ARTISAN®
TECHNOLOGY GROUP

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

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

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

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

Custom engineering
so your equipment works exactly as you specify.

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

Expert team | Trust guarantee | 100% satisfaction

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

Find the *Curtiss-Wright / Systran ADC128F1-6A* at our website: *Click HERE*

ADC128F1

SOFTWARE LIBRARY MANUAL

Document No. B-T-ML-ADC128F1-A-0-A1

FOREWORD

The information in this document has been carefully checked and is believed to be accurate; however, no responsibility is assumed for inaccuracies. SYSTRAN reserves the right to make changes without notice.

SYSTRAN makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.

Copyright 1989-1995 SYSTRAN Corporation. All rights reserved.

IndustryPack® is a registered trademark of GreenSpring Computers, Inc.
Microware® is a registered trademark of the Microware Systems Corporation.
Motorola® is a registered trademark of the Motorola Corporation.
NuBus™ is a trademark of Texas Instruments, Inc.
OS-9® is a registered trademark of Microware Systems Corporation.

Revised: May 11, 1995

SYSTRAN Corporation
4126 Linden Avenue
Dayton, OH 45432-3068
(513) 252-5601

TABLE OF CONTENTS

1.0 INTRODUCTION 1-1
- 1.1 Purpose 1-1
- 1.2 Scope 1-1
- 1.3 Overview 1-1
- 1.4 System Requirements 1-1
- 1.5 Contents 1-1
- 1.6 Software Layers 1-2
- 1.7 Files Contained on the Disk 1-2
- 1.8 Why a Device Driver? 1-3

2.0 USING THE SOFTWARE PACKAGE 2-1
- 2.1 Overview 2-1
- 2.2 Copy the Files 2-1
- 2.3 Change the Device Mapping 2-1
- 2.4 Multiple ADC128F1 Devices 2-2
- 2.5 Compile and Link the Device Descriptor 2-2
- 2.6 Link the Program 2-2
- 2.7 Load the Driver and Descriptor 2-2

3.0 PROGRAMMING EXAMPLES 3-1
- 3.1 Overview 3-1
- 3.2 Error Checking 3-1
- 3.3 Syntax 3-1
- 3.4 Integer Register Value to Floating Point Voltage Conversion 3-2
- 3.5 Using Read Volts Routine 3-3
- 3.6 Using Disable CVT Update 3-4

4.0 LIBRARY ROUTINES 4-1
- 4.1 ADC128F1_Read_ID_PROM() 4-2
- 4.2 ADC128F1_Read_Reg() 4-3
- 4.3 ADC128F1_Read_Volts() 4-4
- 4.4 ADC128F1_Write_Reg() 4-5
- 4.5 ADC128F1_CVT_Update_Control() 4-6

5.0 BINDER ROUTINES 5-1
- 5.1 ADC128F1ReadID() 5-2
- 5.2 ADC128F1ReadReg16() 5-3
- 5.3 ADC128F1WriteReg16() 5-4

6.0 DEVICE DRIVER ROUTINES 6-1
- 6.1 Overview 6-1
- 6.2 I$Open Routine 6-1
- 6.3 I$Close Routine 6-1
- 6.4 I$GetStt Routines 6-2

Copyright 1995, SYSIRAN Corp.

6.5 I$SetStt Routines 6-2

APPENDIX A.
REGISTER ADDRESSING A-1
A.1 ID PROM Addresses A-1
A.2 Register Addresses A-2
A.3 Word Access Address Translation A-3
A.4 Byte Access Address Translation A-4

APPENDIX B.
REGISTER DESCRIPTIONS B-1
B.1 Register Definitions B-1
B.2 Data Registers 0 to 7 . . . . . B-1
B.3 Control Register B-2
B.4 Status Register B-2

FIGURES

Figure 1-1 Diagram of ADC128F1 Software Levels 1-2

TABLES

Table 4-1 Library Routines 4-1
Table 5-1 Binder Routines 5-1
Table A-1 ADC128F1 ID Address Space Listing A-1
Table A-2 ADC128F1 I/O Address Map A-2
Table A-3 Word Access Address Translation Table A-3
Table A-4 Byte Access Address Translation Table A-4
Table B-1 Data Registers 0 to 7 Bit Descriptions B-1
Table B-2 Control Register Bit Description B-2
Table B-3 Status Register Bit Definitions B-2

Copyright 1995, SYSIRAN Corp.

GLOSSARY

[x:y]. Nomenclature designating a bit-range, where "x" is the left-most bit and "y" is the right-most bit. (e.g. Data bus [7:0] refers to the Least Significant eight bits).

ADC. Analog-to-digital convertor.

binder. A routine to allow device driver routines to be called directly from a 'C' language application program.

byte. 8 bits.

byte-lane. 8 bits of a data bus on octal boundaries.

CVT. Current value table.

DAC. Digital to Analog Converters.

I/O service call. Device driver routine call through the operating system.

makefile. Compiling and linking instructions for the Make utility.

MMU. Memory Management Unit.

ns, µs, ms. Nanoseconds, microseconds, and milliseconds, respectively.

