must also be unipolar

Automatic ADC Read Example 2

Another ACC-59E has been added to produce a total of two cards at $078C00 and $079C00, respectively.

1. A/D Processing Ring Size

```
I5060=16                ; Demux 16 ADC pairs
```

2. A/D Ring Pointers

```
I5061,8=$400            ; ADC pairs 1 thru 8 offset width   $400  = $078C00 - $78800
I5069,8=$1400           ; ADC pairs 9 thru 16 offset width  $1400 = $079C00 - $078800
```

3. A/D Convert Codes

	Pair#	Unipolar	Bipolar
1st ACC-59E	1	I5081=0 ; ADC#1 1st ACC-59E Unipolar	I5081=8 ; ADC#1 1st ACC-59E Bipolar
	2	I5082=1 ; ADC#2 1st ACC-59E Unipolar	I5082=9 ; ADC#2 1st ACC-59E Bipolar
	3	I5083=2 ; ADC#3 1st ACC-59E Unipolar	I5083=10 ; ADC#3 1st ACC-59E Bipolar
	4	I5084=3 ; ADC#4 1st ACC-59E Unipolar	I5084=11 ; ADC#4 1st ACC-59E Bipolar
	5	I5085=4 ; ADC#5 1st ACC-59E Unipolar	I5085=12 ; ADC#5 1st ACC-59E Bipolar
	6	I5086=5 ; ADC#6 1st ACC-59E Unipolar	I5086=13 ; ADC#6 1st ACC-59E Bipolar
	7	I5087=6 ; ADC#7 1st ACC-59E Unipolar	I5087=14 ; ADC#7 1st ACC-59E Bipolar
	8	I5088=7 ; ADC#8 1st ACC-59E Unipolar	I5088=15 ; ADC#8 1st ACC-59E Bipolar
2nd ACC-59E	9	I5089=0 ; ADC#1 2nd ACC-59E Unipolar	I5089=8 ; ADC#1 2nd ACC-59E Bipolar
	10	I5090=1 ; ADC#2 2nd ACC-59E Unipolar	I5090=9 ; ADC#2 2nd ACC-59E Bipolar
	11	I5091=2 ; ADC#3 2nd ACC-59E Unipolar	I5091=10 ; ADC#3 2nd ACC-59E Bipolar
	12	I5092=3 ; ADC#4 2nd ACC-59E Unipolar	I5092=11 ; ADC#4 2nd ACC-59E Bipolar
	13	I5093=4 ; ADC#5 2nd ACC-59E Unipolar	I5093=12 ; ADC#5 2nd ACC-59E Bipolar
	14	I5094=5 ; ADC#6 2nd ACC-59E Unipolar	I5094=13 ; ADC#6 2nd ACC-59E Bipolar
	15	I5095=6 ; ADC#7 2nd ACC-59E Unipolar	I5095=14 ; ADC#7 2nd ACC-59E Bipolar
	16	I5096=7 ; ADC#8 2nd ACC-59E Unipolar	I5096=15 ; ADC#8 2nd ACC-59E Bipolar

4. Assigning M-Variables to Demuxed Data Registers

	Unipolar	Bipolar
1st ACC-59E	M5061->Y:$003400,12,12,U ; ADC#1 1st ACC-59E	M5061->Y:$003400,12,12,S ; ADC#1 1st ACC-59E
	M5062->Y:$003402,12,12,U ; ADC#2 1st ACC-59E	M5062->Y:$003402,12,12,S ; ADC#2 1st ACC-59E
	M5063->Y:$003404,12,12,U ; ADC#3 1st ACC-59E	M5063->Y:$003404,12,12,S ; ADC#3 1st ACC-59E
	M5064->Y:$003406,12,12,U ; ADC#4 1st ACC-59E	M5064->Y:$003406,12,12,S ; ADC#4 1st ACC-59E
	M5065->Y:$003408,12,12,U ; ADC#5 1st ACC-59E	M5065->Y:$003408,12,12,S ; ADC#5 1st ACC-59E
	M5066->Y:$00340A,12,12,U ; ADC#6 1st ACC-59E	M5066->Y:$00340A,12,12,S ; ADC#6 1st ACC-59E
	M5067->Y:$00340C,12,12,U ; ADC#7 1st ACC-59E	M5067->Y:$00340C,12,12,S ; ADC#7 1st ACC-59E
	M5068->Y:$00340E,12,12,U ; ADC#8 1st ACC-59E	M5068->Y:$00340E,12,12,S ; ADC#8 1st ACC-59E
2nd ACC-59E	M5069->Y:$003410,12,12,U ; ADC#1 2nd ACC-59E	M5069->Y:$003410,12,12,S ; ADC#1 2nd ACC-59E
	M5070->Y:$003412,12,12,U ; ADC#2 2nd ACC-59E	M5070->Y:$003412,12,12,S ; ADC#2 2nd ACC-59E
	M5071->Y:$003414,12,12,U ; ADC#3 2nd ACC-59E	M5071->Y:$003414,12,12,S ; ADC#3 2nd ACC-59E
	M5072->Y:$003416,12,12,U ; ADC#4 2nd ACC-59E	M5072->Y:$003416,12,12,S ; ADC#4 2nd ACC-59E
	M5073->Y:$003418,12,12,U ; ADC#5 2nd ACC-59E	M5073->Y:$003418,12,12,S ; ADC#5 2nd ACC-59E
	M5074->Y:$00341A,12,12,U ; ADC#6 2nd ACC-59E	M5074->Y:$00341A,12,12,S ; ADC#6 2nd ACC-59E
	M5075->Y:$00341C,12,12,U ; ADC#7 2nd ACC-59E	M5075->Y:$00341C,12,12,S ; ADC#7 2nd ACC-59E
	M5076->Y:$00341E,12,12,U ; ADC#8 2nd ACC-59E	M5076->Y:$00341E,12,12,S ; ADC#8 2nd ACC-59E

Note

Issue a **Save** and reset (**$$$**) after downloading to enable the A/D ring processing.

Manual ADC Read

The manual read method consists of selecting the desired channel with an M-Variable, reading it with another M-Variable, and then storing it into local memory.

Following are the necessary steps for implementing the manual ADC read method. The following example parameters are for an ACC-59E at base address $078C00, allowing the user to choose to read unipolar or bipolar input modes:

1. Point an available M-Variable (12-bit wide) to the lower 12 bits of the ACC-59E base address. This is the "**Data Read**" register of the selected channel. It can be defined as follows:

   Unsigned (positive input voltage only) Data Read pointer for Unipolar Mode:

```
M5000->Y:$078C00,0,12,U
```

   Signed (negative/positive input voltage) Data Read pointer for Bipolar Mode:

```
M5001->Y:$078C00,0,12,S
```

2. Point an available M-Variable (24-bit wide unsigned) to the base address of the ACC-59E. This is the "**Channel Select"** Pointer.

```
M5004->Y:$078C00,0,24,U
```

3. Point an M-Variable to the bit that indicates when the ADC has finished its conversion. This is the **ADC Ready** bit. It becomes 1 when the ADC conversion has finished.

```
M5005->Y:$078F30,5,1                    // ADC Ready Bit (Channels 1-8)
```

   The address of this bit is based on the card base address as follows:

Base Address	ADC Ready Bit
Y:$78C00	M5005->Y:$078F30,5,1
Y:$79C00	M5005->Y:$078F34,5,1
Y:$7AC00	M5005->Y:$078F38,5,1
Y:$7BC00	M5005->Y:$078F3C,5,1
Y:$78D00	M5005->Y:$079F30,5,1
Y:$79D00	M5005->Y:$079F34,5,1
Y:$7AD00	M5005->Y:$079F38,5,1
Y:$7BD00	M5005->Y:$079F3C,5,1
Y:$78E00	M5005->Y:$07AF30,5,1
Y:$79E00	M5005->Y:$07AF34,5,1
Y:$7AE00	M5005->Y:$07AF38,5,1
Y:$7BE00	M5005->Y:$07AF3C,5,1
Y:$78F00	M5005->Y:$07BF30,5,1
Y:$79F00	M5005->Y:$07BF34,5,1
Y:$7AF00	M5005->Y:$07BF38,5,1
Y:$7BF00	M5005->Y:$07BF3C,5,1

Note

Because the ADCs are multiplexed, there is only one ADC Ready Bit, since only one ADC will be processed at a time.

4. Using the **Channel Select** Pointer, specify the channel # (ADC #) and **voltage mode**, as follows:

   M5000 = ADC# - 1 for Unipolar
   M5001 = ADC# + 7 for Bipolar

	Channel Select Value	
ADC #	**Unipolar**	**Bipolar**
1	0	8
2	1	9
3	2	10
4	3	11
5	4	12
6	5	13
7	6	14
8	7	15

5. Wait for the **ADC Ready** bit to become 1, then read and/or copy data from the **Data Read** M-Variables.

Note

These are only suggested definitions for the Data Read and Channel Select pointers. The user can choose whichever M-Variable numbering he or she desires.

ADC Manual Read Example PLCs

Ultimately, the above procedure can be implemented in a PLC script to read all channels consecutively and consistently, creating a "custom automatic" function.

Setting up Turbo UMAC with an ACC-59E at base address $078C00:

Unipolar PLC Example

Reads all 8 ADC channels of the ACC-59E successively as unipolar and stores the results.

```
End Gat
Del Gat
Close

#define DataRead                M5000      ; Data Read register
#define ChSelect                M5004      ; Channel Select Pointer
#define ADCReady                M5005      ; ADC Ready Bit
DataRead->Y:$078C00,0,12,U                 ; Unsigned for unipolar
ChSelect->Y:$078C00,0,24,U                 ; Channel Select
ADCReady->Y:$078F30,5,1                    ; ADC Ready Bit

#define ADC1                    P2001      ; Channel 1 ADC storage
#define ADC2                    P2002      ; Channel 2 ADC storage
#define ADC3                    P2003      ; Channel 3 ADC storage
#define ADC4                    P2004      ; Channel 4 ADC storage
#define ADC5                    P2005      ; Channel 5 ADC storage
#define ADC6                    P2006      ; Channel 6 ADC storage
#define ADC7                    P2007      ; Channel 7 ADC storage
#define ADC8                    P2008      ; Channel 8 ADC storage

Open PLC 1 Clear
ChSelect=0                              ; Select ADC Channel #1 (ChSelect = ADC#-1)
While(ADCReady != 1) EndWhile           ; Wait for ADC conversion to finish
ADC1=DataRead                           ; Copy result into storage

ChSelect=1                              ; Select ADC Channel #2 (ChSelect = ADC#-1)
While(ADCReady != 1) EndWhile           ; Wait for ADC conversion to finish
ADC2=DataRead                           ; Copy result into storage

ChSelect=2                              ; Select ADC Channel #3 (ChSelect = ADC#-1)
While(ADCReady != 1) EndWhile           ; Wait for ADC conversion to finish
ADC3=DataRead                           ; Copy result into storage

ChSelect=3                              ; Select ADC Channel #4 (ChSelect = ADC#-1)
While(ADCReady != 1) EndWhile           ; Wait for ADC conversion to finish
ADC4=DataRead                           ; Copy result into storage

ChSelect=4                              ; Select ADC Channel #5 (ChSelect = ADC#-1)
While(ADCReady != 1) EndWhile           ; Wait for ADC conversion to finish
ADC5=DataRead                           ; Copy result into storage

ChSelect=5                              ; Select ADC Channel #6 (ChSelect = ADC#-1)
While(ADCReady != 1) EndWhile           ; Wait for ADC conversion to finish
ADC6=DataRead                           ; Copy result into storage

ChSelect=6                              ; Select ADC Channel #7 (ChSelect = ADC#-1)
While(ADCReady != 1) EndWhile           ; Wait for ADC conversion to finish
ADC7=DataRead                           ; Copy result into storage

ChSelect=7                              ; Select ADC Channel #8 (ChSelect = ADC#-1)
While(ADCReady != 1) EndWhile           ; Wait for ADC conversion to finish
ADC8=DataRead                           ; Copy result into storage
Close
```

Bipolar PLC Example

Reads all 8 ADC channels of the ACC-59E successively as bipolar and stores the results.

```
End Gat
Del Gat
Close

#define DataRead                M5001      ; Data Read register
#define ChSelect                M5004      ; Channel Select Pointer
#define ADCReady                M5005      ; ADC Ready Bit
DataRead->Y:$078C00,0,12,S                 ; Signed for bipolar
ChSelect->Y:$078C00,0,24,U                 ; Channel Select
ADCReady->Y:$078F30,5,1                    ; ADC Ready Bit

#define ADC1                    P2001      ; Channel 1 ADC storage
#define ADC2                    P2002      ; Channel 2 ADC storage
#define ADC3                    P2003      ; Channel 3 ADC storage
#define ADC4                    P2004      ; Channel 4 ADC storage
#define ADC5                    P2005      ; Channel 5 ADC storage
#define ADC6                    P2006      ; Channel 6 ADC storage
#define ADC7                    P2007      ; Channel 7 ADC storage
#define ADC8                    P2008      ; Channel 8 ADC storage

Open PLC 1 Clear
ChSelect=8                              ; Select ADC Channel #1 (ChSelect = ADC#+7)
While(ADCReady != 1) EndWhile           ; Wait for ADC conversion to finish
ADC1=DataRead                           ; Copy result into storage

ChSelect=9                              ; Select ADC Channel #2 (ChSelect = ADC#+7)
While(ADCReady != 1) EndWhile           ; Wait for ADC conversion to finish
ADC2=DataRead                           ; Copy result into storage

ChSelect=10                             ; Select ADC Channel #3 (ChSelect = ADC#+7)
While(ADCReady != 1) EndWhile           ; Wait for ADC conversion to finish
ADC3=DataRead                           ; Copy result into storage

ChSelect=11                             ; Select ADC Channel #4 (ChSelect = ADC#+7)
While(ADCReady != 1) EndWhile           ; Wait for ADC conversion to finish
ADC4=DataRead                           ; Copy result into storage

ChSelect=12                             ; Select ADC Channel #5 (ChSelect = ADC#+7)
While(ADCReady != 1) EndWhile           ; Wait for ADC conversion to finish
ADC5=DataRead                           ; Copy result into storage

ChSelect=13                             ; Select ADC Channel #6 (ChSelect = ADC#+7)
While(ADCReady != 1) EndWhile           ; Wait for ADC conversion to finish
ADC6=DataRead                           ; Copy result into storage

ChSelect=14                             ; Select ADC Channel #7 (ChSelect = ADC#+7)
While(ADCReady != 1) EndWhile           ; Wait for ADC conversion to finish
ADC7=DataRead                           ; Copy result into storage

ChSelect=15                             ; Select ADC Channel #8 (ChSelect = ADC#+7)
While(ADCReady != 1) EndWhile           ; Wait for ADC conversion to finish
ADC8=DataRead                           ; Copy result into storage
Close
```

Testing the Analog Inputs

The Analog Inputs can be brought into the ACC-59E as single ended (ADC+ & Ground) or differential (ADC+ & ADC-) signals.

Note

In single-ended mode, ADC- should to be tied to analog ground for full resolution and proper operation.

Reading the input signals in software counts using the predefined M-Variables should show the following:

ADC Input	Single-Ended [V] ADC+ <=> AGND	Differential [V] ADC+ <=> ADC-	Software Counts
Unipolar Mode	0	0	0
	10	10	2047
	20	20	4095
Bipolar Mode	-10	-10	-2047
	0	0	0
	10	10	2047

Using an Analog Input for Servo Feedback

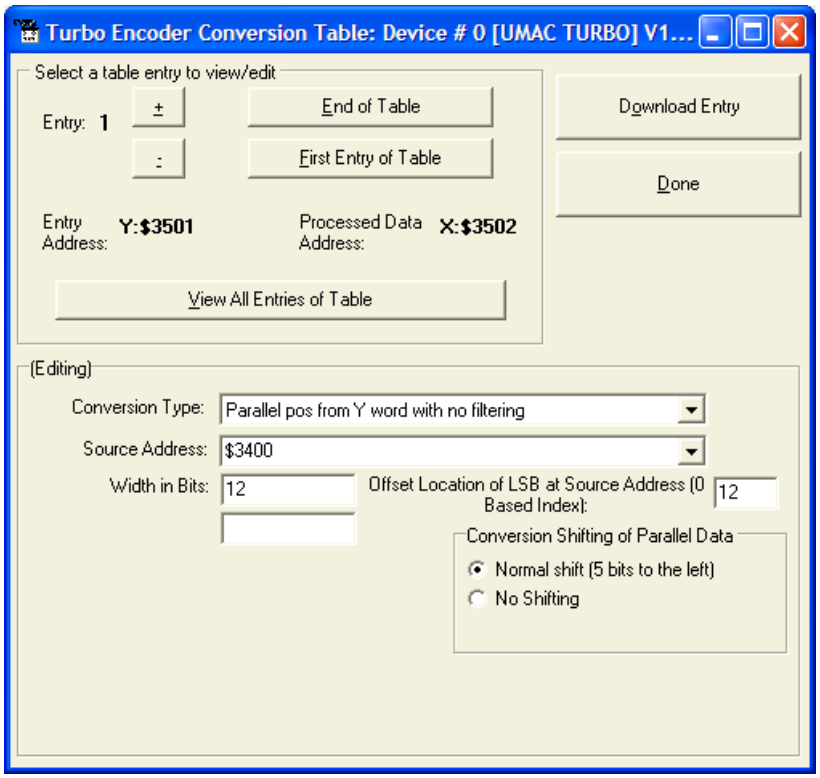
The ACC-59E analog inputs can be used as a feedback device for a servo motor.

Note

Refer to Delta Tau's released application notes or Turbo User Manual for details on implementing cascaded-loop control (i.e., force, height control around position loop).

For simplicity, the automatic ADC read function is recommended for this application.

Example: Setting up Motor #1 with position and velocity feedback from ADC channel 1.
The analog input is brought into the Encoder Conversion Table as a parallel Y word with no filtering:



The equivalent code in Turbo PMAC I8000 Encoder Conversion Table parameters:

```
I8000=$203400    ; Unfiltered parallel pos of location Y:$3400
I8001=$00C00C    ; Width and Offset. (Processed Data Location)
```

The position and velocity pointers are then set to the processed data address (i.e. $3502)

```
I103=$3502       ; Motor #1 position loop feedback address
I104=$3502       ; Motor #1 velocity loop feedback address
```

Analog Input Power-On Position

Some analog devices are absolute along the travel range of the motor (e.g., in hydraulic piston applications). Generally, it is desirable to obtain the motor position (input voltage) on power up or reset. This procedure can be done in a simple PLC on power-up by writing the processed A/D data into the motor actual position register (suggested M-Variable Mxx62).

Note

- If the automatic ADC read method is being used, waiting a delay of about ½ second after the PMAC boots should be allowed for processing the data before copying it into the motor actual position register.
- If the manual ADC read method is being used, it is recommended to add this procedure at the end of the manual read PLC, or in a subsequent PLC with ½ sec delay to allow data processing.

Example: Reading Motor #1 position on power-up or reset, assuming that the automatic read function is used, and that M5061 is predefined to access ADC channel 1.

```
End Gat
Del Gat
Close

#define Mtr1ActPos     M162    ; Motor #1 Actual Position (suggested M-Variables units of 1/32*I108)
Mtr1ActPos->D:$8B
#define Ch1ADC         M5061   ; Channel #1 ADC

Open PLC 1 Clear
I5111=500*8388608/I10 While(I5111>0) EndW    ; ½ sec delay
Mtr1ActPos=Ch1ADC*32*I108                    ; Motor #1 Actual Position (scaled to motor counts)
Disable PLC 1                                ; Scan once on power-up or reset
Close
```

Setting Up the Analog Outputs (DACs)

WARNING

In bipolar mode, the DAC output is not at zero volts on power-up or reset. It is required to set the corresponding M-Variable (register) to 2047 for zero voltage output on power-up or reset.

The M-Variable pointers for the analog outputs (DACs) on the ACC-59E are mapped as follows:

DAC #	Address Location	12-bit Location
1	Base Address + $8	[11:0]
2	Base Address + $9	[11:0]
3	Base Address + $A	[11:0]
4	Base Address + $B	[11:0]

DAC #	Address Location	12-bit Location
5	Base Address + $8	[23:12]
6	Base Address + $9	[23:12]
7	Base Address + $A	[23:12]
8	Base Address + $B	[23:12]

Example: Suggested User M-Variables for DAC Output on Two ACC-59Es:

1st ACC-59E (Base Address $78C00)	2nd ACC-59E (Base Address $79C00)
M6001->Y:$78C08,0,12 ; DAC Channel #1	M6009->Y:$79C08,0,12 ; DAC Channel #1
M6002->Y:$78C09,0,12 ; DAC Channel #2	M6010->Y:$79C09,0,12 ; DAC Channel #2
M6003->Y:$78C0A,0,12 ; DAC Channel #3	M6011->Y:$79C0A,0,12 ; DAC Channel #3
M6004->Y:$78C0B,0,12 ; DAC Channel #4	M6012->Y:$79C0B,0,12 ; DAC Channel #4
M6005->Y:$78C08,12,12 ; DAC Channel #5	M6013->Y:$79C08,12,12 ; DAC Channel #5
M6006->Y:$78C09,12,12 ; DAC Channel #6	M6014->Y:$79C09,12,12 ; DAC Channel #6
M6007->Y:$78C0A,12,12 ; DAC Channel #7	M6015->Y:$79C0A,12,12 ; DAC Channel #7
M6008->Y:$78C0B,12,12 ; DAC Channel #8	M6016->Y:$79C0B,12,12 ; DAC Channel #8

Testing the Analog Outputs

The Analog Outputs out of the ACC-59E can be wired as single ended (DAC+ & Ground) or differential (DAC+ & DAC-) signals.

Writing the software counts shown below to the corresponding M-Variables should result in the following voltages:

DAC Output	Software Counts (Write)	Single-Ended [V] DAC+ ←→ AGND	Differential [V] DAC+ ←→ DAC-
Unipolar Mode	0	0	0
	2047	5	10
	4095	10	20
Bipolar Mode	0	-10	-20
	2047	0	0
	4095	10	20

USING ACC-59E WITH POWER UMAC

Setting Up the Analog Inputs (ADCs)

The A/D converter chips used on the ACC-59E multiplex the resulting data, and therefore it is mandatory to read each input one at a time. With Power UMAC, this can be done in two ways:

- Automatic Read
- Manual (semi-automatic) Read

One should use the automatic read method when one wants to use the ADCs for servo feedback. One should use the manual read method when one needs to read ADCs from more than two ACC-36E or ACC-59E cards or when one wants the timing of the ADC conversion to occur exactly when specified, rather than at an automatic time that the user does not control.

With each method, either the PMAC Script or the C programming language can be used.

Automatic ADC Read

The automatic read function demultiplexes the data into individual registers and stores them once every ***AdcDemux.Enable*** phase cycles in accessible memory locations. It can handle up to a total of 32 ADC channels (16 pairs). There are 16 ADCs (8 pairs) per base address (or ACC-59E card) for a maximum of 2 base addresses (or two ACC-59E cards) as follows:

Pair #	1st ACC-59E		
1	ADC#1	&	ADC#9
2	ADC#2	&	ADC#10
3	ADC#3	&	ADC#11
4	ADC#4	&	ADC#12
5	ADC#5	&	ADC#13
6	ADC#6	&	ADC#14
7	ADC#7	&	ADC#15
8	ADC#8	&	ADC#16

Pair #	2nd ACC-59E		
9	ADC#1	&	ADC#9
10	ADC#2	&	ADC#10
11	ADC#3	&	ADC#11
12	ADC#4	&	ADC#12
13	ADC#5	&	ADC#13
14	ADC#6	&	ADC#14
15	ADC#7	&	ADC#15
16	ADC#8	&	ADC#16

The ACC-59E has only 8 ADC channels, which is less than what the automatic read function can handle. Thus, one configures the software to explicitly sample actually 16 ADCs (8 pairs) per card. However, in reality, the upper 8 ADCs are ignored (i.e., ADC#9 thru ADC#16), and only the lower 8 are used.

These are the necessary steps for setting up the automatic read function:

1. **Configure A/D Ring Pointers**: ***AdcDemux.Address[i]***.
   For each pair of index ***i***, ***AdcDemux.Address[i]*** must be set to the Power PMAC I/O base offset. Typically, pairs 1 – 8 are assigned to the first ACC-59E, pairs 9-16 are assigned to the second ACC-59E.

Ring Pointers	Pair #
AdcDemux.Address[0]	1
AdcDemux.Address[1]	2
AdcDemux.Address[2]	3
AdcDemux.Address[3]	4
AdcDemux.Address[4]	5
AdcDemux.Address[5]	6
AdcDemux.Address[6]	7
AdcDemux.Address[7]	8

Ring Pointers	Pair #
AdcDemux.Address[8]	9
AdcDemux.Address[9]	10
AdcDemux.Address[10]	11
AdcDemux.Address[11]	12
AdcDemux.Address[12]	13
AdcDemux.Address[13]	14
AdcDemux.Address[14]	15
AdcDemux.Address[15]	16

2. **Configure A/D Ring Convert Codes**: ***AdcDemux.ConvertCode[i]***.
   The convert code allows the user to select the input mode, whether unipolar or bipolar, as well as the ADC channel number (ADC#) to sample.

Setup Structure	Definition of k	Mode	Voltages
AdcDemux.ConvertCode[i]=$k00	k = ADC# - 1	Unipolar	Only Positive
	k = ADC# + 7	Bipolar	Positive/Negative

3. **Configure A/D Processing Ring Size**: ***AdcDemux.Enable***.
   AdcDemux.Enable = Number of ADC Pairs to Demux

Note

Setting ***AdcDemux.Enable*** to a value greater than zero activates the automatic ADC read ring.

4. **Access the A/D Results:** The results are stored in the ***AdcDemux.ResultLow[i]*** and ***AdcDemux.ResultHigh[i]*** structures:

1st ACC-59E		
i	***ResultLow[i]***	***ResultHigh[i]***
0	ADC#1 Result	ADC#9 Result
1	ADC#2 Result	ADC#10 Result
2	ADC#3 Result	ADC#11 Result
3	ADC#4 Result	ADC#12 Result
4	ADC#5 Result	ADC#13 Result
5	ADC#6 Result	ADC#14 Result
6	ADC#7 Result	ADC#15 Result
7	ADC#8 Result	ADC#16 Result

2nd ACC-59E		
i	***ResultLow[i]***	***ResultHigh[i]***
8	ADC#1 Result	ADC#9 Result
9	ADC#2 Result	ADC#10 Result
10	ADC#3 Result	ADC#11 Result
11	ADC#4 Result	ADC#12 Result
12	ADC#5 Result	ADC#13 Result
13	ADC#6 Result	ADC#14 Result
14	ADC#7 Result	ADC#15 Result
15	ADC#8 Result	ADC#16 Result

Note

- Since the ACC-59E carries only 8 channels per card, only ***AdcDemux.ResultLow[i]*** will contain meaningful information; ***AdcDemux.ResultHigh[i]*** can be ignored.
- Each ADC pair is automatically updated every ***AdcDemux.Enable*** phase cycles.

ADC Automatic Read Example 1

Setting up Power UMAC with an ACC-59E at I/O base address offset $A00000 to automatically read all 8 ADCs, allowing the user to choose unipolar or bipolar mode:

1. A/D Ring Pointers

```
AdcDemux.Address[0] = $A00000;        // ADC Pair #1
AdcDemux.Address[1] = $A00000;        // ADC Pair #2
AdcDemux.Address[2] = $A00000;        // ADC Pair #3
AdcDemux.Address[3] = $A00000;        // ADC Pair #4
AdcDemux.Address[4] = $A00000;        // ADC Pair #5
AdcDemux.Address[5] = $A00000;        // ADC Pair #6
AdcDemux.Address[6] = $A00000;        // ADC Pair #7
AdcDemux.Address[7] = $A00000;        // ADC Pair #8
```

2. A/D Convert Codes

Pair #	Unipolar	Bipolar
1	`AdcDemux.ConvertCode[0]=$000; // ADC#1`	`AdcDemux.ConvertCode[0]=$800; // ADC#1`
2	`AdcDemux.ConvertCode[1]=$100; // ADC#2`	`AdcDemux.ConvertCode[1]=$900; // ADC#2`
3	`AdcDemux.ConvertCode[2]=$200; // ADC#3`	`AdcDemux.ConvertCode[2]=$A00; // ADC#3`
4	`AdcDemux.ConvertCode[3]=$300; // ADC#4`	`AdcDemux.ConvertCode[3]=$B00; // ADC#4`
5	`AdcDemux.ConvertCode[4]=$400; // ADC#5`	`AdcDemux.ConvertCode[4]=$C00; // ADC#5`
6	`AdcDemux.ConvertCode[5]=$500; // ADC#6`	`AdcDemux.ConvertCode[5]=$D00; // ADC#6`
7	`AdcDemux.ConvertCode[6]=$600; // ADC#7`	`AdcDemux.ConvertCode[6]=$E00; // ADC#7`
8	`AdcDemux.ConvertCode[7]=$700; // ADC#8`	`AdcDemux.ConvertCode[7]=$F00; // ADC#8`

Note

These convert codes are applicable to ADCs #9 through #16 as well by definition, but knowing that the ACC-59E carries only 8 channels, they can be ignored.

3. A/D Processing Ring Size

```
AdcDemux.Enable = 8;     // Demux 8 ADC pairs
```

4. The resulting data is found in *AdcDemux.ResultLow[i]*

ADC Automatic Read Example 2

Another ACC-59E has been added to produce a total of two cards set at base offsets $A00000 and $B00000, respectively.

1. A/D Processing Ring Size

```
AdcDemux.Enable = 16;                // Demux 16 ADC pairs
```

2. A/D Ring Pointers

```
AdcDemux.Address[0]  = $A00000;      // ADC Pair #1
AdcDemux.Address[1]  = $A00000;      // ADC Pair #2
AdcDemux.Address[2]  = $A00000;      // ADC Pair #3
AdcDemux.Address[3]  = $A00000;      // ADC Pair #4
AdcDemux.Address[4]  = $A00000;      // ADC Pair #5
AdcDemux.Address[5]  = $A00000;      // ADC Pair #6
AdcDemux.Address[6]  = $A00000;      // ADC Pair #7
AdcDemux.Address[7]  = $A00000;      // ADC Pair #8

AdcDemux.Address[8]  = $B00000;      // ADC Pair #9
AdcDemux.Address[9]  = $B00000;      // ADC Pair #10
AdcDemux.Address[10] = $B00000;      // ADC Pair #11
AdcDemux.Address[11] = $B00000;      // ADC Pair #12
AdcDemux.Address[12] = $B00000;      // ADC Pair #13
AdcDemux.Address[13] = $B00000;      // ADC Pair #14
AdcDemux.Address[14] = $B00000;      // ADC Pair #15
AdcDemux.Address[15] = $B00000;      // ADC Pair #16
```

3. A/D Convert Codes

	Pair#	Unipolar	Bipolar
1st ACC-59E	1	`AdcDemux.ConvertCode[0] =$000; // ADC#1`	`AdcDemux.ConvertCode[0] =$800; // ADC#1`
	2	`AdcDemux.ConvertCode[1] =$100; // ADC#2`	`AdcDemux.ConvertCode[1] =$900; // ADC#2`
	3	`AdcDemux.ConvertCode[2] =$200; // ADC#3`	`AdcDemux.ConvertCode[2] =$A00; // ADC#3`
	4	`AdcDemux.ConvertCode[3] =$300; // ADC#4`	`AdcDemux.ConvertCode[3] =$B00; // ADC#4`
	5	`AdcDemux.ConvertCode[4] =$400; // ADC#5`	`AdcDemux.ConvertCode[4] =$C00; // ADC#5`
	6	`AdcDemux.ConvertCode[5] =$500; // ADC#6`	`AdcDemux.ConvertCode[5] =$D00; // ADC#6`
	7	`AdcDemux.ConvertCode[6] =$600; // ADC#7`	`AdcDemux.ConvertCode[6] =$E00; // ADC#7`
	8	`AdcDemux.ConvertCode[7] =$700; // ADC#8`	`AdcDemux.ConvertCode[7] =$F00; // ADC#8`
2nd ACC-59E	9	`AdcDemux.ConvertCode[8] =$000; // ADC#9`	`AdcDemux.ConvertCode[8] =$800; // ADC#9`
	10	`AdcDemux.ConvertCode[9] =$100; // ADC#10`	`AdcDemux.ConvertCode[9] =$900; // ADC#10`
	11	`AdcDemux.ConvertCode[10]=$200; // ADC#11`	`AdcDemux.ConvertCode[10]=$A00; // ADC#11`
	12	`AdcDemux.ConvertCode[11]=$300; // ADC#12`	`AdcDemux.ConvertCode[11]=$B00; // ADC#12`
	13	`AdcDemux.ConvertCode[12]=$400; // ADC#13`	`AdcDemux.ConvertCode[12]=$C00; // ADC#13`
	14	`AdcDemux.ConvertCode[13]=$500; // ADC#14`	`AdcDemux.ConvertCode[13]=$D00; // ADC#14`
	15	`AdcDemux.ConvertCode[14]=$600; // ADC#15`	`AdcDemux.ConvertCode[14]=$E00; // ADC#15`
	16	`AdcDemux.ConvertCode[15]=$700; // ADC#16`	`AdcDemux.ConvertCode[15]=$F00; // ADC#16`

4. The resulting data is found in *AdcDemux.ResultLow[i]*

Accessing AdcDemux Structures in C Code (Optional; For C Programmers)

In order to use ***AdcDemux*** structures in C code, use the pointer to shared-memory data structure (pshm).

For example, to read ***AdcDemux.ResultLow[0]***, make sure that ***#include <RtGpShm.h>*** is in the header, and then access ***AdcDemux.ResultLow[0]*** by using ***pshm->AdcDemux.ResultLow[0]*** as an expression.

See the following example.

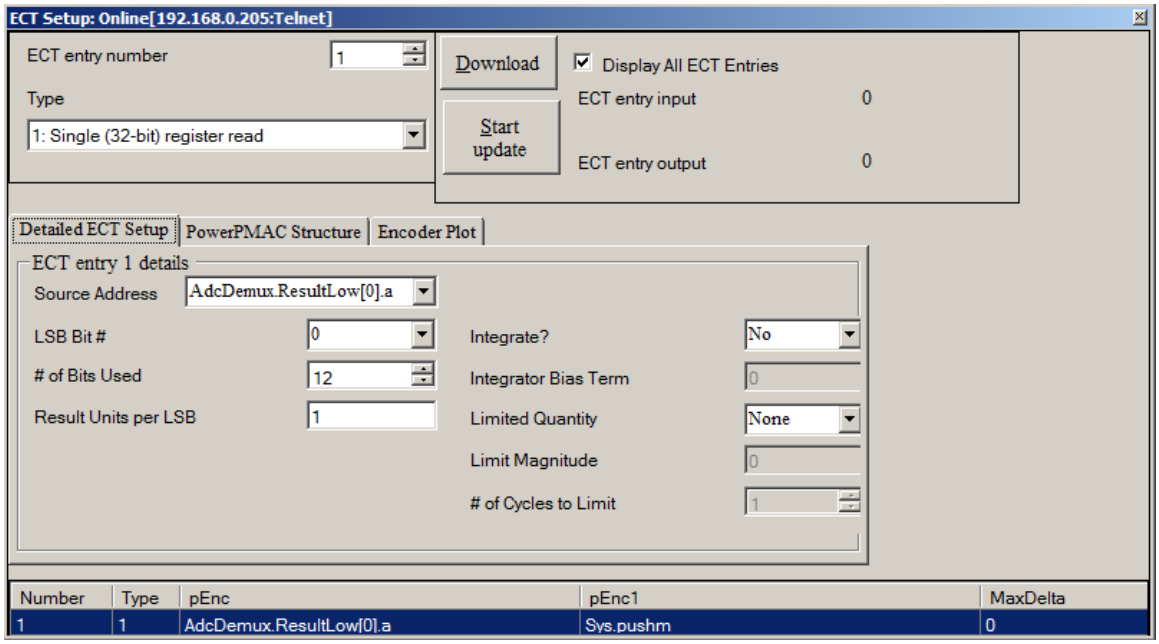
Example: Using Automatic ADC Result Structures in a CPLC

```
#include <RtGpShm.h>
#include <stdio.h>
#include <dlfcn.h>

void user_plcc()
{
        /*** ADC Result Array Format ***\
        ADC_Result[0]  stores ADC Channel 1  result
        ADC_Result[1]  stores ADC Channel 2  result
        ADC_Result[2]  stores ADC Channel 3  result
        ADC_Result[3]  stores ADC Channel 4  result
        ADC_Result[4]  stores ADC Channel 5  result
        ADC_Result[5]  stores ADC Channel 6  result
        ADC_Result[6]  stores ADC Channel 7  result
        ADC_Result[7]  stores ADC Channel 8  result*/
        unsigned int index;
        // ADC Result Storage Array
        int ADC_Result[8];
        // Access ADC results one by one in a for loop
        for(index = 0; index < 8; index++)
        {
                // Store ADC Result in array element
                ADC_Result[index] = pshm->AdcDemux.ResultLow[index];
                /******Insert user code here, where ADC_Result elements
                can be used for the user's calculations*************/
        }
        return;
}
```

Using an Analog Input for Servo Feedback

The ACC-59E analog inputs can be used as a feedback device for a servo motor. For simplicity, the automatic ADC read function is recommended for this application. This example assumes that the automatic ADC read function has already been configured and activated.

Example: Setting up Motor #1 with position and velocity feedback from ADC channel 1. The analog input is brought into the Encoder Conversion Table (ECT) as a single read of a 32-bit register (in the Power PMAC IDE Software: Delta Tau → Configure → Encoder Conversion Table):



The equivalent code in Power PMAC Structures:

```
EncTable[1].type = 1;          // Set entry type to 32-bit word read
EncTable[1].pEnc = AdcDemux.ResultLow[0].a;  // Set encoder address to ADC channel 1
EncTable[1].pEnc1 = Sys.pushm;               // Unused; set to Sys.pushm
EncTable[1].index1 = 20;              // Shift left 20 to put source MSB in bit 31
EncTable[1].index2 = 0;               // Shift left 0 to put source LSB into bit 0
EncTable[1].index3 = 0;               // No limit on first derivative
EncTable[1].index4 = 0;               // No limit on second derivative
EncTable[1].ScaleFactor = 1/pow(2,EncTable[1].index1);      // Scale factor (1/(2^20))
```

The position and velocity pointers are then set to the processed data address:

```
Motor[1].pEnc = EncTable[1].a;        // Outer (position) loop source address
Motor[1].pEnc2 = EncTable[1].a;       // Inner (velocity) loop source address
```

One can then adjust ***Motor[1].PosSf*** and ***Motor[1].Pos2Sf***, the position and velocity scale factors, respectively, if necessary.

Note

Using ***AdcDemux*** structures in the ECT requires firmware version 1.2.1.116 or newer.

Analog Input Power-On Position

Some analog devices are absolute along the travel range of the motor (e.g., in hydraulic piston applications). Generally, it is desirable to obtain the motor position (input voltage) on power up or reset. The following example code will configure Motor #1 to use ADC channel 1 for the power-on position read as an unsigned signal with a scale factor of 1:

```
Motor[1].pAbsPos = AdcDemux.ResultLow[0].a;   // Set position register to ADC channel 1
Motor[1].AbsPosFormat = $00000C00;            // Use 12 bits starting at bit 0, unsigned
                                              // (for signed, change to $01000C00)
Motor[1].AbsPosSF = 1;                        // Scale factor of 1
```

Please refer to the description of ***Motor[n].AbsPosFormat*** in the Power PMAC Saved Data Structure Elements manual for further details.

Manual ADC Read Using ACC-59E Structures

The manual read method with ACC-59E structures consists of selecting the desired channel with the ***ACC59E[n].ConvertCode*** structure, reading it with the ***ACC59E[n].ADCu*** or ***ACC59E[n].ADCs*** structure, and then storing it into local memory, where ***n*** is the index specified by the DIP switch SW1 setting of the card.

Note

Refer to the Addressing ACC-59E section of this manual for a list of values that the index ***n*** assumes.

This procedure can be implemented in a PLC script to read all channels consecutively and consistently, creating a "custom automatic" function.

Following are the necessary steps for implementing the manual ADC read method with structures to allow the user to choose to read unipolar or bipolar input modes:

1. Using the ***ACC59E[n].ConvertCode*** structure, specify the **voltage mode** for each ADC# desired:

   Set ***ACC59E[n].ConvertCode*** = ADC# - 1 for Unipolar Inputs
   Set ***ACC59E[n].ConvertCode*** = ADC# + 7 for Bipolar Inputs

	Channel Select Pointer Value	
ADC#	**Unipolar Inputs**	**Bipolar Inputs**
1	0	8
2	1	9
3	2	10
4	3	11
5	4	12
6	5	13
7	6	14
8	7	15

2. Poll the ADC ready bit, ***ACC59E[n].ADCRdy,*** in a ***while*** loop until it becomes 1, and then read and/or copy the resulting data using the following structures:

   ACC59E[n].ADCu for unipolar signals
   ACC59E[n].ADCs for bipolar signals

ADC Manual Read Example Script PLCs

To set up Power UMAC with an ACC-59E at I/O base address offset $A00000, see the following examples.

Unipolar Script PLC Example

This example selects and reads channels 1 through 8 successively as unipolar in a PLC and stores the results for channels 1 through 8 in global variables ADC1 through ADC8.

```
global ADC1;           // Channel 1 ADC storage variable
global ADC2;           // Channel 2 ADC storage variable
global ADC3;           // Channel 3 ADC storage variable
global ADC4;           // Channel 4 ADC storage variable
global ADC5;           // Channel 5 ADC storage variable
global ADC6;           // Channel 6 ADC storage variable
global ADC7;           // Channel 7 ADC storage variable
global ADC8;           // Channel 8 ADC storage variable

Open PLC 1
ACC59E[0].ConvertCode = 0;           // Select Channel 1
While(ACC59E[0].ADCRdy != 1){}       // Wait for ADC conversion to finish
ADC1 = ACC59E[0].ADCu;               // Read and copy result into storage

ACC59E[0].ConvertCode = 1;           // Select Channel 2
While(ACC59E[0].ADCRdy != 1){}       // Wait for ADC conversion to finish
ADC2 = ACC59E[0].ADCu;               // Read and copy result into storage

ACC59E[0].ConvertCode = 2;           // Select Channel 3
While(ACC59E[0].ADCRdy != 1){}       // Wait for ADC conversion to finish
ADC3 = ACC59E[0].ADCu;               // Read and copy result into storage

ACC59E[0].ConvertCode = 3;           // Select Channel 4
While(ACC59E[0].ADCRdy != 1){}       // Wait for ADC conversion to finish
ADC4 = ACC59E[0].ADCu;               // Read and copy result into storage

ACC59E[0].ConvertCode = 4;           // Select Channel
While(ACC59E[0].ADCRdy != 1){}       // Wait for ADC conversion to finish
ADC5 = ACC59E[0].ADCu;               // Read and copy result into storage

ACC59E[0].ConvertCode = 5;           // Select Channel 6
While(ACC59E[0].ADCRdy != 1){}       // Wait for ADC conversion to finish
ADC6 = ACC59E[0].ADCu;               // Read and copy result into storage

ACC59E[0].ConvertCode = 6;           // Select Channel 7
While(ACC59E[0].ADCRdy != 1){}       // Wait for ADC conversion to finish
ADC7 = ACC59E[0].ADCu;               // Read and copy result into storage

ACC59E[0].ConvertCode = 7;           // Select Channel 8
While(ACC59E[0].ADCRdy != 1){}       // Wait for ADC conversion to finish
ADC8 = ACC59E[0].ADCu;               // Read and copy result into storage
Close
```

Bipolar Script PLC Example

This example selects and reads channels 1 through 8 successively as bipolar in a PLC and stores the results for channels 1 through 8 in global variables.

```
global ADC1;           // Channel 1 ADC storage variable
global ADC2;           // Channel 2 ADC storage variable
global ADC3;           // Channel 3 ADC storage variable
global ADC4;           // Channel 4 ADC storage variable
global ADC5;           // Channel 5 ADC storage variable
global ADC6;           // Channel 6 ADC storage variable
global ADC7;           // Channel 7 ADC storage variable
global ADC8;           // Channel 8 ADC storage variable

Open PLC 1
ACC59E[0].ConvertCode = 8;           // Select ADC Channel 1
While(ACC59E[0].ADCRdy != 1){}       // Wait for ADC conversion to finish
ADC1 = ACC59E[0].ADCs;               // Read and copy result into storage

ACC59E[0].ConvertCode = 9;           // Select ADC Channel 2
While(ACC59E[0].ADCRdy != 1){}       // Wait for ADC conversion to finish
ADC2 = ACC59E[0].ADCs;               // Read and copy result into storage

ACC59E[0].ConvertCode = 10;          // Select ADC Channel 3
While(ACC59E[0].ADCRdy != 1){}       // Wait for ADC conversion to finish
ADC3 = ACC59E[0].ADCs;               // Read and copy result into storage

ACC59E[0].ConvertCode = 11;          // Select ADC Channel 4
While(ACC59E[0].ADCRdy != 1){}       // Wait for ADC conversion to finish
ADC4 = ACC59E[0].ADCs;               // Read and copy result into storage

ACC59E[0].ConvertCode = 12;          // Select ADC Channel 5
While(ACC59E[0].ADCRdy != 1){}       // Wait for ADC conversion to finish
ADC5 = ACC59E[0].ADCs;               // Read and copy result into storage

ACC59E[0].ConvertCode = 13;          // Select ADC Channel 6
While(ACC59E[0].ADCRdy != 1){}       // Wait for ADC conversion to finish
ADC6 = ACC59E[0].ADCs;               // Read and copy result into storage

ACC59E[0].ConvertCode = 14;          // Select ADC Channel 7
While(ACC59E[0].ADCRdy != 1){}       // Wait for ADC conversion to finish
ADC7 = ACC59E[0].ADCs;               // Read and copy result into storage

ACC59E[0].ConvertCode = 15;          // Select ADC Channel 8
While(ACC59E[0].ADCRdy != 1){}       // Wait for ADC conversion to finish
ADC8 = ACC59E[0].ADCs;               // Read and copy result into storage
Close
```

Using ACC-59E Power PMAC Structures in C (Optional; For C Programmers)

To Read

To read ACC-59E structures in C, use the ***GetPmacVar()*** function, which has the following definition:

int GetPmacVar (char *pinstr, double *pdata)

Thus, the first argument is a `char` array containing the string one desires to write to PMAC and the second argument is a `double*` containing the address of the variable in which the user wants to store the response from PMAC. For example, to read ***ACC59E[0].ADCu,*** use

GetPmacVar("ACC59E[0].ADCu",&ADCu)

to store the result in ADCu, where ***ADCu*** is a `double` used to store the result.

To Write

To write to ACC-59E structures in C, use the ***SetPmacVar()*** function, which has the following definition:

int SetPmacVar (char *pinstr, double data)

Thus, the first argument is a `char` array containing the string one desires to write to PMAC and the second argument is a `double` containing the value the user wants to write to the PMAC. For example, to write to ***ACC59E[0].ConvertCode***, use

SetPmacVar("ACC59E[0].ConvertCode", ConvertCode)

to write the `double` ***ConvertCode*** to the ***ACC59E[0].ConvertCode*** structure.

Note

One can use ***sprintf()*** to form the string in runtime, if needed. Otherwise, one can just use a string literal. ***sprintf()*** is a standard GNU C function in the ***stdio.h*** header file. Please see a GNU C reference manual for additional details.

See the following example, which uses ***GetPmacVar()***, ***SetPmacVar()***, and ***sprintf()***.

WARNING

When using C routines, failure to release control of the loop waiting for the ADC conversions to finish may cause PMAC to lock up, possibly creating a runaway condition. Thus, one of the following functions, ***WaitForADC***, has precautions in order to release control of the wait loop in the event that the ADC conversion bits never become 1.