SCF. OS-9 Sequential Character File.

word. 16 bits.

1.0 INTRODUCTION

1.1 Purpose

This is the reference manual for the ADC128F1 Software Library, part number BSDR-ADC128F1. The software used to interface with the ADC128F1 IPack card is intended to aid the development of application programs by providing a collection of 'C' callable routines for configuring and operating the ADC128F1 device.

1.2 Scope

This manual describes how to use the ADC128F1 Software Library and OS-9 driver. The reader should have an understanding of 'C' language programming and the OS-9 operating system.

1.3 Overview

The ADC128F1 is an IPack card with 8 channels of 12-bit analog-to-digital converters (ADC). The input ranges for all channels are fixed at between -10 V and +10 V. The eight individual ADCs continuously convert, writing the converted value to a current value table (CVT). Eight data registers, one per channel, are used to read the CVT entries. A single control register can be used to enable/disable the updating of the CVT. Signals are provided at the I/O connector that allow monitoring/controlling the ADC externally. A single status register can be used to monitor activity on this external port.

1.4 System Requirements

This software is designed to run, as supplied, with applications written in Microware 'Ultra C' running under OS-9 version 2.4 or later. However, since SYSTRAN provides the source code, the routines in this package could serve as models for writing applications in other languages and for running under other operating systems.

1.5 Contents

This package contains support software for interfacing to the ADC128F1 device and includes the following items:

- A 3.5 inch, OS-9 formatted floppy disk containing the source code and object files for 'C' interface routines, and an OS-9 device driver.
- A 3.5 inch, DOS formatted floppy disk containing the source code.

- This manual.

If any of these items are missing, contact SYSTRAN for assistance.

1.6 Software Layers

The ADC128F1 software package is composed of the following software layers:

- 'C' callable library routines that facilitate configuring and operating the ADC128F1 device.
- Binder routines that allow the device driver routines to be called directly from a 'C' language application program.
- An OS-9 SCF device driver.

Figure 1-1 illustrates how these component layers provide a link from the application software to the ADC128F1 hardware:

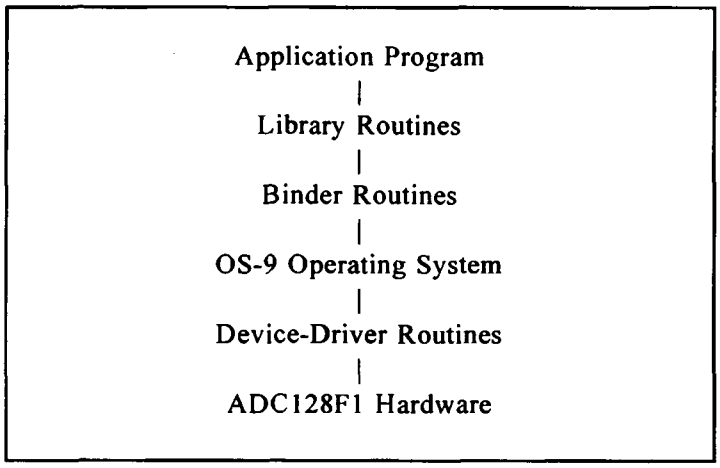


Figure 1-1 Diagram of ADC128F1 Software Levels

Each of these components is described in detail in the following sections.

1.7 Files Contained on the Disk

There are three directories (DRV, LIB, and RELS) contained on the OS-9 formatted disk supplied with this package. The DOS formatted disk contains only the DRV and LIB directories. Each of the files in these directories is listed and briefly described in the following paragraphs.

In the DRV directory:

bnad12f1.a - This is the assembler source code for the binder routines.
drad12f1.a - This is the assembler source code for the device driver.
ad12f1.a - This is the assembler source code for the device descriptor.

statcode.d - This assembler include file contains *I$GetStt* and *I$SetStt* function code definitions. It is used by the binder and the driver.

makefile - This file contains compiling and linking instructions for the device descriptor and driver.

In the LIB directory:

ad12f1ri.c - This is the 'C' source code for the ADC128F1_Read_ID_PROM() routine.

ad12f1rr.c - This is the 'C' source code for the ADC128F1_Read_Reg() routine.

ad12f1rv.c - This is the 'C' source code for the ADC128F1_Read_Volts() routine.

ad12f1uc.c - This is the 'C' source code for the ADC128F1_CVT_Update_Control() routine.

ad12f1wr.c - This is the 'C' source code for the ADC128F1_Write_Reg() routine.

ad12f1dr.h - This include file contains function prototypes for the binder routines.

ad12f1lb.h - This include file contains function prototypes for the library routines.

ad12f1ld.h - This include file contains type definitions used by the library routines.

ad12f1rg.h - This include file contains ADC128F1 register definitions.

gendefs.h - This include file contains general constant definitions.

makefile - This makefile shows how to compile the library files and link them to your application program.