Example: Using ADC Result Structures in a CPLC

Configuring two ACC-59E cards: 1st card has unipolar inputs and is set at base offset $A00000, and 2nd card has bipolar inputs set at base offset $B00000.

```
#include <gplib.h>
#include <stdio.h>
#include <dlfcn.h>

// Definition(s)
#define Card1Number           0       // For Base Offset $A00000
#define Card2Number           1       // For Base Offset $B00000

// Prototype(s)
int WaitForADC(unsigned int Card_Index);

void user_plcc()
{
        // ADC Result Storage Arrays
        unsigned int ADC_Result_Unipolar[8],index;
        int ADC_Result_Bipolar[8],waitResult = 0;
        char buffer[64]="";
        double temp = 0, ConvertCode = 0;
        // Access ADC results one by one in a for loop
        for(index = 0; index < 8; index++)
        {
                // Prepare string
                sprintf(buffer,"ACC59E[%d].ConvertCode",Card1Number);
                ConvertCode = index;                          // Compute convert code
                // Write Unipolar convert code to Channel Select structure
                SetPmacVar(buffer,ConvertCode);
                // Poll conversion complete bits until the conversions are finished
                waitResult = WaitForADC(Card1Number);
                if(waitResult < 0)
                {
                        return; // Conversion was not successful
                }
                // Prepare string
                sprintf(buffer,"ACC59E[%d].ADCu",Card1Number);
                GetPmacVar(buffer,&temp);                     // Read the structure
                // Cast and store low unipolar (unsigned) ADC Result in array element
                ADC_Result_Unipolar[index] = (unsigned int)temp;
                // Prepare string
                sprintf(buffer,"ACC59E[%d].ConvertCode",Card2Number);
                ConvertCode = index + 8;                      // Compute convert code
                // Write bipolar convert code to Channel Select structure
                SetPmacVar(buffer,ConvertCode);
                // Poll conversion complete bits until the conversions are finished
                waitResult = WaitForADC(Card2Number);
                if(waitResult < 0)
                {
                        return; // Conversion was not successful
                }
                // Prepare string
                sprintf(buffer,"ACC59E[%d].ADCs",Card2Number);

                GetPmacVar(buffer,&temp);                     // Read the structure
                // Cast and store low bipolar (signed) ADC Result in array element
                ADC_Result_Bipolar[index] = (int)temp;
                /******Insert user code here, where ADC_Result_Unipolar elements
                and ADC_Result_Bipolar elements
                can be used for the user's calculations*************/
        }
        return;
}
```

```
int WaitForADC(unsigned int Card_Index)
{       // Waits until ADC conversions have completed
        // Inputs:
        // Card_Index: index (n) from POWER section of Addressing ACC-59E table
        // Outputs:
        // return 0 if successfully performed ADC conversion
        // return -1 if conversion did not complete within Timeout ms
        unsigned int Rdy = 0,iterations = 0;
        double Present_Time,Conversion_Start_Time,Time_Difference,Timeout,Timeout_us,temp = 0;
        char str1[24]="";
        struct timespec SleepTime={0};
        SleepTime.tv_nsec=1000000;

        // Get time at (almost) start of conversion (microseconds)
        Conversion_Start_Time = GetCPUClock();
        // Timeout: Maximum permitted time to wait for ADC conversion to finish before
        // error (milliseconds)
        Timeout = 500; // Milliseconds
        Timeout_us = Timeout*1000;                    // Convert to microseconds
        sprintf(str1,"ACC59E[%u].ADCRdy",Card_Index);// Prepare string
        do
        {
                // If the loop has taken a multiple of 50 iterations to finish
                if(iterations == 50){
                        // Release control for 1 ms so PMAC does not go into Watchdog mode
                        // while waiting for conversion to finish
                        nanosleep(&SleepTime,NULL); // Release thread and wait 1 msec
                        iterations = 0;          // Reset iteration counter
                }
                Present_Time = GetCPUClock();   // Obtain current system time
                // Compute difference in time between starting conversion and now
                Time_Difference = Present_Time-Conversion_Start_Time;
                if(Time_Difference > Timeout_us)// If more than Timeout ms have elapsed
                {
                        return (-1);             // Return with error code
                }
                GetPmacVar(str1,&temp);          // Read the ADCRdyLow structure
                Rdy = (unsigned int)temp;        // Cast and store the result
                iterations++;
        }       while(Rdy != 1);                 // Test ADC ready bit
        return 0;                                // Return with success code
}
```

Testing the Analog Inputs

The Analog Inputs can be brought into the ACC-59E as single ended (ADC+ & Ground) or differential (ADC+ & ADC-) signals.

Note

In single-ended mode, ADC- should to be tied to analog ground for full resolution and proper operation.

Reading the input signals in "software counts" should show the following:

ADC Input	Single-Ended [V] ADC+ ←→ AGND	Differential [V] ADC+ ←→ ADC-	Software Counts
Unipolar Mode	0	0	0
	10	10	2047
	20	20	4095
Bipolar Mode	-10	-10	-2047
	0	0	0
	10	10	2047

Setting Up the Analog Outputs (DACs)

WARNING

In bipolar mode, the DAC output is not at zero volts on power-up or reset. It is required to set the corresponding M-Variable (register) to 2047 for zero voltage output on power-up or reset.

Testing the Analog Outputs

The Analog Outputs out of the ACC-59E can be wired as single ended (DAC+ & Ground) or differential (DAC+ & DAC-) signals. Regardless of whether one is using Script I/O pointers, C pointers, or ***ACC59E[n].DAC[i]*** structures, writing the following "software counts" to the corresponding DAC channel registers should result in the following output voltages:

DAC Output	Software Counts (Write)	Single-Ended [V] DAC+ ←→ AGND	Differential [V] DAC+ ←→ DAC-
Unipolar Mode	0	0	0
	2047	5	10
	4095	10	20
Bipolar Mode	0	-10	-20
	2047	0	0
	4095	10	20

Using the Analog Outputs with ACC-59E Power PMAC Structures

One can use the ***ACC59E[n].DAC[i]*** structure to write the aforementioned software count values to the appropriate locations, where ***n*** is the card index as described in the Addressing ACC-59E section, and ***i*** is (DAC# - 1), as shown in the following table:

DAC Channel (DAC#)	Structure to Which to Write
1	***ACC59E[n].DAC[0]***
2	***ACC59E[n].DAC[1]***
3	***ACC59E[n].DAC[2]***
4	***ACC59E[n].DAC[3]***
5	***ACC59E[n].DAC[4]***
6	***ACC59E[n].DAC[5]***
7	***ACC59E[n].DAC[6]***
8	***ACC59E[n].DAC[7]***

Example: Using ACC-59E DAC Structures in Global Definitions.pmh

Configuring a single ACC-59E set at base offset $A00000 with unipolar differential outputs.

```
// Assumes unipolar differential outputs
ACC59E[0].DAC[0] = 0;          // DAC Channel 1 Outputs 0.0 V
ACC59E[0].DAC[1] = 511;        // DAC Channel 2 Outputs 2.5 V
ACC59E[0].DAC[2] = 1023;       // DAC Channel 3 Outputs 5.0 V
ACC59E[0].DAC[3] = 1535;       // DAC Channel 4 Outputs 7.5 V
ACC59E[0].DAC[4] = 2047;       // DAC Channel 5 Outputs 10.0 V
ACC59E[0].DAC[5] = 2559;       // DAC Channel 6 Outputs 12.5 V
ACC59E[0].DAC[6] = 3071;       // DAC Channel 7 Outputs 15.0 V
ACC59E[0].DAC[7] = 4095;       // DAC Channel 8 Outputs 20.0 V
```

Using DAC Structures in C (Optional; For C Programmers)

To use ACC-59E structures in C, one can use ***SetPmacVar("ACC59E[n].DAC[i]", DAC_Value)***, where ***DAC_Value*** is a `double` containing the software counts one desired to write to the ***ACC59E[n].DAC[i]*** structure. See the following example.

Example: DAC Output CPLC with ACC-59E Structures

Configuring a single ACC-59E set at base offset $A00000 to output 5.0 volts on all 8 DAC channels with unipolar differential wiring on all 8 DAC channels. The CPLC disables itself after running once.

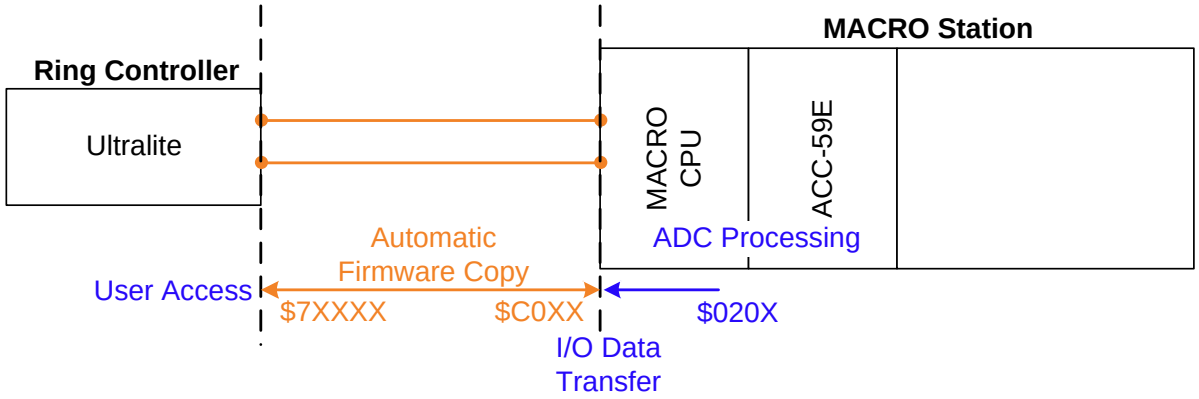
```
#include <gplib.h>
#include <stdio.h>
#include <dlfcn.h>

// Definition(s)
#define CardNumber            0       // For Base Offset $A00000
#define CPLC_Number           1       // User can adjust this number to match his CPLC number
#define Single_Ended_Wiring   0
#define Differential_Wiring   1
#define Unipolar_Mode         0
#define Bipolar_Mode          1
#define Output_Voltage        5.0     // Volts

//Prototype(s)
int GetSoftwareCounts(double Desired_Output_Voltage,
                      int Polarity, int Wiring_Mode, unsigned int *SoftwareCounts);

void user_plcc()
{
        unsigned int index, Software_Counts = 0;
        int ErrorCode;
        char buffer[32]="";
        // Compute appropriate software counts value
        ErrorCode = GetSoftwareCounts(Output_Voltage, Unipolar_Mode, Differential_Wiring,
&Software_Counts);
        if(ErrorCode < 0)
        {
                return; // error
        }
        for(index = 0; index < 8; index++)
        {
                // Prepare string
                sprintf(buffer,"ACC59E[%u].DAC[%u]",CardNumber,index);
                // Cast to double and write to structure the amount in Software_Counts
                SetPmacVar(buffer,(double)Software_Counts);
}
        pshm->UserAlgo.BgCplc[CPLC_Number] = enum_threaddisable;    // Disable this CPLC (CPLC 1)
        return;
}

int GetSoftwareCounts(double Desired_Output_Voltage,
                      int Polarity, int Wiring_Mode, unsigned int *Software_Counts)
{
/* Returns the appropriate software counts to write to the DAC structure
Inputs:
DAC_Channel:                  Desired DAC channel to which to output (integers, 1 to 8)
Desired_Output_Voltage:       Desired voltage to output in units of Volts
Polarity:                             DAC output mode set by jumper J3 (0 = unipolar, 1 = bipolar)
Wiring_Mode:                  DAC output wiring mode (0 = single-ended, 1 = differential)
*Software_Counts:     Address of the software counts variable in which to store the resulting software
counts calculation

Outputs:
return 0 and write the correct word to write the correct software counts value to
                *SoftwareCounts, if everything went correctly
return -1 if Polarity invalid
return -2 if Wiring_Mode invalid
return -3 if Desired_Output_Voltage was invalid and was subsequently truncated,
                      but truncated value was still written to DAC channel
Notes:
This function will truncate the desired output voltage to correspond with
physical card limits.  */
```

```
        int Software_Count_Range = 0, Max_Amplitude = 0,
        Software_Count_Offset = 0, Return_Value = 0;
        double Conversion_Factor = 0;
        // Check whether inputs to function are correct
        if((Polarity != 0 && Polarity != 1))
        {
                return -1;
        }
        if((Wiring_Mode != 0 && Wiring_Mode != 1))
        {
                return -2;
        }
        if (Wiring_Mode == 0){                                  // Single-Ended
                Max_Amplitude = 10;                             // Volts
                if(Desired_Output_Voltage > 10)                 // Saturate voltages at card limits
                {
                        Desired_Output_Voltage = 10;
                        Return_Value = -3;
                }
        }
        else {                                                  // Differential
                Max_Amplitude = 20;                             // Volts
                if(Desired_Output_Voltage > 20)                 // Saturate voltages at card limits
                {
                        Desired_Output_Voltage = 20;
                        Return_Value = -3;
                }
        }
        if (Polarity == 0){                                     // Unipolar
                Software_Count_Range = 4095;
                Software_Count_Offset = 0;
                if(Desired_Output_Voltage < 0)                  // Saturate voltages at card limits
                {
                        Desired_Output_Voltage = 0;
                        Return_Value = -3;
                }
        }
        else{                                                   // Bipolar
                Software_Count_Range = 2047;
                Software_Count_Offset = 2047;
                if(Wiring_Mode == 0)
                {
                        if(Desired_Output_Voltage < (-10))      // Saturate voltages at card limits
                        {
                                Desired_Output_Voltage = (-10);
                                Return_Value = -3;
                        }
                } else {
                        if(Desired_Output_Voltage < (-20))      // Saturate voltages at card limits
                        {
                                Desired_Output_Voltage = (-20);
                                Return_Value = -3;
                        }
                }
        }
        // Conversion factor in units of counts/volt:
        Conversion_Factor = (double)((double)(Software_Count_Range)/(double)(Max_Amplitude));
        *Software_Counts = (unsigned int)((double)Software_Count_Offset +
                                                Desired_Output_Voltage*Conversion_Factor);
        return 0;
}
```

USING ACC-59E WITH TURBO UMAC MACRO

Setting up ACC-59E on a MACRO station requires the following steps:

- Establishing communication with the MACRO Station and enabling nodes
- Enabling ADC Processing (Automatic Read Function) at the MACRO Station
- Transferring Data over I/O Nodes



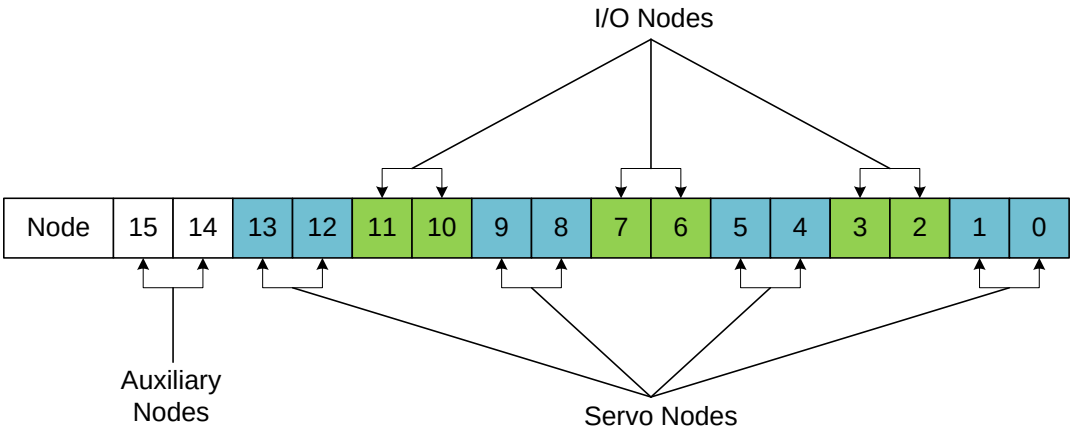
The goal is to allow the user "software" access to the analog inputs brought into the MACRO Station(s) from the Ring Controller (i.e. Turbo PMAC2 Ultralite, or UMAC with ACC-5E).

The automatic read function (ADC processing) demultiplexes the ADC data and puts it into predefined "local" registers ($02XX) at the MACRO Station side. The I/O node data transfer then copies the data from these registers ($02XX) into MACRO Station node registers ($C0XX), which are in turn automatically copied by the firmware (no user settings required) into their complement node registers ($78XXX) at the Ring Controller side.

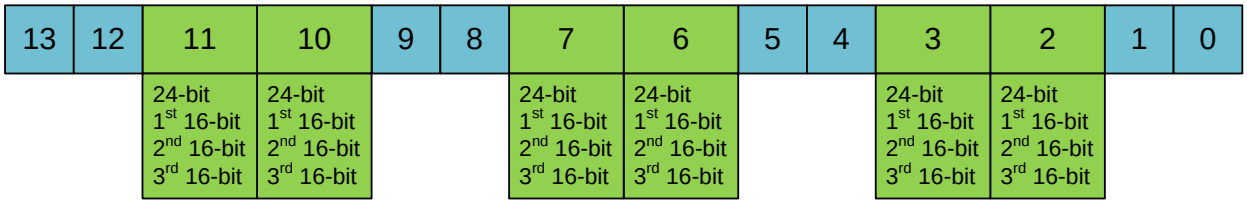
Quick Review: Nodes and Addressing

Each MACRO IC consists of 16 nodes: 2 auxiliary, 8 servo, and 6 I/O nodes.

- Auxiliary nodes are Master/Control registers and internal firmware use.
- Servo nodes carry information such as feedback, commands, and flags for motor control.
- I/O nodes are by default unoccupied and are user configurable for transferring miscellaneous data.



Each I/O node consists of 4 registers; one 24-bit and three 16-bit registers for a total of 72 bits of data.



A given MACRO Station can be populated with either a MACRO8 or MACRO16 CPU:

- MACRO8 supports only 1 MACRO IC (IC#0).
- MACRO16 supports 2 MACRO ICs (IC#0 and IC#1).

The I/O node addresses ($C0XX) for each of the Station MACRO ICs are:

	Station MACRO IC #0 Node Registers					
Node	**2**	**3**	**6**	**7**	**10**	**11**
24-bit	X:$C0A0	X:$C0A4	X:$C0A8	X:$C0AC	X:$C0B0	X:$C0B4
16-bit	X:$C0A1	X:$C0A5	X:$C0A9	X:$C0AD	X:$C0B1	X:$C0B5
16-bit	X:$C0A2	X:$C0A6	X:$C0AA	X:$C0AE	X:$C0B2	X:$C0B6
16-bit	X:$C0A3	X:$C0A7	X:$C0AB	X:$C0AF	X:$C0B3	X:$C0B7

	Station MACRO IC #1 Node Registers					
Node	**2**	**3**	**6**	**7**	**10**	**11**
24-bit	X:$C0E0	X:$C0E4	X:$C0E8	X:$C0EC	X:$C0F0	X:$C0F4
16-bit	X:$C0E1	X:$C0E5	X:$C0E9	X:$C0ED	X:$C0F1	X:$C0F5
16-bit	X:$C0E2	X:$C0E6	X:$C0EA	X:$C0EE	X:$C0F2	X:$C0F6
16-bit	X:$C0E3	X:$C0E7	X:$C0EB	X:$C0EF	X:$C0F3	X:$C0F7

Note

Non-Turbo PMAC2 Ultralite (legacy) I/O node addresses are the same as Station MACRO IC#0 node registers.

A given Turbo PMAC2 Ultralite (or UMAC with ACC-5E) can be populated with up to 4 MACRO ICs (IC#0, IC#1, IC#2, and IC#3) which can be queried with global variable I4902:

If I4902=	Populated MACRO IC #s
$0	None
$1	0
$3	0, 1
$7	0, 1, 2
$F	0, 1, 2, 3

And the I/O node addresses ($7XXXX) for each of the Ultralite MACRO ICs are:

	Ring Controller MACRO IC #0 Node Registers					
Station I/O Node#	**2**	**3**	**6**	**7**	**10**	**11**
Ultralite I/O Node#	**2**	**3**	**6**	**7**	**10**	**11**
24-bit	X:$78420	X:$78424	X:$78428	X:$7842C	X:$78430	X:$78434
16-bit	X:$78421	X:$78425	X:$78429	X:$7842D	X:$78431	X:$78435
16-bit	X:$78422	X:$78426	X:$7842A	X:$7842E	X:$78432	X:$78436
16-bit	X:$78423	X:$78427	X:$7842B	X:$7842F	X:$78433	X:$78437

	Ring Controller MACRO IC #1 Node Registers					
Station I/O Node#	**2**	**3**	**6**	**7**	**10**	**11**
Ultralite I/O Node#	**18**	**19**	**22**	**23**	**26**	**27**
24-bit	X:$79420	X:$79424	X:$79428	X:$7942C	X:$79430	X:$79434
16-bit	X:$79421	X:$79425	X:$79429	X:$7942D	X:$79431	X:$79435
16-bit	X:$79422	X:$79426	X:$7942A	X:$7942E	X:$79432	X:$79436
16-bit	X:$79423	X:$79427	X:$7942B	X:$7942F	X:$79433	X:$79437

	Ring Controller MACRO IC #2 Node Registers					
Station I/O Node#	**2**	**3**	**6**	**7**	**10**	**11**
Ultralite I/O Node#	**34**	**35**	**38**	**39**	**42**	**43**
24-bit	X:$7A420	X:$7A424	X:$7A428	X:$7A42C	X:$7A430	X:$7A434
16-bit	X:$7A421	X:$7A425	X:$7A429	X:$7A42D	X:$7A431	X:$7A435
16-bit	X:$7A422	X:$7A426	X:$7A42A	X:$7A42E	X:$7A432	X:$7A436
16-bit	X:$7A423	X:$7A427	X:$7A42B	X:$7A42F	X:$7A433	X:$7A437

	Ring Controller MACRO IC #3 Node Registers					
Station I/O Node#	**2**	**3**	**6**	**7**	**10**	**11**
Ultralite I/O Node#	**50**	**51**	**54**	**55**	**58**	**59**
24-bit	X:$7B420	X:$7B424	X:$7B428	X:$7B42C	X:$7B430	X:$7B434
16-bit	X:$7B421	X:$7B425	X:$7B429	X:$7B42D	X:$7B431	X:$7B435
16-bit	X:$7B422	X:$7B426	X:$7B42A	X:$7B42E	X:$7B432	X:$7B436
16-bit	X:$7B423	X:$7B427	X:$7B42B	X:$7B42F	X:$7B433	X:$7B437

Enabling MACRO Station ADC Processing

The A/D converter chips used on the ACC-59E multiplex the resulting data, and therefore it is mandatory to read each input one at a time. The only practical way to do this on a MACRO station is using the automatic "built-in" method. This method copies the ADC inputs from the ACC-59E to predefined MACRO station memory locations.

Note

The automatic ADC copy feature is available with:
MACRO8 CPU (602804) firmware version 1.15 or newer
MACRO16 CPU (603719) firmware version 1.202 or newer

For each MACRO IC, the automatic read function handles 16 ADC channels, to permit one to read a maximum of 32 ADC channels with two MACRO ICs. Hence, with MACRO8 CPUs, the maximum number of possible ADC automatic reads is 16. And with MACRO16 CPUs, the maximum number of possible ADC automatic reads is 32. There are 16 ADCs (8 pairs) per base address, i.e. per ACC-59E present, for a maximum of 2 base addresses with 2 ACC-59Es present.

Note

The firmware will process 16 ADCs per I/O card base address, but only ADCs 1-8 are physically present on each ACC-59E, making the total number of physical ADCs processed 16 with two ACC-59Es.

Pair #	1st ACC-59E		
1	ADC#1	&	ADC#9
2	ADC#2	&	ADC#10
3	ADC#3	&	ADC#11
4	ADC#4	&	ADC#12
5	ADC#5	&	ADC#13
6	ADC#6	&	ADC#14
7	ADC#7	&	ADC#15
8	ADC#8	&	ADC#16

Pair #	2nd ACC-59E		
9	ADC#1	&	ADC#9
10	ADC#2	&	ADC#10
11	ADC#3	&	ADC#11
12	ADC#4	&	ADC#12
13	ADC#5	&	ADC#13
14	ADC#6	&	ADC#14
15	ADC#7	&	ADC#15
16	ADC#8	&	ADC#16

Setting up the automatic read function (ADC processing) successfully requires the configuration of the following 3 MACRO Station I-Variables (MI variables):

- **MS{anynode},MI987: A/D Input Enable**

MI987 controls the ADC processing enable.
It can take one of the following settings: =0 disabled
=1 enabled

- **MS{anynode},MI988: A/D Unipolar/Bipolar Control**

MI988 specifies whether a given pair is setup for bipolar (±10 V) or unipolar (0 to +20 V) mode. It is an 8-bit word, with each bit strictly controlling a pair of ADCs: Bit value = 0 → Unipolar
Bit value = 1 → Bipolar

Bit#	Pair		
0	ADC1	&	ADC9
1	ADC2	&	ADC10
2	ADC3	&	ADC11
3	ADC4	&	ADC12
4	ADC5	&	ADC13
5	ADC6	&	ADC14
6	ADC7	&	ADC15
7	ADC8	&	ADC16

For all 16 ADCs unipolar: MS{anynode},MI988=$00
For all 16 ADCs bipolar: MS{anynode},MI988=$FF

- **MS{anynode},MI989: A/D Source Address**

MI989 specifies the starting source address (card) of the ADC inputs (dip switch setting).


Note

The first 16 ADC transfers use MACRO IC 0 parameters, as described above. The next (and last) 16 ADC transfers use the corresponding MACRO IC#1 parameters:

MS{anynode},MI1989
MS{anynode},MI1987
MS{anynode},MI1988

Automatic ADC processing Examples:

Setting up one ACC-59E (at base address $8800) to copy 16 ADCs, with ADCs 1, 2, 3, 4, 9, 10, 11, 12 as unipolar and ADCs 5, 6, 7, 8, 13, 14, 15, 16 as bipolar:

```
MS0,MI987=1         ; Enable automatic ADC read function
MS0,MI988=$F0       ; ADCs 1,2,3,4,9,10,11,12 as unipolar
                    ; and ADCs 5,6,7,8,13,14,15,16 as bipolar
MS0,MI989=$8800     ; Card base address
```

Setting up two ACC-59Es (at base addresses $8800 and $9800) to copy 32 ADCs, with all 16 ADCs of the 1st ACC-59E configured as unipolar and all 16 ADCs of the 2nd ACC-59E as bipolar:

```
// Setting up the automatic read function for the 1st ACC-59E
MS0,MI987=1         ; Enable automatic ADC read function (MACRO IC 0)
MS0,MI988=$00       ; All 16 ADCs unipolar
MS0,MI989=$8800     ; Card base address

// Setting up the automatic read function for the 2nd ACC-59E
MS0,MI1987=1        ; Enable automatic ADC read function (MACRO IC 1)
MS0,MI1988=$FF      ; All 16 ADCs bipolar
MS0,MI1989=$9800    ; Card base address
```


Note

After downloading these settings, it is necessary to issue **MSSAV0** and **MS$$$0** at the MACRO station to activate the automatic ADC processing.

The automatic ADC processing copies the data into the following registers ($020X) on the MACRO station:

1st ACC-59E			
Channel	**Location**	**Channel**	**Location**
ADC1	Y:$0200,0,12	ADC9	Y:$0200,12,12
ADC2	Y:$0201,0,12	ADC10	Y:$0201,12,12
ADC3	Y:$0202,0,12	ADC11	Y:$0202,12,12
ADC4	Y:$0203,0,12	ADC12	Y:$0203,12,12
ADC5	Y:$0204,0,12	ADC13	Y:$0204,12,12
ADC6	Y:$0205,0,12	ADC14	Y:$0205,12,12
ADC7	Y:$0206,0,12	ADC15	Y:$0206,12,12
ADC8	Y:$0207,0,12	ADC16	Y:$0207,12,12

2nd ACC-59E			
Channel	**Location**	**Channel**	**Location**
ADC1	Y:$0208,0,12	ADC9	Y:$0208,12,12
ADC2	Y:$0209,0,12	ADC10	Y:$0209,12,12
ADC3	Y:$020A,0,12	ADC11	Y:$020A,12,12
ADC4	Y:$020B,0,12	ADC12	Y:$020B,12,12
ADC5	Y:$020C,0,12	ADC13	Y:$020C,12,12
ADC6	Y:$020D,0,12	ADC14	Y:$020D,12,12
ADC7	Y:$020E,0,12	ADC15	Y:$020E,12,12
ADC8	Y:$020F,0,12	ADC16	Y:$020F,12,12

Note

Only ADCs 1-8 are meaningful when using ACC-59Es; i.e., the registers containing the results for ADCs 9-16 contain meaningless data when using ACC-59E.

If the user desires, he or she can monitor the demuxed ADC values directly from the MACRO 16 CPU over MACRO ASCII communication using the following suggested MM-Variables (download to MACRO 16 CPU via MACRO ASCII communication) for troubleshooting purposes, or for use in a MACRO PLCC:

Suggested User MM-Variables	
1st ACC-59E	**2nd ACC-59E**
`#define First59E_ADC1 MM1`	`#define Second59E_ADC1 MM9`
`#define First59E_ADC2 MM2`	`#define Second59E_ADC2 MM10`
`#define First59E_ADC3 MM3`	`#define Second59E_ADC3 MM11`
`#define First59E_ADC4 MM4`	`#define Second59E_ADC4 MM12`
`#define First59E_ADC5 MM5`	`#define Second59E_ADC5 MM13`
`#define First59E_ADC6 MM6`	`#define Second59E_ADC6 MM14`
`#define First59E_ADC7 MM7`	`#define Second59E_ADC7 MM15`
`#define First59E_ADC8 MM8`	`#define Second59E_ADC8 MM16`

	Unipolar	**Bipolar**
MACRO 16 CPU		

1st ACC-59E	`First59E_ADC1->Y:$0200,0,12,U`
	`First59E_ADC2->Y:$0201,0,12,U`
	`First59E_ADC3->Y:$0202,0,12,U`
	`First59E_ADC4->Y:$0203,0,12,U`
	`First59E_ADC5->Y:$0204,0,12,U`
	`First59E_ADC6->Y:$0205,0,12,U`
	`First59E_ADC7->Y:$0206,0,12,U`
	`First59E_ADC8->Y:$0207,0,12,U`
		`First59E_ADC1->Y:$0200,0,12,S`
		`First59E_ADC2->Y:$0201,0,12,S`
		`First59E_ADC3->Y:$0202,0,12,S`
		`First59E_ADC4->Y:$0203,0,12,S`
		`First59E_ADC5->Y:$0204,0,12,S`
		`First59E_ADC6->Y:$0205,0,12,S`
		`First59E_ADC7->Y:$0206,0,12,S`
		`First59E_ADC8->Y:$0207,0,12,S`

2nd ACC-59E	`Second59E_ADC1->Y:$0208,0,12,U`
	`Second59E_ADC2->Y:$0209,0,12,U`
	`Second59E_ADC3->Y:$020A,0,12,U`
	`Second59E_ADC4->Y:$020B,0,12,U`
	`Second59E_ADC5->Y:$020C,0,12,U`
	`Second59E_ADC6->Y:$020D,0,12,U`
	`Second59E_ADC7->Y:$020E,0,12,U`
	`Second59E_ADC8->Y:$020F,0,12,U`
		`Second59E_ADC1->Y:$0208,0,12,S`
		`Second59E_ADC2->Y:$0209,0,12,S`
		`Second59E_ADC3->Y:$020A,0,12,S`
		`Second59E_ADC4->Y:$020B,0,12,S`
		`Second59E_ADC5->Y:$020C,0,12,S`
		`Second59E_ADC6->Y:$020D,0,12,S`
		`Second59E_ADC7->Y:$020E,0,12,S`
		`Second59E_ADC8->Y:$020F,0,12,S`

Note

These MM-Variable definitions must be downloaded directly to the MACRO station via MACRO ASCII communication.

Transferring Data over I/O Nodes

The following explains the ADC data transfer from the $020X to $C0XX registers, and suggests user M-Variables for direct access.

It is assumed that the ADC processing has already been configured as explained in the previous step, and that the data is readily available in the $020X registers. Also, it is assumed that communication over the MACRO ring has already been established, and that the user is familiar with node activation on both the Ring Controller and MACRO Station. Thus, any node(s) used in the following examples have to be enabled.

Having multiple cards of different types on the MACRO Station can make the I/O transfer management cumbersome. Therefore, two alternative methods are suggested:

- Automatic I/O node transfer, using MS{anynode}MI173, and MI175
  This method is ideal for up to 2 ACC-59Es in one station.

- Manual I/O node transfer using MS{anynode}, MI20 through MI68.
  This method is recommended when more than two ACC-59E cards are present and/or there are other data cards requiring a large amount of data transfer over I/O nodes. This method can be used in conjunction with the automatic transfers.

Note

This section assumes that MACRO ring, I/O nodes, and ring check error settings have been configured properly.

Automatic I/O Node Data Transfer: MI173, MI174, MI175

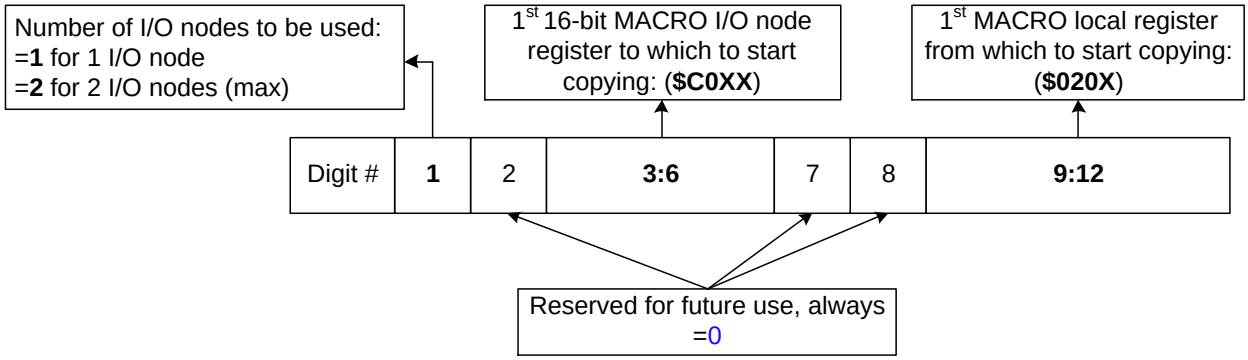
The automatic I/O node transfer of ADCs is achieved using the following MACRO Station (IC#0) parameters:

MS{anynode},MI173 Copies lower 12 bits of up to 6 consecutive registers into six 16-bit nodes
MS{anynode},MI175 Copies lower & upper 12 bits (24 bits) of up to 2 consecutive 24-bit registers

Note

For MACRO Station IC#1: MS{anynode},MI1173
MS{anynode},MI1175

These 48-bit variables are represented as 12 hexadecimal digits; they are set up as follows, where digit #1 is the leftmost digit when constructing the word:

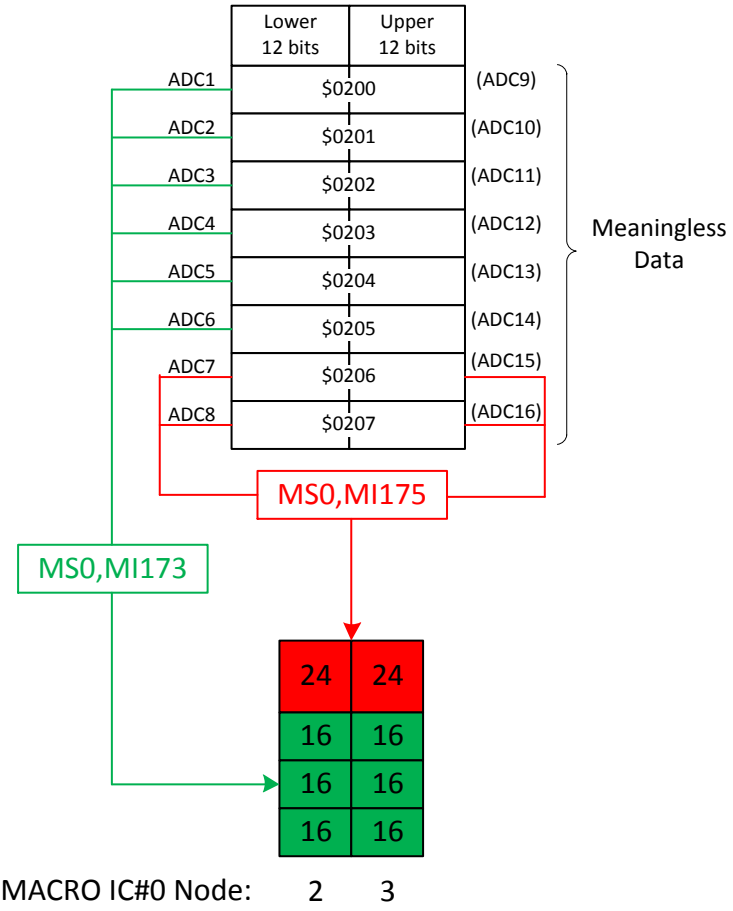


Following are two examples:

- Example 1: Automatic I/O node data transfer of 1 ACC-59E.
- Example 2: Automatic I/O node data transfer of 2 ACC-59Es.

Example 1: Setting up automatic I/O node transfer of 1 ACC-59E (total of 8 physical ADC channels) using I/O nodes 2 and 3.

ADC1 through ADC6 will be copied to the six 16-bit registers of nodes 2 and 3.
ADCs 7, 8 will be copied to the two 24-bit registers of nodes 2 and 3.




```
MS0,MI19=4      ; MACRO Station I/O Data Transfer Period (adjustable)

MS0,MI975=$CC   ; MACRO IC#0 I/O Node Enable, Nodes 2, 3, 6, 7

MS0,MI173=$20C0A1000200        ; ADC1 thru ADC6 (lower 12 bits of $0200 thru $0205)
MS0,MI175=$20C0A0000206        ; ADCs 7, 8, 15, 16 (lower + upper of $0206 thru $0207)
```

Suggested User M-Variables

```
#define First59E_ADC1  M5001
#define First59E_ADC2  M5002
#define First59E_ADC3  M5003
#define First59E_ADC4  M5004
#define First59E_ADC5  M5005
#define First59E_ADC6  M5006
#define First59E_ADC7  M5007
#define First59E_ADC8  M5008
```

Note

These suggested variable numbers are for a Turbo Ultralite only. If the user has a Non-Turbo Ultralite, variables in the range of M0-M1023 must be used.

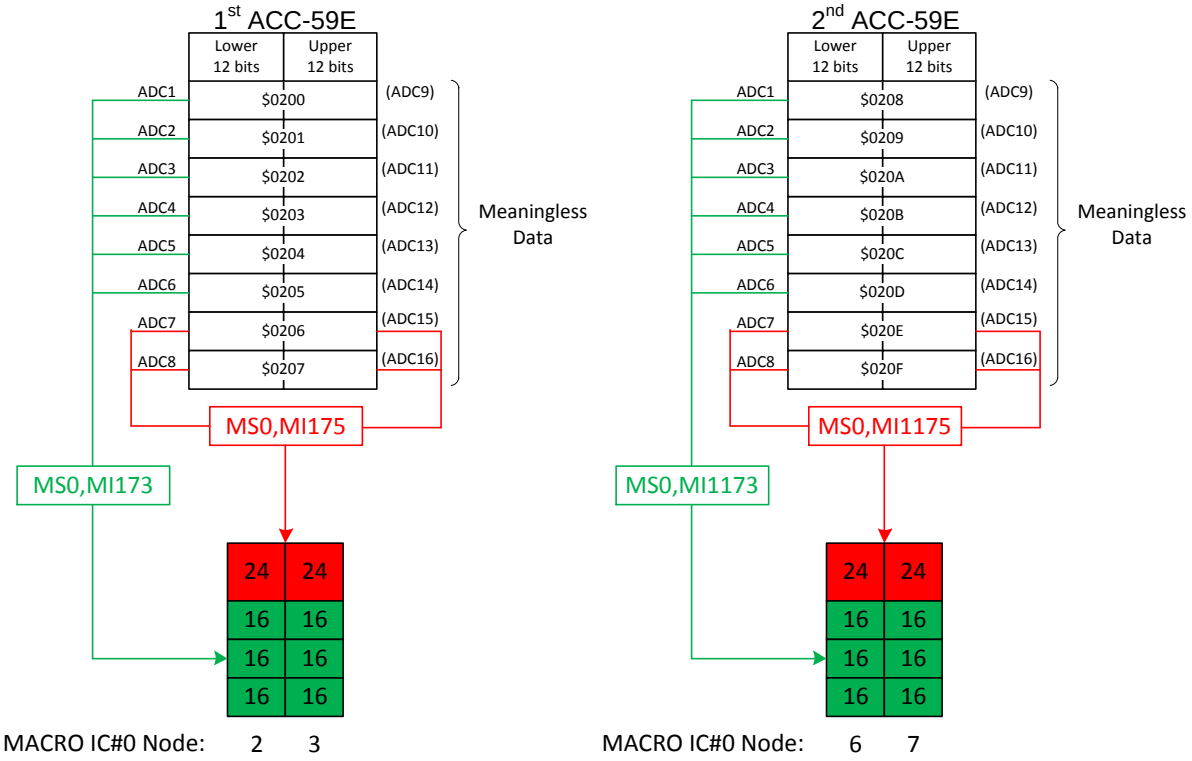
As setup by the ADC processing, the ADC pairs can be either unipolar (unsigned) or bipolar (signed):

Unipolar	Bipolar
Turbo PMAC2 Ultralite (or UMAC with ACC-5E)	
First59E_ADC1->X:$078421,8,12	First59E_ADC1->X:$078421,8,12,S
First59E_ADC2->X:$078422,8,12	First59E_ADC2->X:$078422,8,12,S
First59E_ADC3->X:$078423,8,12	First59E_ADC3->X:$078423,8,12,S
First59E_ADC4->X:$078425,8,12	First59E_ADC4->X:$078425,8,12,S
First59E_ADC5->X:$078426,8,12	First59E_ADC5->X:$078426,8,12,S
First59E_ADC6->X:$078427,8,12	First59E_ADC6->X:$078427,8,12,S
First59E_ADC7->X:$078420,0,12	First59E_ADC7->X:$078420,0,12,S
First59E_ADC8->X:$078424,0,12	First59E_ADC8->X:$078424,0,12,S
Non-Turbo PMAC2 Ultralite / MACRO Station Node Addresses	
First59E_ADC1->X:$C0A1,8,12	First59E_ADC1->X:$C0A1,8,12,S
First59E_ADC2->X:$C0A2,8,12	First59E_ADC2->X:$C0A2,8,12,S
First59E_ADC3->X:$C0A3,8,12	First59E_ADC3->X:$C0A3,8,12,S
First59E_ADC4->X:$C0A5,8,12	First59E_ADC4->X:$C0A5,8,12,S
First59E_ADC5->X:$C0A6,8,12	First59E_ADC5->X:$C0A6,8,12,S
First59E_ADC6->X:$C0A7,8,12	First59E_ADC6->X:$C0A7,8,12,S
First59E_ADC7->X:$C0A0,8,12	First59E_ADC7->X:$C0A0,8,12,S
First59E_ADC8->X:$C0A4,8,12	First59E_ADC8->X:$C0A4,8,12,S

Example 2: Setting up automatic I/O node transfer for two ACC-59Es (a total of 16 physical ADC channels).

First ACC-59E, using I/O nodes 2, 3, 6, and 7 of MACRO IC#0:
ADC1 through ADC6 will be copied to the six 16-bit registers of nodes 2 and 3 of MACRO IC#0.
ADCs 7, 8, 15, 16 will be copied to the two 24-bit registers of nodes 2 and 3 of MACRO IC#0.

Second ACC-59E, using I/O nodes 2, 3, 6, and 7 of MACRO IC#1:
ADC1 through ADC6 will be copied to the six 16-bit registers of nodes 2 and 3 of MACRO IC#1.
ADCs 7, 8, 15, 16 will be copied to the two 24-bit registers of nodes 2 and 3 of MACRO IC#1.




```
MS0,MI19=4      ; MACRO Station I/O Data Transfer Period (adjustable)
MS0,MI975=$CC   ; MACRO IC#0 I/O Node Enable, Nodes 2, 3

MS0,MI173=$20C0A1000200       ; 1st ACC-59E ADC1 thru ADC6 (lower 12 bits of $0200 thru $0205)
MS0,MI175=$20C0A0000206       ; 1st ACC-59E ADCs 7, 8, 15, 16 (lower + upper of $0206 thru $0207)
MS0,MI1975=$CC ; MACRO IC#1 I/O Node Enable, Nodes 2, 3

MS0,MI1173=$20C0E1000208      ; 2nd ACC-59E ADC1 thru ADC6 (lower 12 bits of $0208 thru $020D)
MS0,MI1175=$20C0E000020E      ; 2nd ACC-59E ADCs 7, 8, 15, 16 (lower + upper of $020E thru $020F)
```

Suggested User M-Variables:

1st ACC-59E	2nd ACC-59E
#define First59E_ADC1 M5001	#define Second59E_ADC1 M5009
#define First59E_ADC2 M5002	#define Second59E_ADC2 M5010
#define First59E_ADC3 M5003	#define Second59E_ADC3 M5011
#define First59E_ADC4 M5004	#define Second59E_ADC4 M5012
#define First59E_ADC5 M5005	#define Second59E_ADC5 M5013
#define First59E_ADC6 M5006	#define Second59E_ADC6 M5014
#define First59E_ADC7 M5007	#define Second59E_ADC7 M5015
#define First59E_ADC8 M5008	#define Second59E_ADC8 M5016

Note

These suggested variable numbers are for a Turbo Ultralite only. If the user has a Non-Turbo Ultralite, variables in the range of M0-M1023 must be used.

As setup by the ADC processing, the ADC pairs can be either unipolar (unsigned) or bipolar (signed):

	Unipolar	Bipolar
Turbo PMAC2 Ultralite (or UMAC with ACC-5E)		
1st ACC-59E	First59E_ADC1->X:$078421,8,12 First59E_ADC2->X:$078422,8,12 First59E_ADC3->X:$078423,8,12 First59E_ADC4->X:$078425,8,12 First59E_ADC5->X:$078426,8,12 First59E_ADC6->X:$078427,8,12 First59E_ADC7->X:$078420,8,12 First59E_ADC8->X:$078424,8,12	First59E_ADC1->X:$078421,8,12,S First59E_ADC2->X:$078422,8,12,S First59E_ADC3->X:$078423,8,12,S First59E_ADC4->X:$078425,8,12,S First59E_ADC5->X:$078426,8,12,S First59E_ADC6->X:$078427,8,12,S First59E_ADC7->X:$078420,8,12,S First59E_ADC8->X:$078424,8,12,S
2nd ACC-59E	Second59E_ADC1->X:$079421,8,12 Second59E_ADC2->X:$079422,8,12 Second59E_ADC3->X:$079423,8,12 Second59E_ADC4->X:$079425,8,12 Second59E_ADC5->X:$079426,8,12 Second59E_ADC6->X:$079427,8,12 Second59E_ADC7->X:$079420,8,12 Second59E_ADC8->X:$079424,8,12	Second59E_ADC1->X:$079421,8,12,S Second59E_ADC2->X:$079422,8,12,S Second59E_ADC3->X:$079423,8,12,S Second59E_ADC4->X:$079425,8,12,S Second59E_ADC5->X:$079426,8,12,S Second59E_ADC6->X:$079427,8,12,S Second59E_ADC7->X:$079420,8,12,S Second59E_ADC8->X:$079424,8,12,S
Non-Turbo PMAC2 Ultralite / MACRO Station Node Addresses		
1st ACC-59E	First59E_ADC1->X:$C0A1,8,12 First59E_ADC2->X:$C0A2,8,12 First59E_ADC3->X:$C0A3,8,12 First59E_ADC4->X:$C0A5,8,12 First59E_ADC5->X:$C0A6,8,12 First59E_ADC6->X:$C0A7,8,12 First59E_ADC7->X:$C0A0,8,12 First59E_ADC8->X:$C0A4,8,12	First59E_ADC1->X:$C0A1,8,12,S First59E_ADC2->X:$C0A2,8,12,S First59E_ADC3->X:$C0A3,8,12,S First59E_ADC4->X:$C0A5,8,12,S First59E_ADC5->X:$C0A6,8,12,S First59E_ADC6->X:$C0A7,8,12,S First59E_ADC7->X:$C0A0,8,12,S First59E_ADC8->X:$C0A4,8,12,S
2nd ACC-59E	Second59E_ADC1->X:$C0E1,8,12 Second59E_ADC2->X:$C0E2,8,12 Second59E_ADC3->X:$C0E3,8,12 Second59E_ADC4->X:$C0E5,8,12 Second59E_ADC5->X:$C0E6,8,12 Second59E_ADC6->X:$C0E7,8,12 Second59E_ADC7->X:$C0E0,8,12 Second59E_ADC8->X:$C0E4,8,12	Second59E_ADC1->X:$C0E1,8,12,S Second59E_ADC2->X:$C0E2,8,12,S Second59E_ADC3->X:$C0E3,8,12,S Second59E_ADC4->X:$C0E5,8,12,S Second59E_ADC5->X:$C0E6,8,12,S Second59E_ADC6->X:$C0E7,8,12,S Second59E_ADC7->X:$C0E0,8,12,S Second59E_ADC8->X:$C0E4,8,12,S

Manual I/O Node Data Transfer: MI19…MI68

The manual I/O node transfer of ADCs is achieved using the following MACRO Station parameters:

- **MS{anynode},MI19:** I/O data transfer period.

MI19 controls the data transfer period (in phase cycles) between the MACRO node interface registers and the I/O registers. If MI19 is set to 0, the data transfer is disabled.
MI19 is typically set to 4 phase cycles.

- **MS{anynode},MI20:** Data transfer enable mask.

MI20 controls which of 48 possible data transfer operations (specified by MI21-MI68) are performed at the data transfer period set by MI19.
MI20 is a 48-bit value; each bit controls whether the data transfer specified by one of the variables MI21 through MI68 is performed.