In the RELS directory:

bnad12f1.r - This is the object module for the binder routines.

drad12f1 - This is the loadable/executable device driver module.

drad12f1.r - This is the object module for the device driver.

ad12f1 - This is the loadable device descriptor module.

ad12f1.r - This is the object module for the device descriptor.

ad12f1ri.r - This is the object module for the ADC128F1_Read_ID_PROM() routine.

ad12f1rr.r - This is the object module for the ADC128F1_Read_Reg() routine.

ad12f1rv.r - This is the object module for the ADC128F1_Read_Volts() routine.

ad12f1uc.r - This is the object module for the ADC128F1_CVT_Update_Control() routine.

ad12f1wp.r - This is the object module for the ADC128F1_Write_Port() routine.

ad12f1wr.r - This is the object module for the ADC128F1_Write_Reg() routine.

1.8 Why a Device Driver?

A device driver may be needed if the device generates interrupts or if the computer has a memory management unit (MMU).

Since the ADC128F1 does not generate interrupts, a device driver is needed only if both of the following conditions apply:

- Your computer has an MMU
- The application program will execute at the user-privilege level

A device driver is needed in this case because the computer's operating system uses the MMU to restrict user-state application programs from accessing areas that are outside their own instruction and data space. Since a device driver executes at the system-state privilege level, it allows access to the ADC128F1 registers, even if the host has an MMU and the application program is executing at the user-privilege level.

However, when the above conditions do not apply, non-device-driver register-access routines may be used instead of a device driver. This approach significantly increases performance because the overhead of the I/O service call (call of the device driver routine through the operating system) is not incurred.

Instead of using a device driver, another option available with OS-9 is using colored memory lists. Colored memory lists make normally restricted sections of memory accessible to application programs executing at the user-privilege level. For details about using colored memory, refer to the Microware OS-9 documentation or contact Microware technical support.

Copyright 1995, SYSIRAN Corp.

2.0 USING THE SOFTWARE PACKAGE

2.1 Overview

This section describes how to install the software, map the device I/O and ID base addresses; and to compile, link and load the software.

2.2 Copy the Files

Make a backup copy of the supplied disk. Save the master disk (the one supplied with this package) and use the copy to install the software on your system. Copy all of the files contained on the supplied disk to the development disk on your system. To maintain continuity, copy these files into a directory called ADC128F1, and maintain the same subdirectory structure.

2.3 Change the Device Mapping

The ADC128F1 supports I/O and ID IPack bus transfers. Before the device driver can successfully access the ADC128F1 device, it must know the base addresses of the I/O space (device registers) and the ID space (ID PROM). This device mapping is achieved by setting the I/O and ID base addresses in the device descriptor. The default mapping values as supplied are:

IOBase - The I/O (register) base address is set to 'FFF58100'
IDBase - ID PROM base address is set to 'FFF58180'

This is the mapping for installation in IPack slot B on the Motorola MVME162 board. If the mapping for the ADC128F1 is different as installed in your system, change the I/O and ID base addresses accordingly.

To remap the ADC128F1, edit the **ad12f1.a** file, which contains the assembler source code for the device descriptor. The I/O and ID base addresses are defined by the constant **IOBase** and the **IDBase** constants, respectively. These constants are located toward the end of the file and are clearly marked by comments. Set these constants to the addresses which reflect the actual mapping of the ADC128F1 device for your system. After editing the descriptor, refer to paragraph 2.5: "Compile and Link the Device Descriptor".

2.4 Multiple ADC128F1 Devices

Some applications require more than one ADC128F1 device. The device driver supplied with this package will support more than one ADC128F1; however, each one must have a unique device descriptor. The device descriptor specifies the device address mapping, and each device must exist at a unique address or contention between devices will occur.

To create a new device descriptor, copy the supplied device descriptor source file **ad12f1.a** and give it a different name, such as **ad12f1_b.a**. Follow the previous instructions to configure the new descriptor for the additional device.

2.5 Compile and Link the Device Descriptor

Once the descriptor has been edited, compile it and link the resulting object module to the system library file **sys.l**. A makefile is included in the DRV directory to serve as a guide for compiling and linking the device descriptor and driver.

2.6 Link the Program

Link the supplied relocatable object modules (files ending in **.r**) with your application code. A sample makefile that illustrates how to link your application to the library files is included in the LIB directory. Instructions for compiling the library routines are also included.

2.7 Load the Driver and Descriptor

Before using the ADC128F1 software, load the device driver (**drad12f1**) and the device descriptor (**ad12f1**) into the module directory (located in RAM on the development system). Refer to the programming examples for details on loading a module into the module directory.

3.0 PROGRAMMING EXAMPLES

3.1 Overview

The following examples show how to control and operate the ADC128F1 with the library routines. The three examples demonstrate how to: convert integer register value to floating point voltage conversion, operate the ADC128F1 device using the *ADC128F1_Read_Volts()* routine, and operate the ADC128F1 device using the *ADC128F1_CVT_Update_Control()* routine to disable CVT update.

In each of the following examples, the driver and descriptor are explicitly loaded into the module directory before they are used. The driver and descriptor could alternately be added to the list of modules that OS-9 loads at boot time or they could be loaded from the **startup** file.

At the end of the examples, the driver and descriptor are unlinked from the module directory. However, if you are using one of the alternate loading methods described in the previous paragraph, you may not want to unlink these modules.

3.2 Error Checking

All of the binder and library routines return a status value to indicate whether the call was a success or a failure. If the call fails, the error code is placed in the global variable **errno**. In the following programming examples, error checking and handling is omitted for the sake of clarity and brevity. The application program should check the status of each call and handle any errors appropriately.

3.3 Syntax

The include files in the programming examples and the descriptions of the routines are specified with the following syntax:

#include

This syntax is valid only if the include files are located in the default DEFS directory on the system. If the include files are located elsewhere, the syntax will be different. The following example is an include file in the same directory as the file you are compiling:

#include "file.h"

Refer to your compiler documentation for more information.

3.4 Integer Register Value to Floating Point Voltage Conversion

The ADC128F1 uses a 12-bit two's-complement binary representation of the input voltages. The library routine *ADC128F1_Read_Volts()* performs the conversion from the integer register value to a floating point voltage value. If you would prefer to do your own conversions, the formula is shown below.

Converting the register value to a voltage:

Vin = SignExtendedRegVal * 0.0048828

where:
Vin is the ADC input voltage,
SignExtendedRegVal is the sign extended value read from the ADC data register (see explanation below).

Sign Extending the 12-bit Register Value

The following illustrates how to sign extend the 12-bit register value to a 16-bit signed integer:

```
Read the ADC data register as a 16-bit word. Bits 12 to 15 are indeterminate
                                                    (not driven by the ADC128F1).
Test bit 11 (bit numbering starting from 0) of the read value.
If bit 11 = 1,
        Then set bits 12 to 15 to a 1 (i.e. sign extend it),
Else
        if bit 11 = 0,
                Then clear bits 12 to 15
        Endif
Endif.
```

The resulting 16-bit value is a two's-complement signed value that can be multiplied by 0.0048828 to obtain the voltage.

3.5 Using Read Volts Routine

This example illustrates operation of the ADC128F1 device using the *ADC128F1_Read_Volts()* routine. This example reads only channel 1. However, all eight channels are read in the same manner.

```
#define ADC128F1_CHAN_1 1                    /* define channel constants for clarity */

#include <module.h>                          /* required for modload and munlink */
#include <ad12f1lb.h>                        /* library routine prototypes */
#include <ad12f1rg.h>                        /* ADC128F1 register definitions */

main() {

/* local variables */
int status;
mh_com *drad12f1_mp, *ad12f1_mp;
int adc128f1path;
float volts_val;

drad12f1_mp = modload("./drad12f1", 0);      /* load the ADC128F1 device driver */
ad12f1_mp = modload("./ad12f1", 0);          /* load the ADC128F1 device descriptor */
adc128f1path = open("/ad12f1", 0);           /* open the ADC128F1 device descriptor */

/* Read ADC channel 1 voltage. (Notice the last argument is the address of */
/* the volts_val variable. */
status = ADC128F1_Read_Volts(adc128f1path, ADC128F1_CHAN_1, &volts_val);

/* The value in the variable volts_val should now equal the input voltage for channel 1. */

close(adc128f1path);                         /* close the ADC128F1 device */
status = munlink(drad12f1_mp);               /* unlink the device driver module */
status = munlink(ad12f1_mp);                 /* unlink the device descriptor module */

}
```

3.6 Using Disable CVT Update

This example illustrates operation of the ADC128F1 device using the *ADC128F1_CVT_Update_Control()* routine and demonstrates how to temporarily disable the update of the CVT. Since all eight channels sample simultaneously, disabling update of the CVT while reading multiple channels ensures that all values in the CVT were sampled at the same time.

```
#include <module.h>                     /* required for modload and munlink */
#include <ad12f1lb.h>                   /* library routine prototypes */
#include <ad12f1rg.h>                   /* ADC128F1 register definitions */

main() {

/* local variables */
int status;
mh_com *drad12f1_mp, *ad12f1_mp;
int adc128f1path;
int i;                                  /* loop control variable */
float volts_vals[9];                    /* declare array of 9 floats. 9 instead of 8 */
                                        /* so we can use elements 1 to 8 for direct */
                                        /* correlation to channel number (skip element 0) */

/* Note: the second argument to ADC128F1_CVT_Update_Control is an enumerated type */
/* Enable_Disable_Type that has two possible values DISABLE and ENABLE.  It is defined */
/* in ad12f1ld.h and automatically included by ad12f1lb.h. */

drad12f1_mp = modload("./drad12f1", 0);     /* load the ADC128F1 device driver */
ad12f1_mp = modload("./ad12f1", 0);         /* load the ADC128F1 device descriptor */
adc128f1path = open("/ad12f1", 0);          /* open the ADC128F1 device descriptor */

/* Disable update of CVT */
status = ADC128F1_CVT_Update_Control(adc128f1path, DISABLE);

/* Read all 8 ADC channels. */
for( i = 1 ; i < 9 ; i++ ) {
    /* Notice the last argument is the address of the volts_vals element. */
    status = ADC128F1_Read_Volts(adc128f1path, i, &volts_vals[i]);
}

/*  The 8 voltage values just read were all sampled by the ADC128F1 at the same time. */

/* Re-enable update of CVT */
status = ADC128F1_CVT_Update_Control(adc128f1path, ENABLE);

close(adc128f1path);                    /* close the ADC128F1 device */
status = munlink(drad12f1_mp);          /* unlink the device driver module */
status = munlink(ad12f1_mp);            /* unlink the device descriptor module */

}
```

4.0 LIBRARY ROUTINES

The library routines, which are written in 'C', are the highest level routines in the ADC128F1 software package. The ADC128F1 device can be configured and operated by direct routine calls from a 'C' application program. A bit level understanding of the ADC128F1 registers is not required..