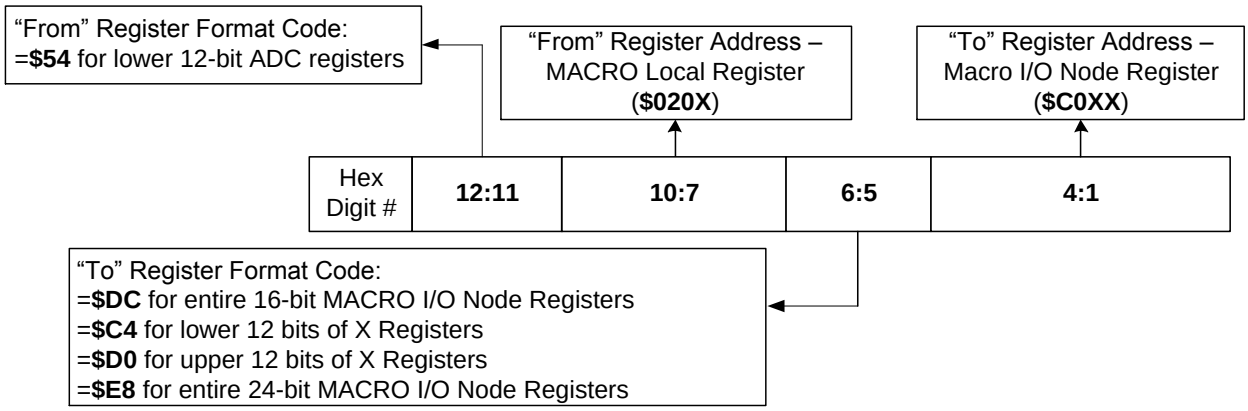
Examples:
If MI20=$1 → bit #0 is set to 1 → MI21 transfer is performed.
If MI20=$3 → bits #0 and #1 are set to 1 → MI21 and MI22 are performed.
If MI20=$5 → bits #0 and #2 are set to 1 → MI21 and MI23 are performed.
If MI20=$F → bits #0 through #3 are set to 1 → MI21 through MI24 are performed.

- **MS{anynode},MI21-MI68:** Data transfer source and destination address.

MI21-MI68 each specify a data transfer (copying) operation that will occur on the MACRO Station at a rate specified by Station Variable MI19 and are enabled by Station variable MI20.

Each variable specifies the address from which the data will be copied (read), and the address to which the data will be copied (written).

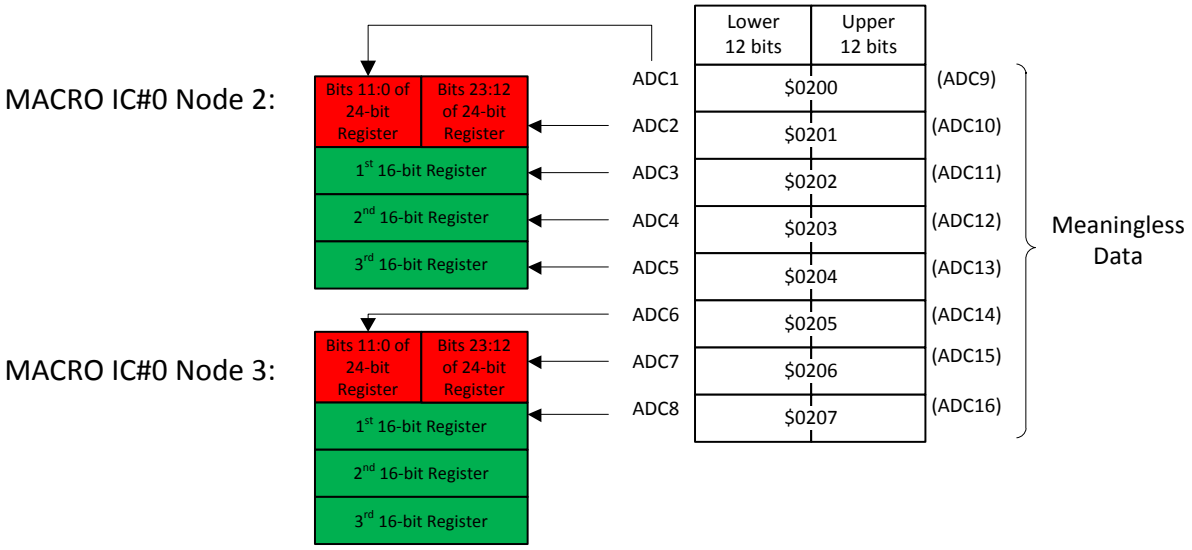
MI21-MI68 are 48-bit variables, represented as 12 hexadecimal digits as follows, where digit #1 is the rightmost digit when constructing the word:



Following are two examples:

- Example 1: Manual I/O node data transfer of 1 ACC-59E.
- Example 2: Manual I/O node data transfer of 2 ACC-59Es.

Example 1: One ACC-59E at $8800, Bipolar




```
//***** ACC-59E Manual MACRO I/O Transfer Example *****//
// Uses MI19, MI20, and MI21-MI30
// Configures 1st ACC-59E at base address $8800 (=$78C00 on Turbo), all Bipolar

// Setting up the automatic read function for the 1st ACC-59E
MS0,MI987=1     ; Enable Automatic ADC Demuxing
MS0,MI988=$FF   ; All 16 ADCs Bipolar
MS0,MI989=$8800 ; Card base address
MS0,MI975=$0C   ; MACRO IC#0 I/O node Enable, nodes 2, 3
MS0,MI19=4      ; Perform the transfer every 4 phase cycles
MS0,MI20=$3FF    ; Transfer 10 I/O points every MI19 phase cycles (use MI21-MI30)

// For the 1st ACC-59E
MS0,MI21=$540200C40700 ; ADC 1 goes to lower 12 bits of open memory at X:$0700
MS0,MI22=$540201D00700 ; ADC 2 goes to upper 12 bits of open memory at X:$0700
MS0,MI23=$E80700E8C0A0 ; ADC 1 goes to lower 12 bits of 24-bit reg. of Node 2, ADC 2 to upper 12 bits
MS0,MI24=$540202DCC0A1 ; ADC 3 goes to 1st 16-bit register of Node 2
MS0,MI25=$540203DCC0A2 ; ADC 4 goes to 2nd 16-bit register of Node 2
MS0,MI26=$540204DCC0A3 ; ADC 5 goes to 3rd 16-bit register of Node 2
MS0,MI27=$540205C40701 ; ADC 6 goes to lower 12 bits of open memory at X:$0701
MS0,MI28=$540206D00701 ; ADC 7 goes to upper 12 bits of open memory at X:$0701
MS0,MI29=$E80701E8C0A4 ; ADC 6 goes to lower 12 bits of 24-bit reg. of node 3, ADC 7 to upper 12 bits
MS0,MI30=$540207DCC0A5 ; ADC 8 goes to 1st 16-bit register of Node 3
```

Note

This example uses open memory X:$0700 and X:$0701 on the MACRO station to prepare the 24-bit words to copy to Node 2's and Node 3's 24-bit registers.

Suggested User M-Variables:

1st ACC-59E
#define First59E_ADC1 M5001
#define First59E_ADC2 M5002
#define First59E_ADC3 M5003
#define First59E_ADC4 M5004
#define First59E_ADC5 M5005
#define First59E_ADC6 M5006
#define First59E_ADC7 M5007
#define First59E_ADC8 M5008

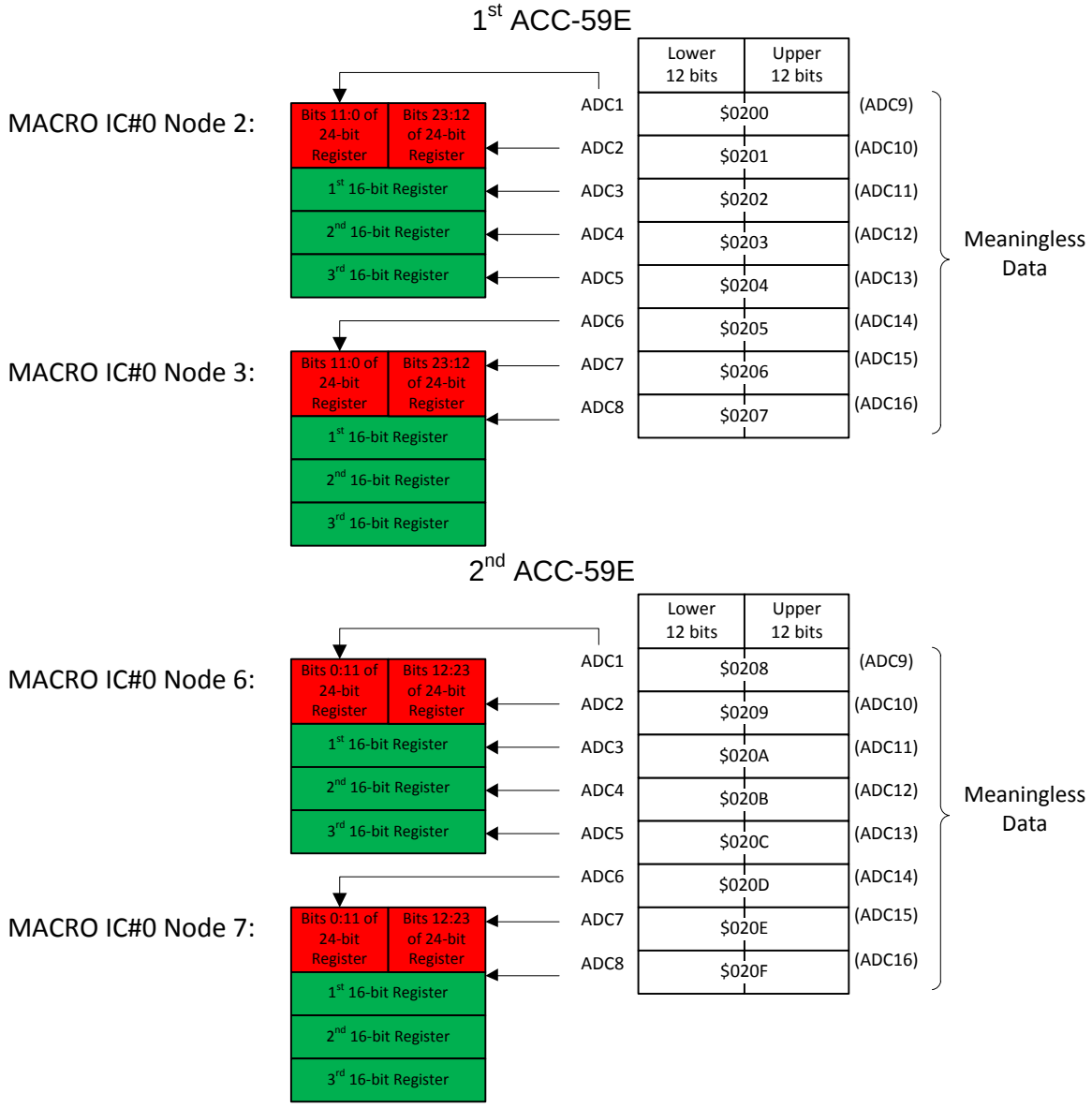
Note

These suggested variable numbers are for a Turbo Ultralite only. If the user has a Non-Turbo Ultralite, variables in the range of M0-M1023 must be used.

As set up by the ADC processing, the ADC pairs can be either unipolar (unsigned) or bipolar (signed):

	Unipolar	Bipolar
Turbo PMAC2 Ultralite (or UMAC with ACC-5E)		
1st ACC-59E	First59E_ADC1->X:$78420,0,12 First59E_ADC2->X:$78420,12,12 First59E_ADC3->X:$78421,8,12 First59E_ADC4->X:$78422,8,12 First59E_ADC5->X:$78423,8,12 First59E_ADC6->X:$78424,0,12 First59E_ADC7->X:$78424,12,12 First59E_ADC8->X:$78425,8,12	First59E_ADC1->X:$78420,0,12,S First59E_ADC2->X:$78420,12,12,S First59E_ADC3->X:$78421,8,12,S First59E_ADC4->X:$78422,8,12,S First59E_ADC5->X:$78423,8,12,S First59E_ADC6->X:$78424,0,12,S First59E_ADC7->X:$78424,12,12,S First59E_ADC8->X:$78425,8,12,S
Non-Turbo PMAC2 Ultralite / MACRO Station Node Addresses		
1st ACC-59E	First59E_ADC1->X:$C0A0,0,12 First59E_ADC2->X:$C0A0,12,12 First59E_ADC3->X:$C0A1,8,12 First59E_ADC4->X:$C0A2,8,12 First59E_ADC5->X:$C0A3,8,12 First59E_ADC6->X:$C0A4,0,12 First59E_ADC7->X:$C0A4,12,12 First59E_ADC8->X:$C0A5,8,12	First59E_ADC1->X:$C0A0,0,12,S First59E_ADC2->X:$C0A0,12,12,S First59E_ADC3->X:$C0A1,8,12,S First59E_ADC4->X:$C0A2,8,12,S First59E_ADC5->X:$C0A3,8,12,S First59E_ADC6->X:$C0A4,0,12,S First59E_ADC7->X:$C0A4,12,12,S First59E_ADC8->X:$C0A5,8,12,S

Example 2: 1st ACC-59E at $8800, Unipolar, and 2nd ACC-59E at $9800, Bipolar




```
//***** ACC-59E Manual MACRO I/O Transfer Example *****//
// Uses MI19, MI20, and MI21-MI40
// Configures 1st ACC-59E at base address $8800 (=$78C00 in Turbo), all Unipolar,
// and configures 2nd ACC-59E at base address $9800 (=$79C00 in Turbo), all Bipolar

// Setting up the automatic read function for the 1st ACC-59E
MS0,MI987=1     ; Enable Automatic ADC Demuxing
MS0,MI988=$00   ; All 16 ADCs Unipolar
MS0,MI989=$8800 ; Card base address
MS0,MI975=$CC   ; MACRO IC#0 I/O node Enable, nodes 2, 3, 6, 7

// Setting up the automatic read function for the 2nd ACC-59E
MS0,MI1987=1     ; Enable automatic ADC read function (MACRO IC 1)
MS0,MI1988=$FF   ; All 16 ADCs Bipolar
MS0,MI1989=$9800 ; Card base address
MS0,MI1975=$00   ; MACRO IC#1 I/O node Enable, no nodes needed

MS0,MI19=4      ; Perform the transfer every 4 phase cycles
MS0,MI20=$FFFFF  ; Transfer 20 I/O points every MI19 phase cycles (use MI21-MI04)
```

```
// For the 1st ACC-59E
MS0,MI21=$540200C40700 ; ADC 1 goes to lower 12 bits of open memory at X:$0700
MS0,MI22=$540201D00700 ; ADC 2 goes to upper 12 bits of open memory at X:$0700
MS0,MI23=$E80700E8C0A0 ; ADC 1 goes to lower 12 bits of 24-bit reg. of Node 2, ADC 2 to upper 12 bits
MS0,MI24=$540202DCC0A1 ; ADC 3 goes to 1st 16-bit register of Node 2
MS0,MI25=$540203DCC0A2 ; ADC 4 goes to 2nd 16-bit register of Node 2
MS0,MI26=$540204DCC0A3 ; ADC 5 goes to 3rd 16-bit register of Node 2
MS0,MI27=$540205C40701 ; ADC 6 goes to lower 12 bits of open memory at X:$0701
MS0,MI28=$540206D00701 ; ADC 7 goes to upper 12 bits of open memory at X:$0701
MS0,MI29=$E80701E8C0A4 ; ADC 6 goes to lower 12 bits of 24-bit reg. of node 3, ADC 7 to upper 12 bits
MS0,MI30=$540207DCC0A5 ; ADC 8 goes to 1st 16-bit register of Node 3

// For the 2nd ACC-59E
MS0,MI31=$540208C40702 ; ADC 1 goes to lower 12 bits of open memory at X:$0702
MS0,MI32=$540209D00702 ; ADC 2 goes to upper 12 bits of open memory at X:$0702
MS0,MI33=$E80702E8C0A8 ; ADC 1 goes to lower 12 bits of 24-bit reg. of Node 6, ADC 2 to upper 12 bits
MS0,MI34=$54020ADCC0A9 ; ADC 3 goes to 1st 16-bit register of Node 6
MS0,MI35=$54020BDCC0AA ; ADC 4 goes to 2nd 16-bit register of Node 6
MS0,MI36=$54020CDCC0AB ; ADC 5 goes to 3rd 16-bit register of Node 6
MS0,MI37=$54020DC40703 ; ADC 6 goes to lower 12 bits of open memory at X:$0703
MS0,MI38=$54020ED00703 ; ADC 7 goes to upper 12 bits of open memory at X:$0703
MS0,MI39=$E80703E8C0AC ; ADC 6 goes to lower 12 bits of 24-bit reg. of node 7, ADC 7 to upper 12 bits
MS0,MI40=$54020FDCC0AD ; ADC 8 goes to 1st 16-bit register of Node 7
```

Note

This example uses open memory X:$0702 and X:$0703 on the MACRO station to prepare the 24-bit words to copy to Node 6's and Node 7's 24-bit registers.

Suggested User M-Variables	
1st ACC-59E	**2nd ACC-59E**
#define First59E_ADC1 M5001	#define Second59E_ADC1 M5009
#define First59E_ADC2 M5002	#define Second59E_ADC2 M5010
#define First59E_ADC3 M5003	#define Second59E_ADC3 M5011
#define First59E_ADC4 M5004	#define Second59E_ADC4 M5012
#define First59E_ADC5 M5005	#define Second59E_ADC5 M5013
#define First59E_ADC6 M5006	#define Second59E_ADC6 M5014
#define First59E_ADC7 M5007	#define Second59E_ADC7 M5015
#define First59E_ADC8 M5008	#define Second59E_ADC8 M5016

Note

These suggested variable numbers are for a Turbo Ultralite only. If the user has a Non-Turbo Ultralite, variables in the range of M0-M1023 must be used.

As set up by the ADC processing, the ADC pairs can be either unipolar (unsigned) or bipolar (signed):

	Unipolar	**Bipolar**
Turbo PMAC2 Ultralite (or UMAC with ACC-5E)		
1st ACC-59E	First59E_ADC1->X:$78420,0,12 First59E_ADC2->X:$78420,12,12 First59E_ADC3->X:$78421,8,12 First59E_ADC4->X:$78422,8,12 First59E_ADC5->X:$78423,8,12 First59E_ADC6->X:$78424,0,12 First59E_ADC7->X:$78424,12,12 First59E_ADC8->X:$78425,8,12	First59E_ADC1->X:$78420,0,12,S First59E_ADC2->X:$78420,12,12,S First59E_ADC3->X:$78421,8,12,S First59E_ADC4->X:$78422,8,12,S First59E_ADC5->X:$78423,8,12,S First59E_ADC6->X:$78424,0,12,S First59E_ADC7->X:$78424,12,12,S First59E_ADC8->X:$78425,8,12,S
2nd ACC-59E	Second59E_ADC1->X:$78428,0,12 Second59E_ADC2->X:$78428,12,12 Second59E_ADC3->X:$78429,8,12 Second59E_ADC4->X:$7842A,8,12 Second59E_ADC5->X:$7842B,8,12 Second59E_ADC6->X:$7842C,0,12 Second59E_ADC7->X:$7842C,12,12 Second59E_ADC8->X:$7842D,8,12	Second59E_ADC1->X:$78428,0,12,S Second59E_ADC2->X:$78428,12,12,S Second59E_ADC3->X:$78429,8,12,S Second59E_ADC4->X:$7842A,8,12,S Second59E_ADC5->X:$7842B,8,12,S Second59E_ADC6->X:$7842C,0,12,S Second59E_ADC7->X:$7842C,12,12,S Second59E_ADC8->X:$7842D,8,12,S
Non-Turbo PMAC2 Ultralite / MACRO Station Node Addresses		
1st ACC-59E	First59E_ADC1->X:$C0A0,0,12 First59E_ADC2->X:$C0A0,12,12 First59E_ADC3->X:$C0A1,8,12 First59E_ADC4->X:$C0A2,8,12 First59E_ADC5->X:$C0A3,8,12 First59E_ADC6->X:$C0A4,0,12 First59E_ADC7->X:$C0A4,12,12 First59E_ADC8->X:$C0A5,8,12	First59E_ADC1->X:$C0A0,0,12,S First59E_ADC2->X:$C0A0,12,12,S First59E_ADC3->X:$C0A1,8,12,S First59E_ADC4->X:$C0A2,8,12,S First59E_ADC5->X:$C0A3,8,12,S First59E_ADC6->X:$C0A4,0,12,S First59E_ADC7->X:$C0A4,12,12,S First59E_ADC8->X:$C0A5,8,12,S
2nd ACC-59E	Second59E_ADC1->X:$C0A8,0,12 Second59E_ADC2->X:$C0A8,12,12 Second59E_ADC3->X:$C0A9,8,12 Second59E_ADC4->X:$C0AA,8,12 Second59E_ADC5->X:$C0AB,8,12 Second59E_ADC6->X:$C0AC,0,12 Second59E_ADC7->X:$C0AC,12,12 Second59E_ADC8->X:$C0AD,8,12	Second59E_ADC1->X:$C0A8,0,12,S Second59E_ADC2->X:$C0A8,12,12,S Second59E_ADC3->X:$C0A9,8,12,S Second59E_ADC4->X:$C0AA,8,12,S Second59E_ADC5->X:$C0AB,8,12,S Second59E_ADC6->X:$C0AC,0,12,S Second59E_ADC7->X:$C0AC,12,12,S Second59E_ADC8->X:$C0AD,8,12,S

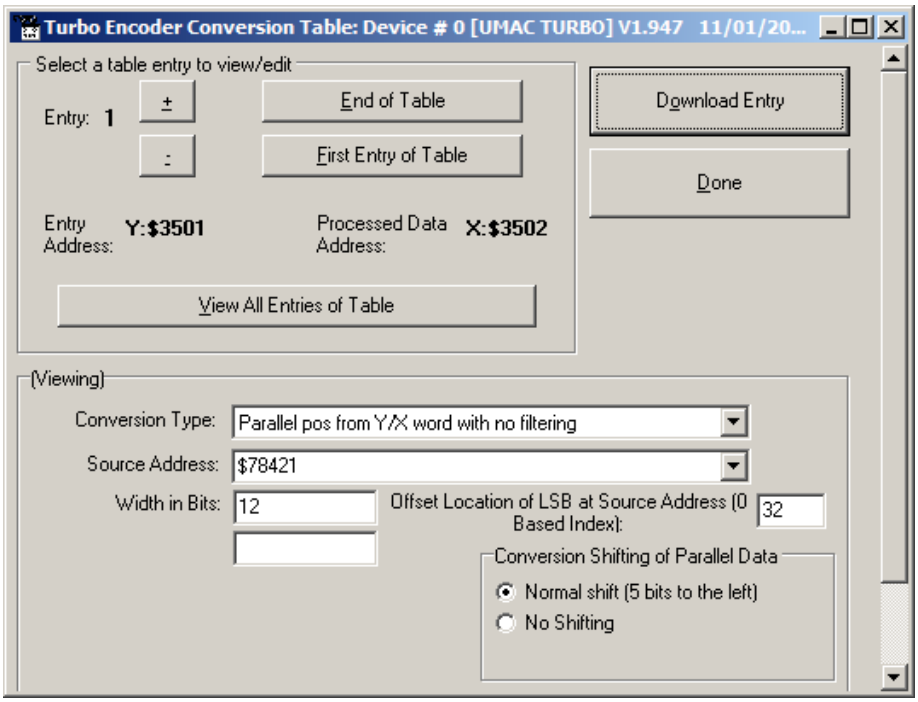
Using an Analog Input for Servo Feedback over MACRO

The ACC-59E analog inputs can be used as a feedback device for a servo motor, even over MACRO.

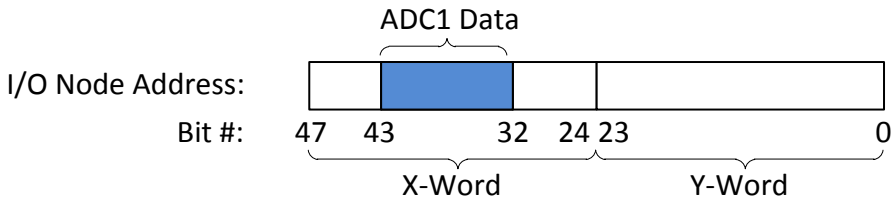
Note

- Refer to Delta Tau's released application notes or Turbo User Manual for cascaded-loop control (i.e. force, height control around position loop).
- The automatic ADC read function is recommended for this application.

Example:
Setting up Motor #1 position and velocity feedback from an ADC channel coming over the MACRO ring on MACRO IC#0, Node 2, 1st 16-bit register (bits 8-19 of X:$78421).
The analog input is brought into the Encoder Conversion Table as a Parallel Y/X word with no filtering:



The Offset Location of LSB at Source Address is set to 32 because this type of ECT entry begins at bit 0 of the Y-word (24 bits wide) of the I/O Node Address. So, to get to bit 8 of the X-word, one must offset the LSB by 32 (=24+8):



Under "Conversion Shifting of Parallel Data", use "Normal shift" to scale the ADC result to its proper magnitude.

The equivalent code in Turbo PMAC Encoder Conversion Table parameters:

```
I8000=$678421  ; Unfiltered parallel pos of location X:$78421
I8001=$00C020  ; Width and Offset.
```

The position and velocity pointers are then set to the processed data address (i.e. $3502):

```
I103=$3502     ; Motor #1 position loop feedback address
I104=$3502     ; Motor #1 velocity loop feedback address
```

Note

If "No Shifting" is used in this example (see "Conversion Shifting of Parallel Data" in the screenshot on the preceding page), the result of the Encoder Conversion Table entry (in the X:$3502 register in this example) must be multiplied by 32 (=2^5) in order to obtain the same value stored in bits 8-19 of X:$78421.

Analog Input Power-On Position over MACRO

Some analog devices are absolute along the travel range of the motor (e.g., in hydraulic piston applications). Generally, it is desirable to obtain the motor position (input voltage) on power up or reset.

This procedure can be done in a simple PLC on power-up by writing the processed A/D data into the motor actual position register (suggested M-Variable Mxx62).

Note

- If the automatic ADC read method is being used, waiting a delay of about ½ second after the PMAC boots should be allowed for processing the data before copying it into the motor actual position register.
- If the manual ADC read method is being used, it is recommended to add this procedure at the end of the manual read PLC, or in a subsequent PLC with ½ sec delay to allow data processing.

Example: Reading Motor #1 position on power-up or reset, assuming that the automatic read function is used, and that M5001 is predefined to read bits 8-19 of the 1st 16-bit register of Node 2 of MACRO IC#0 where the result of the first ACC-59E's ADC channel 1 (unipolar) resides.

```
End Gat
Del Gat
Close

#define Mtr1ActPos     M162    ; Motor #1 Actual Position
               ; Suggested M-Variable units of 1/32*I108
Mtr1ActPos->D:$8B

#define First59E_ADC1         M5001   ; Channel 1 ADC
First59E_ADC1->X:$078421,8,12,U       ; Unipolar

Open PLC 1 Clear
I5111=500*8388608/I10
While(I5111>0) EndWhile               ; ½ sec delay
Mtr1ActPos=First59E_ADC1*32*I108      ; Motor #1 Actual Position (scaled to motor counts)
Disable PLC 1          ; Scan once on power-up or reset
Close
```

Setting Up the Analog Outputs (DACs) over MACRO

WARNING

In bipolar mode, the DAC output is not at zero volts on power-up or reset. It is required to set the corresponding M-Variable (register) to 2047 for zero voltage output on power-up or reset.

The MM-Variable pointers for the analog outputs (DACs) on the ACC-59E are mapped as follows:

DAC #	Address Location	12-bit Location
1	Base Address + $8	[11:0]
2	Base Address + $9	[11:0]
3	Base Address + $A	[11:0]
4	Base Address + $B	[11:0]

DAC #	Address Location	12-bit Location
5	Base Address + $8	[23:12]
6	Base Address + $9	[23:12]
7	Base Address + $A	[23:12]
8	Base Address + $B	[23:12]

Example: Suggested User M-Variables for DAC Output on Two ACC-59Es:

1st ACC-59E (MACRO Base Address $8800)	2nd ACC-59E (MACRO Base Address $9800)
MM101->Y:$8808,0,12,U ; DAC Channel #1	MM109->Y:$9808,0,12,U ; DAC Channel #1
MM102->Y:$8809,0,12,U ; DAC Channel #2	MM110->Y:$9809,0,12,U ; DAC Channel #2
MM103->Y:$880A,0,12,U ; DAC Channel #3	MM111->Y:$980A,0,12,U ; DAC Channel #3
MM104->Y:$880B,0,12,U ; DAC Channel #4	MM112->Y:$980B,0,12,U ; DAC Channel #4
MM105->Y:$8808,12,12,U ; DAC Channel #5	MM113->Y:$9808,12,12,U ; DAC Channel #5
MM106->Y:$8809,12,12,U ; DAC Channel #6	MM114->Y:$9809,12,12,U ; DAC Channel #6
MM107->Y:$880A,12,12,U ; DAC Channel #7	MM115->Y:$980A,12,12,U ; DAC Channel #7
MM108->Y:$880B,12,12,U ; DAC Channel #8	MM116->Y:$980B,12,12,U ; DAC Channel #8

Note

These MM-Variable definitions must be downloaded directly to the MACRO station.

Testing the Analog Outputs

The Analog Outputs out of the ACC-59E can be wired as single ended (DAC+ & Ground) or differential (DAC+ & DAC-) signals.

Writing the software counts shown below to the corresponding M-Variables should result in the following voltages:

DAC Output	Software Counts (Write)	Single-Ended [V] DAC+ ←→ AGND	Differential [V] DAC+ ←→ DAC-
Unipolar Mode	0	0	0
	2047	5	10
	4095	10	20
Bipolar Mode	0	-10	-20
	2047	0	0
	4095	10	20

Analog Output (DAC) MACRO I/O Transfer

This section describes how to manually transfer DAC data across MACRO by using a PLC and the **MSW{node#}** command.

Example: Setting up two ACC-59Es with bipolar DAC outputs on a MACRO station.

In this example, the user can simply write the desired value to M6001 (see table below) through M6016 on the Ultralite as though the DACs were on the Ultralite:

Physical DAC Channel	Ultralite M-Variable	MACRO Station MM-Variable
1st ACC-59E, DAC #1	M6001	MM101
1st ACC-59E, DAC #2	M6002	MM102
1st ACC-59E, DAC #3	M6003	MM103
1st ACC-59E, DAC #4	M6004	MM104
1st ACC-59E, DAC #5	M6005	MM105
1st ACC-59E, DAC #6	M6006	MM106
1st ACC-59E, DAC #7	M6007	MM107
1st ACC-59E, DAC #8	M6008	MM108
2nd ACC-59E, DAC #1	M6009	MM109
2nd ACC-59E, DAC #2	M6010	MM110
2nd ACC-59E, DAC #3	M6011	MM111
2nd ACC-59E, DAC #4	M6012	MM112
2nd ACC-59E, DAC #5	M6013	MM113
2nd ACC-59E, DAC #6	M6014	MM114
2nd ACC-59E, DAC #7	M6015	MM115
2nd ACC-59E, DAC #8	M6016	MM116

The PLC will then copy, using the **MSW{node#}** command, the values in M6001 through M6016 to MM101 through M116, respectively, on the MACRO station where the physical DACs reside. This example requires that M6001 through M6016 be free on the Ultralite, and MM101 through MM116 be defined as shown on the previous page of this manual under the "Suggested User M-Variables for DAC Output on Two ACC-59Es" example.

Example Code:

```
// PLC Example to write to DAC outputs using Turbo Ultralite M-Variables
// For two ACC-59Es
M6001..6016->*                ; Self-assign M-Variables to store DAC values on Ultralite
M6001..6016=2047              ; Initial State for DACs (to be saved to the PMAC)
                              ; (2047=0 V for bipolar DACs, 0=0 V for unipolar DACs) -User Input
P6001..6016=0                 ; Latching Flags
I5=I5|2                       ; Enable background PLCs if yet not enabled

Open PLC 2 Clear
If (P6001!=M6001)        ; If 1st Card DAC1's value changed
       MSW0,MM101,M6001 ; MM101 is 1st Card DAC1, Copy M6001 from Ultralite to MM101 on MACRO Station
       I5111=1 While(I5111>0) EndW   ; Wait 1 servo cycle delay to force MSW0 to execute
       P6001=M6001                   ; Latch 1st Card DAC1 value
EndIf
If (P6002!=M6002)        ; If 1st Card DAC2's value changed
       MSW0,MM102,M6002 ; MM102 is 1st Card DAC2, Copy M6002 from Ultralite to MM102 on MACRO Station
       I5111=1 While(I5111>0) EndW   ; Wait 1 servo cycle delay to force MSW0 to execute
       P6002=M6002                   ; Latch 1st Card DAC2 value
EndIf
If (P6003!=M6003)        ; If 1st Card DAC3's value changed
       MSW0,MM103,M6003 ; MM103 is 1st Card DAC3, Copy M6003 from Ultralite to MM103 on MACRO Station
       I5111=1 While(I5111>0) EndW   ; Wait 1 servo cycle delay to force MSW0 to execute
       P6003=M6003                   ; Latch 1st Card DAC3 value
EndIf
If (P6004!=M6004)        ; If 1st Card DAC4's value changed
       MSW0,MM104,M6004 ; MM104 is 1st Card DAC4, Copy M6004 from Ultralite to MM104 on MACRO Station
       I5111=1 While(I5111>0) EndW   ; Wait 1 servo cycle delay to force MSW0 to execute
       P6004=M6004                   ; Latch 1st Card DAC4 value
EndIf
If (P6005!=M6005)        ; If 1st Card DAC5's value changed
       MSW0,MM105,M6005 ; MM105 is 1st Card DAC5, Copy M6005 from Ultralite to MM105 on MACRO Station
       I5111=1 While(I5111>0) EndW   ; Wait 1 servo cycle delay to force MSW0 to execute
       P6005=M6005                   ; Latch 1st Card DAC5 value
EndIf
If (P6006!=M6006)        ; If 1st Card DAC6's value changed
       MSW0,MM106,M6006 ; MM106 is 1st Card DAC6, Copy M6006 from Ultralite to MM106 on MACRO Station
       I5111=1 While(I5111>0) EndW   ; Wait 1 servo cycle delay to force MSW0 to execute
       P6006=M6006                   ; Latch 1st Card DAC6 value
EndIf
If (P6007!=M6007)        ; If 1st Card DAC7's value changed
       MSW0,MM107,M6007 ; MM107 is 1st Card DAC7, Copy M6007 from Ultralite to MM107 on MACRO Station
       I5111=1 While(I5111>0) EndW   ; Wait 1 servo cycle delay to force MSW0 to execute
       P6007=M6007                   ; Latch 1st Card DAC7 value
EndIf
If (P6008!=M6008)        ; If 1st Card DAC8's value changed
       MSW0,MM108,M6008 ; MM108 is 1st Card DAC8, Copy M6008 from Ultralite to MM108 on MACRO Station
       I5111=1 While(I5111>0) EndW   ; Wait 1 servo cycle delay to force MSW0 to execute
       P6008=M6008                   ; Latch 1st Card DAC8 value
EndIf
If (P6009!=M6009)        ; If 2nd Card DAC1's value changed
       MSW0,MM109,M6009 ; MM109 is 2nd Card DAC1, Copy M6009 from Ultralite to MM109 on MACRO Station
       I5111=1 While(I5111>0) EndW   ; Wait 1 servo cycle delay to force MSW0 to execute
       P6009=M6009                   ; Latch 2nd Card DAC1 value
EndIf
If (P6010!=M6010)        ; If 2nd Card DAC2's value changed
       MSW0,MM110,M6010 ; MM110 is 2nd Card DAC2, Copy M6010 from Ultralite to MM110 on MACRO Station
       I5111=1 While(I5111>0) EndW   ; Wait 1 servo cycle delay to force MSW0 to execute
       P6010=M6010                   ; Latch 2nd Card DAC2 value
EndIf
If (P6011!=M6011)        ; If 2nd Card DAC3's value changed
       MSW0,MM111,M6011 ; MM111 is 2nd Card DAC3, Copy M6011 from Ultralite to MM111 on MACRO Station
       I5111=1 While(I5111>0) EndW   ; Wait 1 servo cycle delay to force MSW0 to execute
       P6011=M6011                   ; Latch 2nd Card DAC3 value
EndIf
```

```
If (P6012!=M6012)        ; If 2nd Card DAC4's value changed
       MSW0,MM112,M6012 ; MM112 is 2nd Card DAC4, Copy M6012 from Ultralite to MM112 on MACRO Station
       I5111=1 While(I5111>0) EndW   ; Wait 1 servo cycle delay to force MSW0 to execute
       P6012=M6012                   ; Latch 2nd Card DAC4 value
EndIf
If (P6013!=M6013)        ; If 2nd Card DAC5's value changed
       MSW0,MM113,M6013 ; MM113 is 2nd Card DAC5, Copy M6013 from Ultralite to MM113 on MACRO Station
       I5111=1 While(I5111>0) EndW   ; Wait 1 servo cycle delay to force MSW0 to execute
       P6013=M6013                   ; Latch 2nd Card DAC5 value
EndIf
If (P6014!=M6014)        ; If 2nd Card DAC6's value changed
       MSW0,MM114,M6014 ; MM114 is 2nd Card DAC6, Copy M6014 from Ultralite to MM114 on MACRO Station
       I5111=1 While(I5111>0) EndW   ; Wait 1 servo cycle delay to force MSW0 to execute
       P6014=M6014                   ; Latch 2nd Card DAC6 value
EndIf
If (P6015!=M6015)        ; If 2nd Card DAC7's value changed
       MSW0,MM115,M6015 ; MM115 is 2nd Card DAC7, Copy M6015 from Ultralite to MM115 on MACRO Station
       I5111=1 While(I5111>0) EndW   ; Wait 1 servo cycle delay to force MSW0 to execute
       P6015=M6015                   ; Latch 2nd Card DAC7 value
EndIf
If (P6016!=M6016)        ; If 2nd Card DAC8's value changed
       MSW0,MM116,M6016 ; MM116 is 2nd Card DAC8, Copy M6016 from Ultralite to MM116 on MACRO Station
       I5111=1 While(I5111>0) EndW   ; Wait 1 servo cycle delay to force MSW0 to execute
       P6016=M6016                   ; Latch 2nd Card DAC8 value
EndIf
Close
```

Troubleshooting DAC Outputs over MACRO

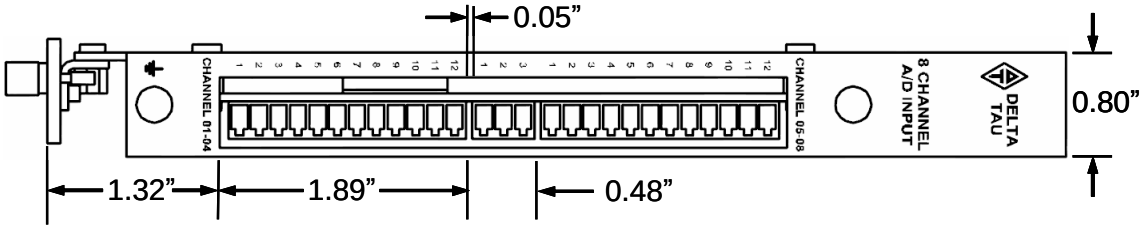
Another way to transfer DAC information is described as follows:

One can write directly to MACRO I/O nodes using M-Variables to get the desired DAC values to transfer across the MACRO ring to the MACRO station and then one can use MI19 through MI68 on the MACRO station to transfer DAC data from the I/O nodes on the MACRO station to the physical ACC-59E DAC registers on the MACRO station. However, since the "Manual I/O Node Data Transfer: MI19…MI68" section of this manual already uses MI19-MI68 for ADC transfer, an example of doing this will not be given here.

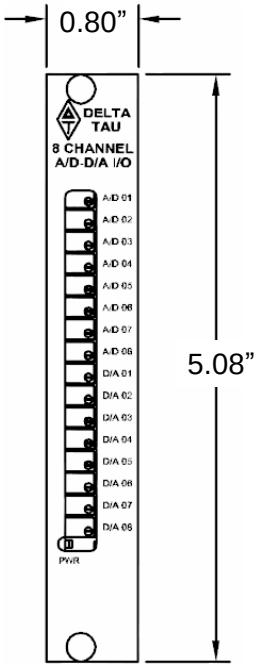
ACC-59E LAYOUT & PINOUTS

Terminal Block Option

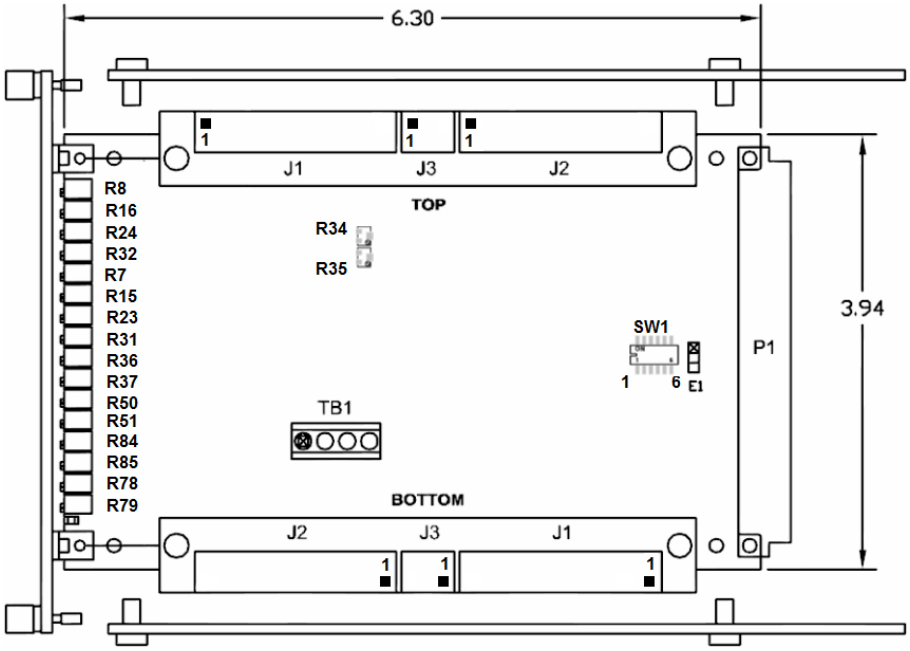
Top View



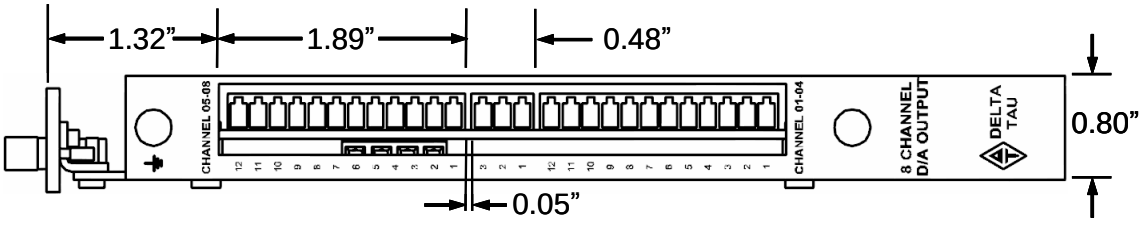
Front View



Side View



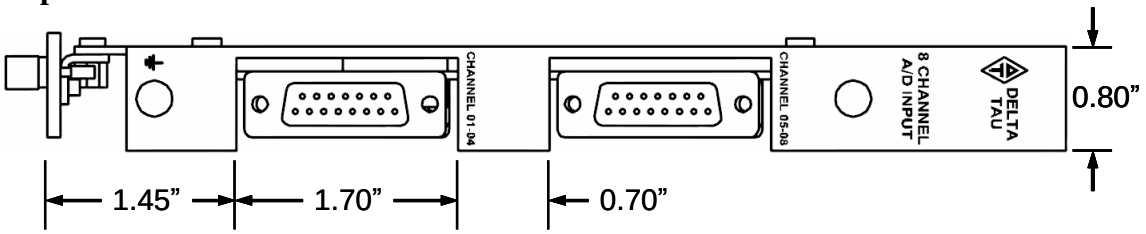
Bottom View



All dimensions are in inches.

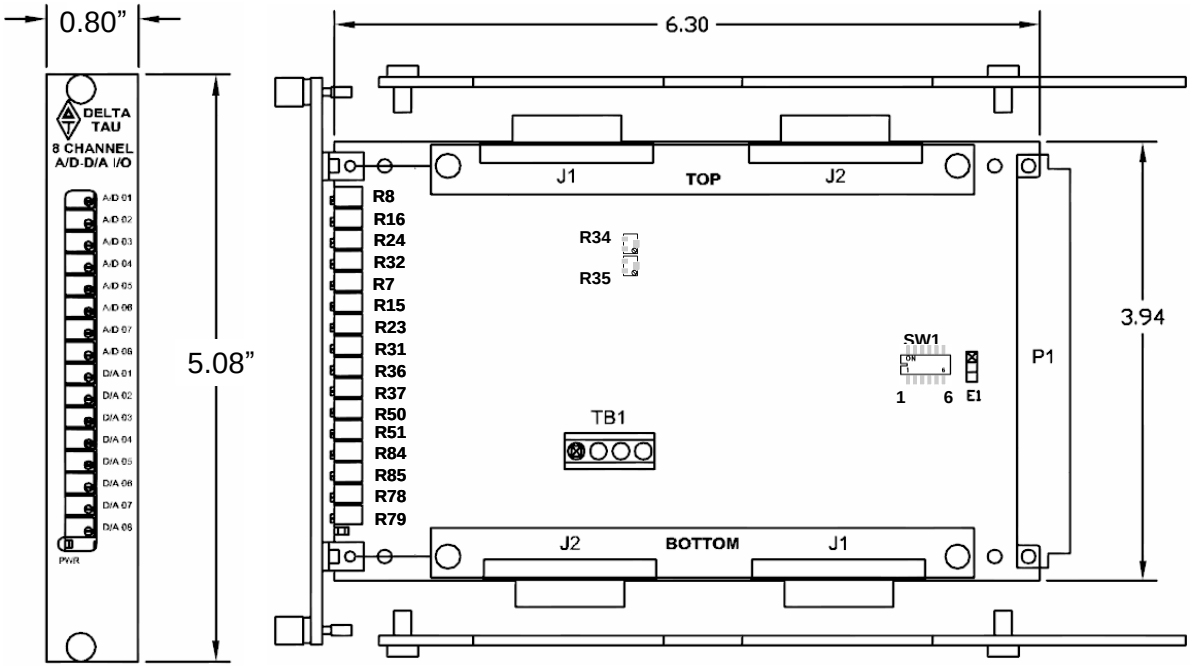
DB15 Option

Top View

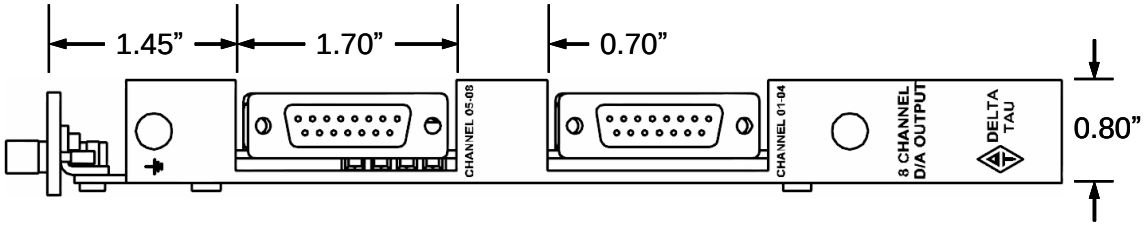


Front View

Side View



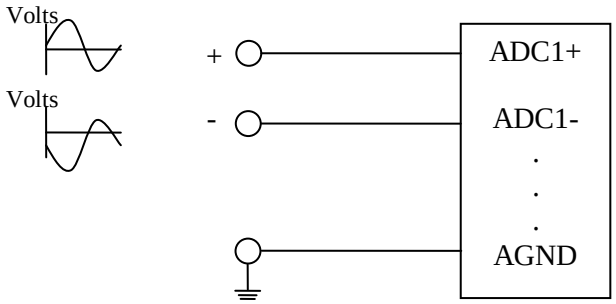
Bottom View



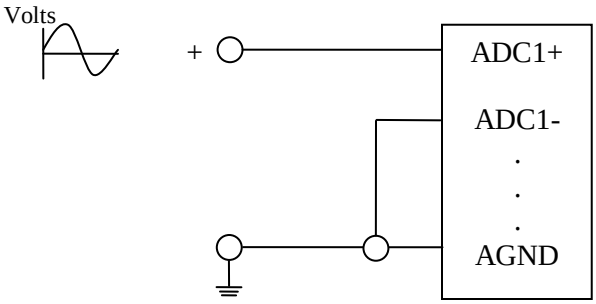
All dimensions are in inches.

Sample Wiring Diagram

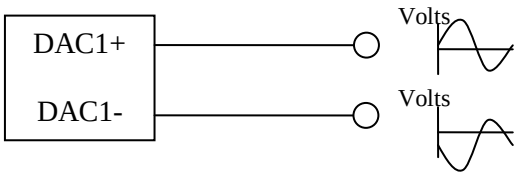
Differential Analog Input Signal



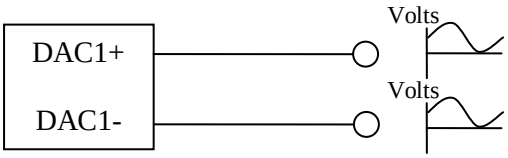
Single-Ended Analog Input Signal



Bipolar Analog Output Signal



Unipolar Analog Output Signal



P1: Backplane Bus

This connector is used to interface to the UMAC's processor via the 3U backplane bus. The signals brought in through this connector are buffered on board.

TB1: External Power Supply

Caution

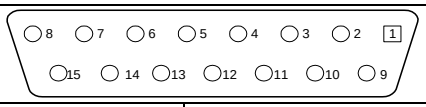
Do not use TB1 when the ACC-59E is plugged into the backplane.

This 4-pin terminal block provides the connection for an external power supply (standalone mode).