Table 4-1 lists the library routines and gives a short functional description of each. More detailed descriptions of how to use the library routines are presented in the following paragraphs.

Table 4-1 Library Routines

LIBRARY ROUTINE NAME	PAGE	LIBRARY ROUTINE FUNCTION
ADC128F1_Read_ID_PROM()	4-2	Reads the ID PROM area.
ADC128F1_Read_Reg()	4-3	Reads from a 16-bit data register.
ADC128F1_Read_Volts()	4-4	Reads the data register corresponding to the specified channel, converts to a voltage, and returns the voltage value.
ADC128F1_Write_Reg()	4-5, 4-6	Writes to a 16-bit register.
ADC128F1_CVT_Update_Control()	4-6	Enables/disables the update of the CVT.

4.1 ADC128F1_Read_ID_PROM()

Routine Name: ADC128F1_Read_ID_PROM()

Syntax:

```
#include <ad12f1lb.h>
#include <ad12f1rg.h>
#include <errno.h>

int ADC128F1_Read_ID_PROM(path, byte_array);
int path;
u_int8 byte_array[12];
```

Description: *ADC128F1_Read_ID_PROM()* reads the contents of the ADC128F1 ID PROM and places the values in the array pointed to by **byte_array**. The **path** argument is the file path returned from the *open()* call. The **byte_array** argument should point to a byte array large enough to hold the 12 ID PROM byte values.

If an error occurs, '-1' is returned and the error code is placed in the global variable **errno**. Refer to the Microware manual *Using OS-9* for a description of the error messages.

See Also: ADC128F1ReadID() (Page 5-2).

Possible Errors: "E$BusErr".



NOTE: Before using this routine you must call *open()* successfully.

4.2 ADC128F1_Read_Reg()

Routine Name: ADC128F1_Read_Reg()

Syntax:

```
#include <ad12f1lb.h>
#include <ad12f1rg.h>
#include <errno.h>

int ADC128F1_Read_Reg(path, reg_id, word_value_ptr);
int path;
int reg_id;
u_int16 *word_value_ptr;
```

Description: *ADC128F1_Read_Reg()* reads the specified 16-bit data register and places the value in the location pointed to by **word_value_ptr**. The **path** argument is the file path returned from the *open()* call. The value of **reg_id** can range from '0' to '9', corresponding to the ten registers.

If an error occurs, '-1' is returned and the error code is placed in the global variable **errno**. Refer to the Microware manual *Using OS-9* for a description of the error messages.

See Also: ADC128F1ReadReg16 (Page 5-3).

Possible Errors: "E$BusErr" and "E$IllArg".



NOTE: Before using this routine you must call *open()* successfully.

4.3 ADC128F1_Read_Volts()

Routine Name: ADC128F1_Read_Volts()

Syntax:

```
#include <ad12f1lb.h>
#include <ad12f1rg.h>
#include <errno.h>

int ADC128F1_Read_Setting(path, adc_id, float_value_ptr);
int path;
int adc_id;
float *float_value_ptr;
```

Description: *ADC128F1_Read_Volts()* reads the ADC channel data register specified by **adc_id**, converts the value to a voltage, and places the value in the location pointed to by **float_value_ptr**. The **path** argument is the file path returned from the *open()* call. The value of **adc_id** can range from '1' to '8', corresponding to the eight ADC channels

If an error occurs, '-1' is returned and the error code is placed in the global variable **errno**. Refer to the Microware manual *Using OS-9* for a description of the error messages.

See Also: ADC128F1ReadReg16 (Page 5-3).

Possible Errors: "E$BusErr" and "E$IllArg".



NOTE: Before using this routine you must call *open()* successfully.

4.4 ADC128F1_Write_Reg()

Routine Name: ADC128F1_Write_Reg()

Syntax:

```
#include <ad12f1lb.h>
#include <ad12f1rg.h>
#include <errno.h>

int ADC128F1_Write_Reg(path, reg_id, word_value);
int path;
int reg_id;
u_int16 word_value;
```

Description: *ADC128F1_Write_Reg()* writes **word_value** to the 16-bit data register. The **path** argument is the file path returned from the *open()* call. The value of **reg_id** can only be '8', corresponding to the only writable register, the control register.

If an error occurs, '-1' is returned and the error code is placed in the global variable **errno**. Refer to the Microware manual *Using OS-9* for a description of the error messages.