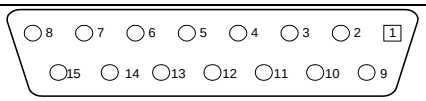
TB1: 4-Point Terminal Block			
Pin #	**Symbol**	**Function**	**Description**
1	GND	Common	Digital ground
2	+5V	Input	External supply
3	+15V	Input	External supply
4	-15V	Input	External supply

DB15 Breakout Option

J1 Top: ADC1 through ADC4

J1 Top: D-sub DA-15F Mating: D-sub DA-15M			
Pin #	**Symbol**	**Function**	**Description**
1	ADC1+	Input	Analog Input #1
2	ADC2+	Input	Analog Input #2
3	ADC3+	Input	Analog Input #3
4	ADC4+	Input	Analog Input #4
5			
6			
7	AGND	Common	Ground
8	-15V	Output	
9	ADC1-	Input	Analog Input #1/
10	ADC2-	Input	Analog Input #2/
11	ADC3-	Input	Analog Input #3/
12	ADC4-	Input	Analog Input #4/
13			
14			
15	+15V	Output	

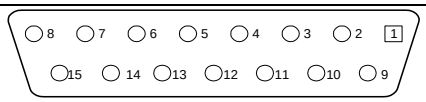
J1 Top: ADC5 through ADC8

J2 Top: D-sub DA-15F Mating: D-sub DA-15M			
Pin #	**Symbol**	**Function**	**Description**
1	ADC5+	Input	Analog Input #1
2	ADC6+	Input	Analog Input #2
3	ADC7+	Input	Analog Input #3
4	ADC8+	Input	Analog Input #4
5			
6			
7	AGND	Common	Ground
8	-15V	Output	Negative Supply
9	ADC5-	Input	Analog Input #1/
10	ADC6-	Input	Analog Input #2/
11	ADC7-	Input	Analog Input #3/
12	ADC8-	Input	Analog Input #4/
13			
14			
15	+15V	Output	Positive Supply

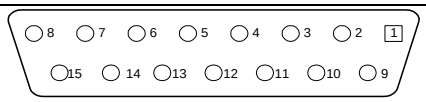
Note

- On-board fuses limit the drawn current to 0.5 Amperes.
- The common ground (pin #7) is tied to the digital ground of the UMAC rack.
- Tie the ADC- pin to ground if using single-ended wiring to ensure full resolution and proper operation of the ADC channel.

J1 Bottom: DAC1 through DAC4

J1 Bottom: D-sub DA-15F Mating: D-sub DA-15M			
Pin #	**Symbol**	**Function**	**Description**
1	DAC1+	Output	DAC Output #1
2	DAC2+	Output	DAC Output #2
3	DAC3+	Output	DAC Output #3
4	DAC4+	Output	DAC Output #4
5			
6			
7	AGND	Common	Ground
8			
9	DAC1-	Output	DAC Output #1/
10	DAC2-	Output	DAC Output #2/
11	DAC3-	Output	DAC Output #3/
12	DAC4-	Output	DAC Output #4/
13			
14			
15	+5V	Output	

J2 Bottom: DAC5 through DAC8

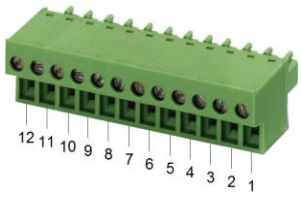
J2 Bottom: D-sub DA-15F Mating: D-sub DA-15M			
Pin #	**Symbol**	**Function**	**Description**
1	DAC5+	Output	DAC Output #1
2	DAC6+	Output	DAC Output #2
3	DAC7+	Output	DAC Output #3
4	DAC8+	Output	DAC Output #4
5			
6			
7	AGND	Common	Ground
8			
9	DAC5-	Output	DAC Output #1/
10	DAC6-	Output	DAC Output #2/
11	DAC7-	Output	DAC Output #3/
12	DAC8-	Output	DAC Output #4/
13			
14			
15	+5V	Output	

Note

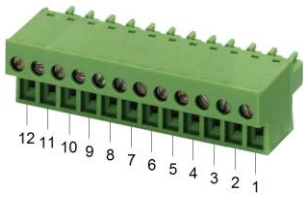
The common ground (pin #7) is tied to the digital ground of the UMAC rack.

Terminal Block Option

TB1 Top: ADC1 through ADC4

TB1 Top: 12-Point Terminal Block			
Pin #	**Symbol**	**Function**	**Description**
1	ADC1+	Input	Analog Input #1
2	ADC1-	Input	Analog Input #1/
3	ADC2+	Input	Analog Input #2
4	ADC2-	Input	Analog Input #2/
5	ADC3+	Input	Analog Input #3
6	ADC3-	Input	Analog Input #3/
7	ADC4+	Input	Analog Input #4
8	ADC4-	Input	Analog Input #4/
9	NC	NC	
10	NC	NC	
11	NC	NC	
12	NC	NC	

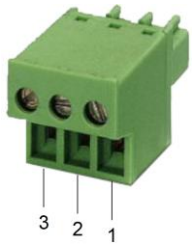
TB2 Top: ADC5 through ADC8

TB2 Top: 12-Point Terminal Block			
Pin #	**Symbol**	**Function**	**Description**
1	ADC5+	Input	Analog Input #5
2	ADC5-	Input	Analog Input #5/
3	ADC6+	Input	Analog Input #6
4	ADC6-	Input	Analog Input #6/
5	ADC7+	Input	Analog Input #7
6	ADC7-	Input	Analog Input #7/
7	ADC8+	Input	Analog Input #8
8	ADC8-	Input	Analog Input #8/
9	NC	NC	
10	NC	NC	
11	NC	NC	
12	NC	NC	

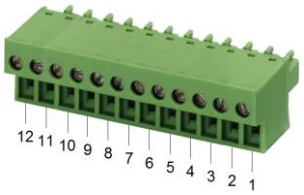
Note

Tie the ADC- pin to ground if using single-ended wiring to ensure full resolution and proper operation of the ADC channel.

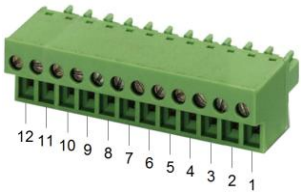
TB3 Top: Power Supply Outputs

TB3 Top: 3-Point Terminal Block				
Pin #	**Symbol**	**Function**	**Description**	**Notes**
1	AGND	Common	Common Reference for ADC1-16	Tied to UMAC's digital ground
2	+15V	Output	+15 V from UMAC Power Supply	Fused (0.5 A)
3	-15V	Output	-15 V from UMAC Power Supply	Fused (0.5 A)

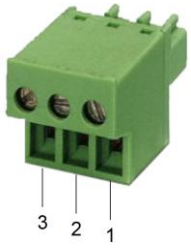
TB1 Bottom – DAC1 through DAC4

TB1 Bottom: 12-Point Terminal Block			
Pin #	**Symbol**	**Function**	**Description**
1	DAC1+	Output	DAC Output #1+
2	DAC1-	Output	DAC Output #1-
3	DAC2+	Output	DAC Output #2+
4	DAC2-	Output	DAC Output #2-
5	DAC3+	Output	DAC Output #3+
6	DAC3-	Output	DAC Output #3-
7	DAC4+	Output	DAC Output #4+
8	DAC4-	Output	DAC Output #4-
9	NC	NC	
10	NC	NC	
11	NC	NC	
12	NC	NC	

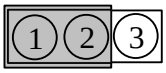
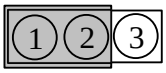
TB2 Bottom – DAC5 through DAC8

TB2 Bottom: 12-Point Terminal Block			
Pin #	**Symbol**	**Function**	**Description**
1	DAC5+	Output	DAC Output #5+
2	DAC5-	Output	DAC Output #5-
3	DAC6+	Output	DAC Output #6+
4	DAC6-	Output	DAC Output #6-
5	DAC7+	Output	DAC Output #7+
6	DAC7-	Output	DAC Output #7-
7	DAC8+	Output	DAC Output #8+
8	DAC8-	Output	DAC Output #8-
9	NC	NC	
10	NC	NC	
11	NC	NC	
12	NC	NC	

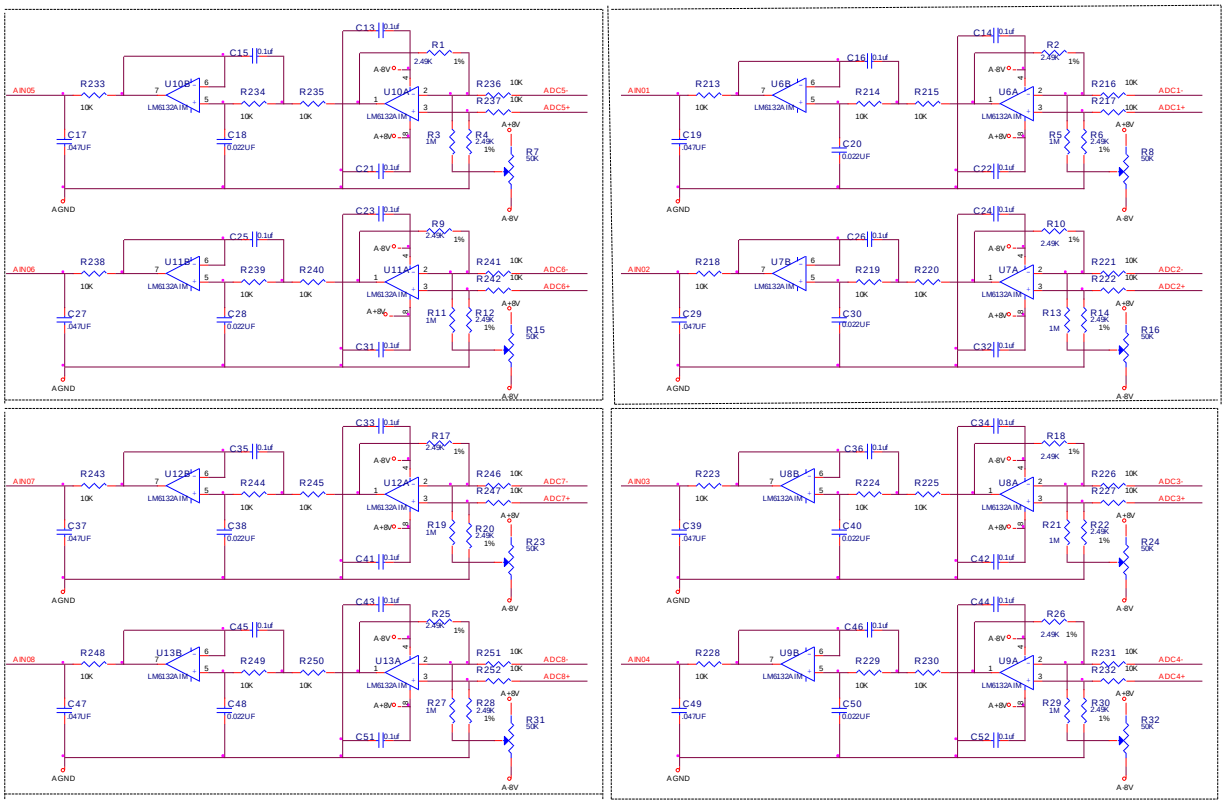
TB3 Bottom – Power Supply Outputs

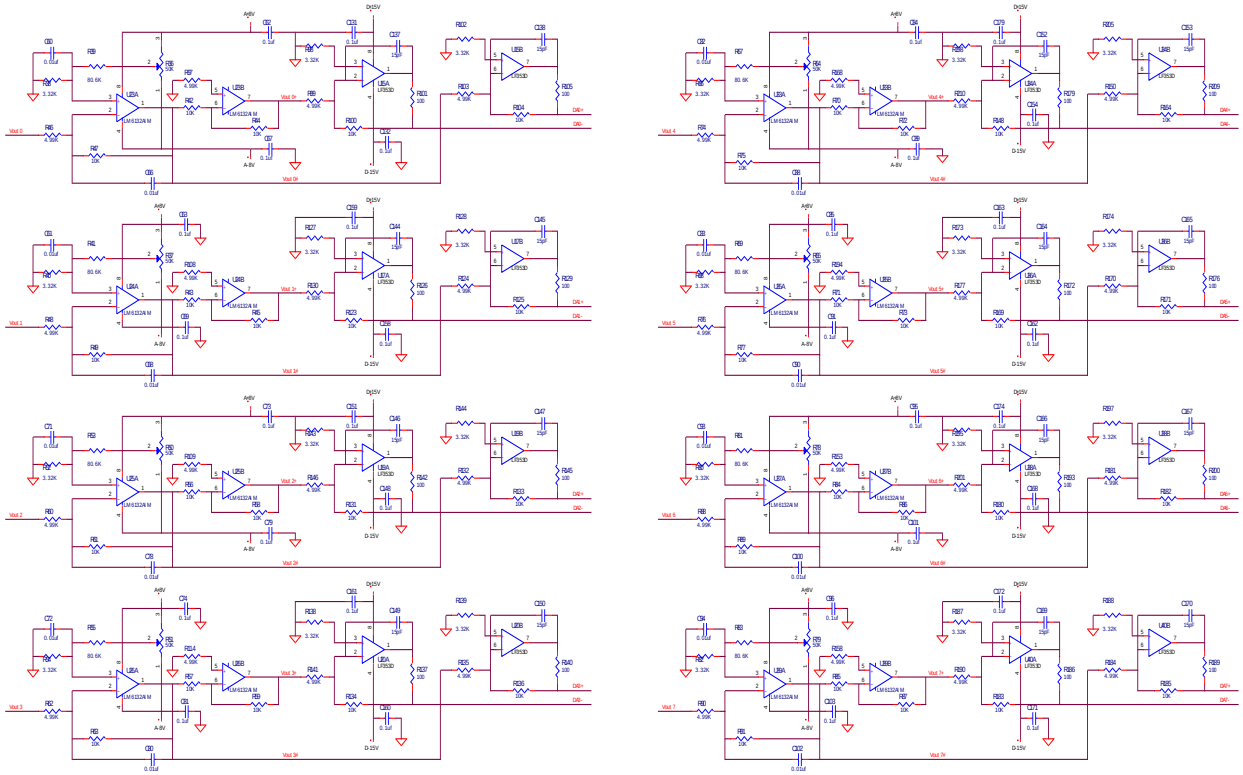
TB3 Bottom: 3-Point Terminal Block			
Pin #	**Symbol**	**Function**	**Description**
1	GND	Input/Output	Common for +5 V Output
2	+5V	Output	Digital Output
3	NC	NC	Do Not Connect

APPENDIX A: JUMPER SETTINGS

Jumper	Configuration	Default
E1 	Jump pins 1 to 2 for Turbo UMAC & MACRO Stations Rev. 104 and newer	Set by factory
	Jump pins 2 to 3 for Legacy MACRO Stations Rev. 103 and older	
J3 	Jump pins 1 to 2 for Bipolar DAC Outputs	Pins 2 to 3 Jumped
	Jump pins 2 to 3 for Unipolar DAC Outputs	

APPENDIX B: SCHEMATICS





APPENDIX C: USING POINTERS

Below are alternate methods for accessing the data structures of Power PMAC in order to read ADCs and write to DACs using on the ACC-59E.

Manual ADC Read Using Pointers

The manual read method with pointers consists of selecting the desired channel with an I/O pointer, reading it with another pointer, and then storing it into local memory. This procedure can be implemented in a PLC script to read all channels consecutively and consistently, creating a "custom automatic" function.

Following are the necessary steps for implementing the manual ADC read method (with the example of an ACC-59E at I/O base address offset $A00000) to allow the user to choose to read unipolar or bipolar input modes:

1. Point an available I/O pointer (12-bit wide) to bits 8-19 of the ACC-59E I/O base address offset. This is the "**Data Read**" register of the selected channel. It can be defined as follows:

   Unsigned (positive input voltage only) Data Read pointer for Unipolar Mode:

```
ptr UnipolarDataRead->u.io:$A00000.8.12;
```

   Signed (negative or positive input voltage) Data Read pointer for Bipolar Mode:

```
ptr BipolarDataRead->s.io:$A00000.8.12;
```

2. Point an available M-Variable (24-bit wide unsigned) to the base address of the ACC-59E. This is the **"Channel Select"** Pointer.

```
ptr ChannelSelect->u.io:$A00000.8.24;
```

3. Point an available M-Variable (1-bit wide unsigned) to the ADC Ready bit of the ACC-59E. This is the "**ADC Ready**" Pointer. This bit becomes 1 when the ADC conversion has finished.

```
ptr ADCReady->u.io:$D00180.13.1;
```

   The ADC Ready bit is bit 13 of the ADC Ready Offset shown in the following table:

Index (n)	Base Offset	ADC Ready Offset
0	$A00000	$D00180
4	$A08000	$D08180
8	$A10000	$D10180
12	$A18000	$D18180
1	$B00000	$D00190
5	$B08000	$D08190
9	$B10000	$D10190
13	$B18000	$D18190
2	$C00000	$D001C0
6	$C08000	$D081C0
10	$C10000	$D101C0
14	$C18000	$D181C0
3	$D00000	$D001D0
7	$D08000	$D081D0
11	$D10000	$D101D0
15	$D18000	$D181D0

4. Using the **Channel Select** Pointer, specify the **voltage mode** for each ADC# desired:

   Set ***UnipolarDataRead*** = ADC# - 1 for Unipolar Inputs
   Set ***BipolarDataRead*** = ADC# + 7 for Bipolar Inputs

	Channel Select Pointer Value	
ADC#	**Unipolar Inputs**	**Bipolar Inputs**
1	0	8
2	1	9
3	2	10
4	3	11
5	4	12
6	5	13
7	6	14
8	7	15

5. Wait for the **ADC Ready** bit to become 1, and then read and/or copy the data contained in the **Data Read** register(s) described above.

ADC Manual Read Example Script PLCs

Ultimately, the above procedure can be implemented in a PLC script to read all channels consecutively and consistently, creating a "custom automatic" function. To set up Power UMAC with an ACC-59E at I/O base address offset $A00000, see the following example.

Unipolar Script PLC Example

This example selects and reads channels 1 through 8 successively as unipolar in a PLC and stores the results for channels 1 through 8 in global variables.

```
ptr DataRead->u.io:$A00000.8.12;            // Data Read register, unsigned for unipolar
ptr ChSelect->u.io:$A00000.8.24;            // Channel Select Pointer
ptr ADCReady->u.io:$D00180.13.1;            // ADC Ready Bit

global ADC1;          // Channel 1 ADC storage variable
global ADC2;          // Channel 2 ADC storage variable
global ADC3;          // Channel 3 ADC storage variable
global ADC4;          // Channel 4 ADC storage variable
global ADC5;          // Channel 5 ADC storage variable
global ADC6;          // Channel 6 ADC storage variable
global ADC7;          // Channel 7 ADC storage variable
global ADC8;          // Channel 8 ADC storage variable

Open PLC 1
ChSelect = 0;         // Select Channel 1 (ChSelect = ADC#-1)
While(ADCReady != 1){}        // Wait for ADC to finish conversion
ADC1 = DataRead;              // Read and copy result into storage

ChSelect = 1;         // Select Channel 2 (ChSelect = ADC#-1)
While(ADCReady != 1){}        // Wait for ADC to finish conversion
ADC2 = DataRead;              // Read and copy result into storage

ChSelect = 2;         // Select Channel 3 (ChSelect = ADC#-1)
While(ADCReady != 1){}        // Wait for ADC to finish conversion
ADC3 = DataRead;              // Read and copy result into storage

ChSelect = 3;         // Select Channel 4 (ChSelect = ADC#-1)
While(ADCReady != 1){}        // Wait for ADC to finish conversion
ADC4 = DataRead;              // Read and copy result into storage

ChSelect = 4;         // Select Channel 5 (ChSelect = ADC#-1)
While(ADCReady != 1){}        // Wait for ADC to finish conversion
ADC5 = DataRead;              // Read and copy result into storage

ChSelect = 5;         // Select Channel 6 (ChSelect = ADC#-1)
While(ADCReady != 1){}        // Wait for ADC to finish conversion
ADC6 = DataRead;              // Read and copy result into storage

ChSelect = 6;         // Select Channel 7 (ChSelect = ADC#-1)
While(ADCReady != 1){}        // Wait for ADC to finish conversion
ADC7 = DataRead;              // Read and copy result into storage

ChSelect = 7;         // Select Channel 8 (ChSelect = ADC#-1)
While(ADCReady != 1){}        // Wait for ADC to finish conversion
ADC8 = DataRead;              // Read and copy result into storage
Close
```

Bipolar Script PLC Example

This example selects and reads channels 1 through 8 successively as bipolar in a PLC and stores the results for channels 1 through 8 in global variables.

```
ptr DataRead->s.io:$A00000.8.12;             // Data Read register, signed for bipolar
ptr ChSelect->u.io:$A00000.8.24;             // Channel Select Pointer
ptr ADCReady->u.io:$D00180.21.1;             // ADC Ready Bit

global ADC1;           // Channel 1 ADC storage variable
global ADC2;           // Channel 2 ADC storage variable
global ADC3;           // Channel 3 ADC storage variable
global ADC4;           // Channel 4 ADC storage variable
global ADC5;           // Channel 5 ADC storage variable
global ADC6;           // Channel 6 ADC storage variable
global ADC7;           // Channel 7 ADC storage variable
global ADC8;           // Channel 8 ADC storage variable

Open PLC 1
ChSelect = 8;          // Select ADC Channel 1 (ChSelect = ADC#+7)
While(ADCReady != 1){}         // Wait for ADC to finish conversion
ADC1 = DataRead;               // Read and copy result into storage

ChSelect = 9;          // Select ADC Channel 2 (ChSelect = ADC#+7)
While(ADCReady != 1){}         // Wait for ADC to finish conversion
ADC2 = DataRead;               // Read and copy result into storage

ChSelect = 10;         // Select ADC Channel 3 (ChSelect = ADC#+7)
While(ADCReady != 1){}         // Wait for ADC to finish conversion
ADC3 = DataRead;               // Read and copy result into storage

ChSelect = 11;         // Select ADC Channel 4 (ChSelect = ADC#+7)
While(ADCReady != 1){}         // Wait for ADC to finish conversion
ADC4 = DataRead;               // Read and copy result into storage

ChSelect = 12;         // Select ADC Channel 5 (ChSelect = ADC#+7)
While(ADCReady != 1){}         // Wait for ADC to finish conversion
ADC5 = DataRead;               // Read and copy result into storage

ChSelect = 13;         // Select ADC Channel 6 (ChSelect = ADC#+7)
While(ADCReady != 1){}         // Wait for ADC to finish conversion
ADC6 = DataRead;               // Read and copy result into storage

ChSelect = 14;         // Select ADC Channel 7 (ChSelect = ADC#+7)
While(ADCReady != 1){}         // Wait for ADC to finish conversion
ADC7 = DataRead;               // Read and copy result into storage

ChSelect = 15;         // Select ADC Channel 8 (ChSelect = ADC#+7)
While(ADCReady != 1){}         // Wait for ADC to finish conversion
ADC8 = DataRead;               // Read and copy result into storage
Close
```

Unipolar and Bipolar CPLC Example

This example assumes one ACC-59E set at base offset $A00000 with unipolar inputs and one set at $B00000 with bipolar inputs. The CPLC samples all 8 channels of the unipolar card, then all 8 channels of the bipolar card. This CPLC incorporates two functions, ***ACC59E_ADC*** and ***ACC59E_WaitForADC***, which take user parameters about the card index, pair selection, and signal polarity, and then return ADC results through pointers. The benefit of using these functions is that the user does not need to program in the actual card base address, but only the corresponding index (***n***); PMAC will then determine the appropriate addressing automatically. These functions bypass the ACC59E structures and access the memory addresses directly by using ***Sys.OffsetCardIO[n]*** to get the card base offset and ***Sys.OffsetCardIOCid[n]*** to get the ADC Ready offset. Please read the comments under the function definitions for more details on how to use the functions.