See Also: ADC128F1WriteReg16 (Page 5-4).

Possible Errors: "E$BusErr" and "E$IllArg".



NOTE: Before using this routine you must call *open()* successfully.

4.5 ADC128F1_CVT_Update_Control()

Routine Name: ADC128F1_CVT_Update_Control()

Syntax:

```
#include <ad12f1lb.h>
#include <ad12f1rg.h>
#include <errno.h>

int ADC128F1_CVT_Update_Control(path, ena_dis);
int path;
enum Enable_Disable_Type  ena_dis;
```

Description: *ADC128F1_CVT_Update_Control()* enables/disables updates to the CVT. The **path** argument is the file path returned from the *open()* call. The **ena_dis** argument is an enumerated type defined in **ad12f1ld.h** which is included by **ad12f1lb.h**. The possible values for **ena_dis** are ENABLE and DISABLE..

If an error occurs, '-1' is returned and the error code is placed in the global variable **errno**. Refer to the Microware manual *Using OS-9* for a description of the error messages.

See Also: ADC128F1WriteReg16 (Page 5-4).

Possible Errors: "E$BusErr" and "E$IllArg".



NOTE: Before using this routine you must call *open()* successfully.

5.0 BINDER ROUTINES

The binder routines, written in Motorola 680X0 assembler language, provide an interface between the device-driver *I$GetStt* and *I$SetStt* routines, and an 'Ultra C' program. These binder routines, which are used to access ADC128F1 registers, are called by the library routines. They may also be called directly from a 'C' application program. The respective device-driver routine is then called by the binder routine through an OS-9 system call.

This section contains a detailed description of each binder routine. Table 5-1 lists the binder routines and gives a short functional description of each.

Table 5-1 Binder Routines

BINDER ROUTINE NAME	PAGE	BINDER ROUTINE FUNCTION
ADC128F1ReadID()	5-2	Reads the ID PROM area.
ADC128F1ReadReg16()	5-3	Reads from a 16-bit data register.
ADC128F1WriteReg16()	5-4	Writes to a 16-bit data register.

5.1 ADC128F1ReadID()

Routine Name: ADC128F1ReadID()

Syntax:

```
#include <ad12f1dr.h>
#include <errno.h>

int ADC128F1ReadID(path, reg_offset, word_value_ptr)
int path;
int reg_offset;
u_int16 *word_value_ptr;
```

Description: *ADC128F1ReadID()* reads the ADC128F1 ID PROM location at **reg_offset** from the ADC128F1 ID PROM base address and places the value in the location pointed to by **word_value_ptr**. Only the lower 8 bits of the word contain valid data because the ID PROM values are bytes at every other location. The **path** argument is the file path returned from the *open()* call. The value of **reg_offset** can be 0, 2, ... 20, 22, representing the offsets to the twelve ID PROM values from the ID PROM base address. If successful, a '0' is returned.

If an error occurs, '-1' is returned and the error code is placed in the global variable **errno**. Refer to the Microware manual *Using OS-9* for a description of the error messages.

See Also: *I$GetStt* I/O system call in the OS-9 manual.

Possible Errors: "E$UnkSvc" and "E$BusErr".



NOTE: Before using this routine you must call *open()* successfully.

5.2 ADC128F1ReadReg16()

Routine Name: ADC128F1ReadReg16()

Syntax:
```
#include <ad12f1dr.h>
#include <errno.h>

int ADC128F1ReadReg16(path, reg_offset, word_value_ptr)
int path;
int reg_offset;
u_int16 *word_value_ptr;
```

Description: *ADC128F1ReadReg16()* reads the ADC128F1 register at **reg_offset** from the ADC128F1 register base address (the IO base address) and places the value in the location pointed to by **word_value_ptr**. The **path** argument is the file path returned from the *open()* call. The value of **reg_offset** can be 0, 2,... 16, 18, representing the offsets to the ten registers from the register base address. If successful, a '0' is returned. .

If an error occurs, '-1' is returned and the error code is placed in the global variable **errno**. Refer to the Microware manual *Using OS-9* for a description of the error messages.

See Also: *I$GetStt* I/O system call in the OS-9 manual.

Possible Errors: "E$UnkSvc" and "E$BusErr".



NOTE: Before using this routine you must call *open()* successfully.

5.3 ADC128F1WriteReg16()

Routine Name: ADC128F1WriteReg16()

Syntax:
```
#include <ad12f1dr.h>
#include <errno.h>

int ADC128F1WriteReg16(path, reg_offset, word_value)
int path;
int reg_offset;
u_int16 word_value;
```

Description: *ADC128F1WriteReg16()* writes the 16-bit **word_value** to the ADC128F1 data register at **reg_offset** from the register base address (the IO base address). The **path** argument is the file path returned from the *open()* call. The value of **reg_offset** can only be '16', representing the offset to the only writable register, the control register. If successful, a '0' is returned.

If an error occurs, '-1' is returned and the error code is placed in the global variable **errno**. Refer to the Microware manual *Using OS-9* for a description of the error messages.

See Also: *I$SetStt* I/O system call in the OS-9 manual.

Possible Errors: "E$UnkSvc" and "E$BusErr".



NOTE: Before using this routine you must call *open()* successfully.

6.0 DEVICE DRIVER ROUTINES

6.1 Overview

The device driver for the ADC128F1 is an OS-9 SCF (Sequential Character File) type driver. The *I$Open* and *I$Close* I/O service calls cause the operating system and the SCF file manager to initialize and to terminate the device driver, respectively. The *I$GetStt* and *I$SetStt* I/O service calls are used to read from and to write to the ADC128F1 registers. These routines are discussed further in the following paragraphs. The other standard OS-9 SCF driver I/O service calls (*I$Read*, *I$ReadLn*, *I$Write*, *I$WriteLn*, and *I$Create*) are not implemented and return an error.

The driver routines are intended to be invoked through the supplied binder routines which can be called directly from 'Ultra C'. The binder routines are extremely efficient and they should be used to access the driver. Therefore, the following descriptions do not include directions on how to call driver routines directly. Refer to the binder routine source listings or the Microware documentation for information on direct calls.

The *I$GetStt* and *I$SetStt* routines install a local bus error handler before any register access is made. If a bus error occurs, the routine returns a status of '-1' and places "E$BusErr" in the global variable **errno**. This allows an application program to keep running in the event of a non-responding device. The local bus error handler is removed and the original restored when the access is finished.

6.2 I$Open Routine

I$Open must be called before the driver is used. The OS-9 system call *I$Open* returns a path to the opened file. In the case of the ADC128F1, the Open call opens the device descriptor "ad12f1". The Open entry point in the device driver initializes the device static-storage area. Also, the operating system and file manager "prepare" the driver for use when this call is made.

6.3 I$Close Routine

I$Close should be called when the driver is no longer needed. The OS-9 system call *I$Close* closes the path to the device descriptor. The Close entry point in the device driver merely returns with no error. However, the operating system and file manager "clean up" after the driver when this call is made.

6.4 I$GetStt Routines

The ADC128F1 driver has two *I$GetStt* calls. One call reads a 16-bit word from one of the data registers. The other call reads a 16-bit word from one of the ID PROM locations. These routines are invoked by calling the binder routines *ADC128F1ReadReg16()* and *ADC128F1ReadID()*, respectively

6.5 I$SetStt Routines

The ADC128F1 driver has one *I$SetStt* call that writes a 16-bit word value to the ADC128F1 control register. This routine is invoked by calling the binder routine *ADC128F1WriteReg16()*.

APPENDIX A.
REGISTER ADDRESSING

This ID PROM and register addressing information is included in the *ADC128F1 User Manual* and is repeated as an appendix to this manual for the sake of completeness.

A.1 ID PROM Addresses

Table A-1 shows the ADC128F1 emulated ID address space PROM listing. The addresses shown are the IPack addresses. To determine the correlation of the local bus to the IPack address, see paragraphs A.3 and A.4 on word and byte address translation.

Table A-1 ADC128F1 ID Address Space Listing

IPack ADDRESS	DESCRIPTION	DATA READ
IPA = 00 *hex*	ASCII "I"	49 *hex*
IPA = 01 *hex*	ASCII "P"	50 *hex*
IPA = 02 *hex*	ASCII "A"	41 *hex*
IPA = 03 *hex*	ASCII "C"	43 *hex*
IPA = 04 *hex*	SYSTRAN's ID	45 *hex*
IPA = 05 *hex*	ADC128F1's Model Number	6A *hex*
IPA = 06 *hex*	Revision Level	30 *hex*
IPA = 07 *hex*	Reserved	00 *hex*
IPA = 08 *hex*	Low Byte Driver ID	00 *hex*
IPA = 09 *hex*	High Byte Driver ID	00 *hex*
IPA = 0A *hex*	Number of Bytes Used	0C *hex*
IPA = 0B *hex*	CRC	6A *hex*

A.2 Register Addresses

Table A-2 shows the ADC128F1 register map. The addresses shown are the IPack addresses. To determine the correlation of the local bus to the IPack address, see paragraphs A.3 "Word Access Address Translation" and A.4 "Byte Access Address Translation".

Table A-2 ADC128F1 I/O Address Map

IPack ADDRESS	BYTE-LANE 1	BYTE-LANE 0	DESCRIPTION
IPA = 00 *hex*	[XXXX] [11:8]	[7:0]	Data Register 0;12 bits
IPA = 01 *hex*	[XXXX] [11:8]	[7:0]	Data Register 0;12 bits
IPA = 02 *hex*	[XXXX] [11:8]	[7:0]	Data Register 0;12 bits
IPA = 03 *hex*	[XXXX] [11:8]	[7:0]	Data Register 0;12 bits
IPA = 04 *hex*	[XXXX] [11:8]	[7:0]	Data Register 0;12 bits
IPA = 05 *hex*	[XXXX] [11:8]	[7:0]	Data Register 0;12 bits
IPA = 06 *hex*	[XXXX] [11:8]	[7:0]	Data Register 0;12 bits
IPA = 07 *hex*	[XXXX] [11:8]	[7:0]	Data Register 0;12 bits
IPA = 08 *hex*	[XXXX,XXXX]	[XXXX,XXX][0]	Control Register; 1 bit
IPA = 09 *hex*	[XXXX,XXXX]	[XXX] [5:0]	Status Register;

NOTE: [X] means bit is not used

A.3 Word Access Address Translation

Table A-3 shows the relationship between VME, PC-AT, and NuBus local bus addresses, and the IPack address for word accesses. In the table, the word BASE represents the I/O or ID base address. All addresses are in hexadecimal.