WARNING

When using C routines, failure to release control of the loop waiting for the ADC conversions to finish may cause PMAC to lock up, possibly creating a runaway condition. Thus, one of the following functions, ***WaitForADC***, has precautions in order to release control of the wait loop in the event that the ADC conversion bits never become 1.

```
#include <gplib.h>
#include <stdio.h>
#include <dlfcn.h>

// Definition(s)
#define Card1Index     0       // For base offset $A00000
#define Card2Index     1       // For base offset $B00000
#define Unipolar_Code  0       // For unipolar input signals
#define Bipolar_Code   1       // For bipolar input signals

// Prototype(s)
int ACC59E_ADC(unsigned int Card_Index, unsigned int ADC_Channel,unsigned int Polarity,
int *ADC_Result);
int ACC59E_WaitForADC(unsigned int Card_Index);

void user_plcc(){
        unsigned int ADC_Resultu[8],index,Channel_Number;
        int ADC_Results[8],ErrorCode,ADC_Result;
        for(index = 0; index < 8; index++){
                Channel_Number = index + 1;
                // Request ADC result, unsigned
                ErrorCode = ACC59E_ADC(Card1Index,Channel_Number,Unipolar_Code,&ADC_Result);
                if(ErrorCode < 0){
                        return; // error
                }
                ADC_Resultu[index] = ADC_Result;      // Store result in array
                // Request ADC result, signed
                ErrorCode = ACC59E_ADC(Card2Index,Channel_Number,Bipolar_Code,&ADC_Result);
                if(ErrorCode < 0){
                        return; // error
                }
                ADC_Results[index] = ADC_Result;      // Store result in array
                // Store results in P-Variables so that motion programs
                // can access them (optional)
                pshm->P[5000 + index] = ADC_Resultu[index];
                pshm->P[6000 + index] = ADC_Results[index];
        }
        return;
}
```

```
int ACC59E_ADC(unsigned int Card_Index, unsigned int ADC_Channel,unsigned int Polarity,
 int *ADC_Result)
{       // Returns the ADC result of the ADC channel specified.
        /*Inputs:
        ADC_Channel: The number of the ADC Channel desired
        Polarity:       0 = unipolar, 1 = bipolar
        Timeout:        Timeout duration (milliseconds) to wait for ADC conversion to finish. The
        function will break if this is exceeded.
        Outputs:
        return 0 if ADC conversion was successful; stores the ADC result in *ADC_Result if the
        conversion was successful
        return -1 if user entered invalid Card_Index
        return -2 if user entered invalid ADC_Channel
        return -3 if user entered invalid Polarity
        return -4 if ADC conversion timed out*/
        unsigned int Address,BaseOffset,ConvertCode;
        int WaitResult;
        volatile unsigned int *pACC59E_ADC_ChSelect; // Unsigned for bipolar input signals
        volatile unsigned int *pACC59E_Data_Read_Unipolar;   // Signed for bipolar input signals
        volatile int *pACC59E_Data_Read_Bipolar;
        if(Card_Index < 0 || Card_Index > 15){
                return -1;
        }
        if((ADC_Channel < 1) || (ADC_Channel > 8)){
                return -2;
        }
        if(Polarity != 0 && Polarity != 1)
        {
                return -3;
        }
        if(Polarity == 0)                               // Unipolar input signal
        {
                ConvertCode = ADC_Channel - 1;
        } else {                                        // Bipolar input signal
                ConvertCode = ADC_Channel + 7;
        }
        BaseOffset = pshm->OffsetCardIO[Card_Index];
        if(BaseOffset == 0){
                return -1;
        }
        Address = (unsigned int)piom + BaseOffset/4;
        pACC59E_ADC_ChSelect = (volatile unsigned int *)Address;
        // Shift and mask to write the convert code to the correct place in the ADC word
        *pACC59E_ADC_ChSelect = ((ConvertCode) << 8) & 0xFFFFFF00;
        // Wait for the ADC to finish converting
        WaitResult = ACC59E_WaitForADC(Card_Index);
        if(WaitResult == -1)                            // If the ADC conversion timed out
        {
                return -4;                              // Return with error code
        } else {                                        // Otherwise return ADC result
                if(Polarity == 0){
                        pACC59E_Data_Read_Unipolar = (volatile unsigned int*)Address;
                        // Shift and cast to get just the ADC result with the proper sign
                        *ADC_Result = ((unsigned int)(((*pACC59E_Data_Read_Unipolar) << 12) >> 20));

                } else {
                        pACC59E_Data_Read_Bipolar = (volatile int*)Address;
                        // Shift and cast to get just the ADC result with the proper sign
                        *ADC_Result = ((int)(((*pACC59E_Data_Read_Bipolar) << 12) >> 20));
                }
        }
        return 0;
}
```

```
int ACC59E_WaitForADC(unsigned int Card_Index)
{       // Waits until ADC conversions have completed

        // Inputs:
        // Card_Index: index (n) from POWER section of Addressing ACC-59E table

        // Outputs:
        // return 0 if successfully performed ADC conversion
        // return -1 if conversion did not complete within Timeout ms
        volatile unsigned int *pRdy;
        unsigned int Rdy = 0,iterations = 0;
        double Present_Time,Conversion_Start_Time,Time_Difference,Timeout,Timeout_us;
        struct timespec SleepTime={0};
        SleepTime.tv_nsec=1000000;
        // Get time at (almost) start of conversion (microseconds)
        // Timeout: Maximum permitted time to wait for ADC conversion to
        Conversion_Start_Time = GetCPUClock();
        // finish before error (milliseconds)
        Timeout = 500; // Milliseconds
        Timeout_us = Timeout*1000;                              // Convert to microseconds
        pRdy = piom + (pshm->OffsetCardIOCid[Card_Index]/4); // Point to ADC ready bit
        do
        {
                // If the loop has taken a multiple of 50 iterations to finish
                if(iterations == 50){
                        // Release control for 1 ms so PMAC does not go into
                        // Watchdog mode while waiting for conversion to finish
                        nanosleep(&SleepTime,NULL); // Release thread and wait 1 msec
                        iterations = 0;         // Reset iteration counter
                }
                Present_Time = GetCPUClock(); // Obtain current system time
                // Compute difference in time between starting conversion and now
                Time_Difference = Present_Time-Conversion_Start_Time;
                if(Time_Difference > Timeout_us)// If more than Timeout ms have elapsed
                {
                        return (-1);            // Return with error code
                }
                // Shift and cast to get just 13th bit (the ADC ready bit)
                Rdy = (unsigned int)((*pRdy << 18) >> 31);
                iterations++;
        }       while(Rdy != 1);        // Test ADC ready bit
        return 0;                       // Return with success code
}
```

DAC Output Using Pointers

For each DAC channel, one can point an unsigned integer global Script I/O pointer or C pointer to the appropriate address location and 12-bit range as follows:

DAC #	DAC Address Offset	12-bit Location
1	Base Offset + $40	[19:8]
2	Base Offset + $44	[19:8]
3	Base Offset + $48	[19:8]
4	Base Offset + $4C	[19:8]

DAC #	DAC Address Offset	12-bit Location
5	Base Offset + $40	[31:20]
6	Base Offset + $44	[31:20]
7	Base Offset + $48	[31:20]
8	Base Offset + $4C	[31:20]

The Base Offset is given in the Addressing ACC-59E section of this manual.

Script I/O Pointers

Example:

Setting up the analog outputs with Script I/O pointers for an ACC-59E set at base offset $A00000:

```
ptr DAC1->u.io:$A00040.8.12;  // DAC Output #1
ptr DAC2->u.io:$A00044.8.12;  // DAC Output #2
ptr DAC3->u.io:$A00048.8.12;  // DAC Output #3
ptr DAC4->u.io:$A0004C.8.12;  // DAC Output #4
ptr DAC5->u.io:$A00040.20.12; // DAC Output #5
ptr DAC6->u.io:$A00044.20.12; // DAC Output #6
ptr DAC7->u.io:$A00048.20.12; // DAC Output #7
ptr DAC8->u.io:$A0004C.20.12; // DAC Output #8
```

Example: Using DAC Pointers in Global Definitions.pmh

Configuring a single ACC-59E set at base offset $A00000 with unipolar differential outputs.

```
// Assumes unipolar differential outputs
DAC1 = 0;      // DAC Channel 1 Outputs 0.0 V
DAC2 = 511;    // DAC Channel 2 Outputs 2.5 V
DAC3 = 1023;   // DAC Channel 3 Outputs 5.0 V
DAC4 = 1535;   // DAC Channel 4 Outputs 7.5 V
DAC5 = 2047;   // DAC Channel 5 Outputs 10.0 V
DAC6 = 2559;   // DAC Channel 6 Outputs 12.5 V
DAC7 = 3071;   // DAC Channel 7 Outputs 15.0 V
DAC8 = 4095;   // DAC Channel 8 Outputs 20.0 V
```

Using C Pointers (Optional; For C Programmers)

To access the DAC channels directly using C code without using the ACC-59E structures, point `volatile unsigned int*` pointers to the same addresses to which the Script I/O pointers point in the above section and write thereto using software counts just like with Script I/O pointers. See the following example.

DAC Output CPLC with Direct Memory Access Example

This PLC commands DAC channels 1 – 8 to output 0, 2, 5, 8, 11, 14, 17, and 20 volts, respectively, and then disables itself. The commanded output voltage is selected with the `Output_Voltage` variable.

```
#include <RtGpShm.h>
#include <stdio.h>
#include <dlfcn.h>

// Definition(s)
#define CPLC_Number           5       // User can adjust this number to match his CPLC number
#define Single_Ended_Wiring   0
#define Differential_Wiring   1
#define Unipolar_Mode         0
#define  Bipolar_Mode         1

// Prototype(s)
int ACC59E_DAC_Output_Word(unsigned int DAC_Channel, double Desired_Output_Voltage,
                        int Polarity, int Wiring_Mode, volatile unsigned int pACC59E_DAC);

void user_plcc()
{
        // Define array of pointers to contain pointers to DAC channel addresses
        volatile unsigned int pACC59E_DAC_Address[8],Wiring_Mode,DAC_Mode,DAC_Channel;
        unsigned int Array_Index;
        int ErrorCode;
        double Output_Voltage = 0;
        // Allocate memory for 8 pointers in the array for the 8 DAC channels
        // Assign the pointers to the appropriate DAC channel addresses
        pACC59E_DAC_Address[0]  = ((volatile unsigned int)piom + 0xA00040/4); // DAC Channel 1
        pACC59E_DAC_Address[1]  = ((volatile unsigned int)piom + 0xA00044/4); // DAC Channel 2
        pACC59E_DAC_Address[2]  = ((volatile unsigned int)piom + 0xA00048/4); // DAC Channel 3
        pACC59E_DAC_Address[3]  = ((volatile unsigned int)piom + 0xA0004C/4); // DAC Channel 4
        pACC59E_DAC_Address[4]  = ((volatile unsigned int)piom + 0xA00040/4); // DAC Channel 5
        pACC59E_DAC_Address[5]  = ((volatile unsigned int)piom + 0xA00044/4); // DAC Channel 6
        pACC59E_DAC_Address[6]  = ((volatile unsigned int)piom + 0xA00048/4); // DAC Channel 7
        pACC59E_DAC_Address[7]  = ((volatile unsigned int)piom + 0xA0004C/4); // DAC Channel 8

        // Assign output values in software counts
        for (Array_Index = 0; Array_Index < 8; Array_Index++)
        {
                Output_Voltage = Array_Index*20/7;    // Compute desired output voltage
                DAC_Channel = Array_Index + 1;
                ErrorCode = ACC59E_DAC_Output_Word(DAC_Channel, Output_Voltage,Unipolar_Mode,
                Differential_Wiring, pACC59E_DAC_Address[Array_Index]);
                if(ErrorCode < 0)
                {
                        return; // error
                }
        }
        pshm->UserAlgo.BgCplc[CPLC_Number] = enum_threaddisable;    // Disable this CPLC
        return;
}
```

```
int ACC59E_DAC_Output_Word(unsigned int DAC_Channel, double Desired_Output_Voltage,
                        int Polarity, int Wiring_Mode, volatile unsigned int pACC59E_DAC_Address)
                                {
/* Returns the appropriate 32-bit word to write to the DAC channel address.
Inputs:
DAC_Channel:                    Desired DAC channel to which to output (integers, 1 to 8)
Desired_Output_Voltage:         Desired voltage to output in units of Volts
Polarity:                       DAC output mode set by jumper J3 (0 = unipolar, 1 = bipolar)
Wiring_Mode:                    DAC output wiring mode (0 = single-ended, 1 = differential)
pACC59E_DAC_Address:            Address of the DAC channel selected
Outputs:
return 0 and write the correct word to write to the DAC channel address to *pACC59E_DAC, if everything
went correctly
return -1 if DAC_Channel invalid
return -2 if Desired_Output_Voltage was invalid and was subsequently truncated, but truncated value was
still written to DAC channel
return -3 if Polarity invalid
return -4 if Wiring_Mode invalid
Notes:
This function will truncate the desired output voltage to correspond with
physical card limits.  */
        // DAC_Word:            The entire current 32-bit word at the DAC channel address
        volatile unsigned int Software_Counts = 0,DAC_Word;
        volatile unsigned int *pACC59E_DAC;
        int Software_Count_Range = 0, Max_Amplitude = 0,Software_Count_Offset = 0, Return_Value = 0;
        double Conversion_Factor = 0;
        pACC59E_DAC = (volatile unsigned int*)pACC59E_DAC_Address;
        // Check whether inputs to function are correct
        if(DAC_Channel < 1 || DAC_Channel > 8)
        {
                return -1;
        }
        if((Polarity != 0 && Polarity != 1))
        {
                return -3;
        }
        if((Wiring_Mode != 0 && Wiring_Mode != 1))
        {
                return -4;
        }
        if (Wiring_Mode == 0){                                  // Single-Ended
                Max_Amplitude = 10;                             // Volts
                if(Desired_Output_Voltage > 10)                 // Saturate voltages at card limits
                {
                        Desired_Output_Voltage = 10;
                        Return_Value = -2;
                }
        }
        else {                                                  // Differential
                Max_Amplitude = 20;                             // Volts
                if(Desired_Output_Voltage > 20)                 // Saturate voltages at card limits
                {
                        Desired_Output_Voltage = 20;
                        Return_Value = -2;
                }
        }
        if (Polarity == 0){                                     // Unipolar
                Software_Count_Range = 4095;
                Software_Count_Offset = 0;
                if(Desired_Output_Voltage < 0)                  // Saturate voltages at card limits
                {
                        Desired_Output_Voltage = 0;
                        Return_Value = -2;
                }
        }
        else{                                                   // Bipolar
                Software_Count_Range = 2047;
                Software_Count_Offset = 2047;
                if(Wiring_Mode == 0)
                {
                        if(Desired_Output_Voltage < (-10))      // Saturate voltages at card limits
```

```
                        {
                                Desired_Output_Voltage = (-10);
                                Return_Value = -2;
                        }
                } else {
                        if(Desired_Output_Voltage < (-20))    // Saturate voltages at card limits
                        {
                                Desired_Output_Voltage = (-20);
                                Return_Value = -2;
                        }
                }
        }
        // Conversion factor in units of counts/volt:
        Conversion_Factor = (double)((double)(Software_Count_Range)/(double)(Max_Amplitude));
        Software_Counts = (volatile unsigned int)((double)Software_Count_Offset +
                                                Desired_Output_Voltage*Conversion_Factor);
        DAC_Word = *pACC59E_DAC;
        if (DAC_Channel <= 4){                                          // Channels 1-4
                *pACC59E_DAC = (((Software_Counts << 8)  & 0xFFFFFF00)^(DAC_Word & 0xFFF00000));

        }
        else {                                                          // Channels 5-8
                *pACC59E_DAC = (((Software_Counts << 20) & 0xFFFFFF00)^(DAC_Word & 0x000FFF00));

        }
        return Return_Value;
}
```

DAC Output Using DAC C Functions

This example assumes a single ACC-59E set at base offset $B00000. The CPLC commands 5.0 volts to be the output on channels 1-8 and then disables itself, leaving the channels activated. The CPLC uses the function ACC59E_DAC, with subfunctions ***ACC59E_Get_DAC_Channel_Address*** and ***ACC59E_DAC_Output_Word***. ***ACC59E_DAC*** only requires input from the user for the card index number ***n*** from the POWER section of the Addressing ACC-59E section of this manual, the DAC channel number desired, the desired output voltage, the polarity of the signal (unipolar/bipolar), and the wiring mode (single-ended/differential). ACC59E_Get_ADC_Channel_Address uses ***Sys.OffsetCardIO[n]*** to determine the card base offset, so the user does not need to program in the card address manually, only the card index. See the comments under the function definitions for details on how to use the functions.

```
#include <RtGpShm.h>
#include <stdio.h>
#include <dlfcn.h>

// Definition(s)
#define CardIndex                       0       // For base offset $A00000
#define CPLC_Number                     1       // The user can adjust this to match
                                                // the number of his CPLC
#define Output_Voltage_Constant         5.0     // Volts
#define Unipolar_Code                   0       // For unipolar outputs
#define Bipolar_Code                    1       // For bipolar outputs
#define Single_Ended_Code               0       // For single-ended outputs
#define Differential_Code               1       // For differential outputs

// Prototype(s)
int ACC59E_Get_DAC_Channel_Address(unsigned int Card_Index,unsigned int DAC_Channel, volatile
                                    unsigned int *pACC59E);
int ACC59E_DAC_Output_Word(unsigned int DAC_Channel, double Desired_Output_Voltage,
                        int Polarity, int Wiring_Mode, volatile unsigned int *pACC59E_DAC);
int ACC59E_DAC(unsigned int Card_Index, unsigned int DAC_Channel, double Desired_Output_Voltage,
                                    unsigned int Polarity, unsigned int Wiring_Mode);

void user_plcc()
{
        int WriteResult,Channel_Index;
        for(Channel_Index = 1; Channel_Index < 9; Channel_Index++)
        {
                // Set this channel to 5.0 volts unipolar differential
                WriteResult = ACC59E_DAC(CardIndex,Channel_Index,Output_Voltage_Constant,
                                Unipolar_Code,Differential_Code);
                if(WriteResult < 0)
                {
                        return; // error
                }
        }
        pshm->UserAlgo.BgCplc[CPLC_Number] = enum_threaddisable;    // Disable this CPLC
        return;
}
```

```
int ACC59E_DAC(unsigned int Card_Index, unsigned int DAC_Channel, double Desired_Output_Voltage,
unsigned int Polarity, unsigned int Wiring_Mode)
{       // Output Desired_Output_Voltage to the DAC_Channel specified.
/*Inputs:
Card_Index: Index of the card based on the address determined by SW1 settings
DAC_Channel:                    Desired DAC channel to which to output (integers, 1 to 8)
Desired_Output_Voltage:         Desired voltage to output in units of Volts
DAC_Word:                       The entire current 32-bit word at the DAC channel address
Polarity:                       DAC output mode set by jumper J3 (0 = unipolar, 1 = bipolar)
Wiring_Mode:                    DAC output wiring mode (0 = single-ended, 1 = differential)
Outputs:
return 0 indicates that the function successfully wrote to DAC channel
return -1 if user entered invalid Card_Index
return -2 if user entered invalid DAC_Channel
return -3 if user entered invalid Polarity
return -4 if user entered invalid Wiring_Mode
return -5 indicates that Desired_Output_Voltage was invalid but truncated and still written to DAC
channel
return -6 indicates that parameters were entered correctly but the PMAC could not write to the DAC
channel successfully.*/
        volatile unsigned int *pACC59E;
        int ErrorCode;
        if(Card_Index < 0 || Card_Index > 15)
        {
                return -1;
        }
        if(DAC_Channel < 1 || DAC_Channel > 16)
        {
                return -2;
        }
        if(Polarity != 0 && Polarity != 1)
        {
                return -3;
        }
        if(Wiring_Mode != 0 && Wiring_Mode != 1)
        {
                return -4;
        }
        ErrorCode = ACC59E_Get_DAC_Channel_Address(Card_Index,DAC_Channel,pACC59E);
        if(ErrorCode < 0)
        {
                return -1;
        }
        ErrorCode = ACC59E_DAC_Output_Word(DAC_Channel, Desired_Output_Voltage, Polarity,
                                                Wiring_Mode, pACC59E);
        switch(ErrorCode)
        {
                case 0:
                        return 0;
                case -2:
                        return -5;
                default:
                        return -6;
                        break;
        }
}
```

```
int ACC59E_Get_DAC_Channel_Address(unsigned int Card_Index,unsigned int DAC_Channel, volatile
 unsigned int *pACC59E)
{       // Computes the appropriate address of the desired DAC Channel

/*Inputs:
Card_Index:     Index of the card based on the address determined by SW1 settings
DAC_Channel:    Number of the DAC Channel desired
*pACC59E:       Address of the DAC channel selected

Outputs:
return -1 if the address was not successfully computed
return 0 and set pACC59E to the appropriate address if the address was successfully computed*/
        unsigned int index,BaseOffset = 0,ChannelOffset;
        BaseOffset = pshm->OffsetCardIO[Card_Index];
        if(BaseOffset == 0)
        {
                return -1;
        }
        switch(DAC_Channel)
        {
                case 1:
                        ChannelOffset = BaseOffset + 0x40;
                        break;
                case 2:
                        ChannelOffset = BaseOffset + 0x44;
                        break;
                case 3:
                        ChannelOffset = BaseOffset + 0x48;
                        break;
                case 4:
                        ChannelOffset = BaseOffset + 0x4C;
                        break;
                case 5:
                        ChannelOffset = BaseOffset + 0x40;
                        break;
                case 6:
                        ChannelOffset = BaseOffset + 0x44;
                        break;
                case 7:
                        ChannelOffset = BaseOffset + 0x48;
                        break;
                case 8:
                        ChannelOffset = BaseOffset + 0x4C;
                        break;
                default:
                        return 0;
        }
        pACC59E = piom + ChannelOffset/4;
        return 0;
}
```

```
int ACC59E_DAC_Output_Word(unsigned int DAC_Channel, double Desired_Output_Voltage,
                        int Polarity, int Wiring_Mode, volatile unsigned int *pACC59E_DAC)
{
/* Returns the appropriate 32-bit word to write to the DAC channel address.

Inputs:
DAC_Channel:                    Desired DAC channel to which to output (integers, 1 to 8)
Desired_Output_Voltage:         Desired voltage to output in units of Volts
Polarity:                               DAC output mode set by jumper J3 (0 = unipolar, 1 = bipolar)
Wiring_Mode:                    DAC output wiring mode (0 = single-ended, 1 = differential)
*pACC59E_DAC:                   Address of the DAC channel selected

Outputs:
return 0 and write the correct word to write to the DAC channel address to *pACC59E_DAC, if everything
went correctly
return -1 if DAC_Channel invalid
return -2 if Desired_Output_Voltage was invalid and was subsequently truncated, but truncated value was
still written to DAC channel
return -3 if Polarity invalid
return -4 if Wiring_Mode invalid
Notes:
This function will truncate the desired output voltage to correspond with
physical card limits.  */
        // DAC_Word:            The entire current 32-bit word at the DAC channel address
        volatile unsigned int Software_Counts = 0,DAC_Word;
        int Software_Count_Range = 0, Max_Amplitude = 0,Software_Count_Offset = 0, Return_Value = 0;
        double Conversion_Factor = 0;

        // Check whether inputs to function are correct
        if(DAC_Channel < 1 || DAC_Channel > 8)
        {
                return -1;
        }
        if((Polarity != 0 && Polarity != 1))
        {
                return -3;
        }
        if((Wiring_Mode != 0 && Wiring_Mode != 1))
        {
                return -4;
        }
        if (Wiring_Mode == 0){                                  // Single-Ended
                Max_Amplitude = 10;                             // Volts
                if(Desired_Output_Voltage > 10)                 // Saturate voltages at card limits
                {
                        Desired_Output_Voltage = 10;
                        Return_Value = -2;
                }
        }
        else {                                                  // Differential
                Max_Amplitude = 20;                             // Volts
                if(Desired_Output_Voltage > 20)                 // Saturate voltages at card limits
                {
                        Desired_Output_Voltage = 20;
                        Return_Value = -2;
                }
        }
        if (Polarity == 0){                                     // Unipolar
                Software_Count_Range = 4095;
                Software_Count_Offset = 0;
                if(Desired_Output_Voltage < 0)                  // Saturate voltages at card limits
                {
                        Desired_Output_Voltage = 0;
                        Return_Value = -2;
                }
        }
        else {                                                  // Bipolar
                Software_Count_Range = 2047;
                Software_Count_Offset = 2047;
                if(Wiring_Mode == 0)
                {
```

```
                        if(Desired_Output_Voltage < (-10))    // Saturate voltages at card limits
                        {
                                Desired_Output_Voltage = (-10);
                                Return_Value = -2;
                        }
                } else {
                        if(Desired_Output_Voltage < (-20))    // Saturate voltages at card limits
                        {
                                Desired_Output_Voltage = (-20);
                                Return_Value = -2;
                        }
                }
        }
        // Conversion factor in units of counts/volt:
        Conversion_Factor = (double)((double)(Software_Count_Range)/(double)(Max_Amplitude));
        Software_Counts = (volatile unsigned int)((double)Software_Count_Offset +
                                                  Desired_Output_Voltage*Conversion_Factor);
        DAC_Word = *pACC59E_DAC;
        if (DAC_Channel <= 4){                                  // Channels 1-4
                *pACC59E_DAC = (((Software_Counts << 8)  & 0xFFFFFF00)^(DAC_Word & 0xFFF00000));

        }
        else {                                                  // Channels 5-8
                *pACC59E_DAC = (((Software_Counts << 20) & 0xFFFFFF00)^(DAC_Word & 0x000FFF00));

        }
        return Return_Value;
}
```