Table A-3 Word Access Address Translation Table

VME BUS ADDRESS	PC-AT BUS ADDRESS	NuBus ADDRESS	IPack ADDRESS
BASE + 0	BASE + 0	BASE + 2	IPA = 00 *hex*
BASE + 2	BASE + 2	BASE + 6	IPA = 01 *hex*
BASE + 4	BASE + 4	BASE + A	IPA = 02 *hex*
BASE + 6	BASE + 6	BASE + E	IPA = 03 *hex*
BASE + 8	BASE + 8	BASE + 12	IPA = 04 *hex*
BASE + A	BASE + A	BASE + 16	IPA = 05 *hex*
BASE + C	BASE + C	BASE + 1A	IPA = 06 *hex*
BASE + E	BASE + E	BASE + 1E	IPA = 07 *hex*
BASE + 10	BASE + 10	BASE + 22	IPA = 08 *hex*
BASE + 12	BASE + 12	BASE + 26	IPA = 09 *hex*
BASE + 14	BASE + 14	BASE + 2A	IPA = 0A *hex*
BASE + 16	BASE + 16	BASE + 2E	IPA = 0B *hex*

Copyright 1995, SYSIRAN Corp.

A.4 Byte Access Address Translation

Table A-4 shows the relationship between VME, PC-AT, and NuBus local bus addresses, and the IPack address for byte accesses. In the table, the word BASE represents the I/O or ID base address. All addresses are in hexadecimal.

Table A-4 Byte Access Address Translation Table

VME BUS ADDRESS	PC-AT BUS ADDRESS	NuBus ADDRESS	IPack ADDRESS	BYTE-LANE*
BASE + 1	BASE + 0	BASE + 3	IPA = 00 *hex*	0
BASE + 0	BASE + 1	BASE + 2	IPA = 00 *hex*	1
BASE + 3	BASE + 2	BASE + 7	IPA = 01 *hex*	0
BASE + 2	BASE + 3	BASE + 6	IPA = 01 *hex*	1
BASE + 5	BASE + 4	BASE + B	IPA = 02 *hex*	0
BASE + 4	BASE + 5	BASE + A	IPA = 02 *hex*	1
BASE + 7	BASE + 6	BASE + F	IPA = 03 *hex*	0
BASE + 6	BASE + 7	BASE + E	IPA = 03 *hex*	1
BASE + 9	BASE + 8	BASE + 13	IPA = 04 *hex*	0
BASE + 8	BASE + 9	BASE + 12	IPA = 04 *hex*	1
BASE + B	BASE + A	BASE + 17	IPA = 05 *hex*	0
BASE + A	BASE + B	BASE + 16	IPA = 05 *hex*	1
BASE + D	BASE + C	BASE + 1B	IPA = 06 *hex*	0
BASE + C	BASE + D	BASE + 1A	IPA = 06 *hex*	1
BASE + F	BASE + E	BASE + 1F	IPA = 07 *hex*	0
BASE + E	BASE + F	BASE + 1E	IPA = 07 *hex*	1
BASE + 11	BASE + 10	BASE + 23	IPA = 08 *hex*	0
BASE + 10	BASE + 11	BASE + 22	IPA = 08 *hex*	1
BASE + 13	BASE + 12	BASE + 27	IPA = 09 *hex*	0
BASE + 12	BASE + 13	BASE + 26	IPA = 09 *hex*	1
BASE + 15	BASE + 14	BASE + 2B	IPA = 0A *hex*	0
BASE + 14	BASE + 15	BASE + 2A	IPA = 0A *hex*	1
BASE + 17	BASE + 16	BASE + 2F	IPA = 0B *hex*	0
BASE + 16	BASE + 17	BASE + 2E	IPA = 0B *hex*	1

*Byte-lane 1 is not applicable for ID space

APPENDIX B.
REGISTER DESCRIPTIONS

B.1 Register Definitions

These register descriptions are included in the *ADC128F1 User Manual* and are repeated here for the sake of completeness.

The ADC128F1 has ten 16-bit-wide registers: eight data registers, one control register, and one status register. The ADC128F1 supports word (16-bit) reads and writes. The registers are described in detail below.

The Read/Write (R/W) value in the definition table indicates access. "R" indicates read-only access, "W" indicates write-only access, and "RW" indicates both read and write access.

B.2 Data Registers 0 to 7

The eight 16 bit read-only data registers are used to read the input voltage of the eight ADC channels. Data register 0 corresponds to ADC channel 1, register 1 to channel 2, ... register 7 to channel 8.

Table B-1 Data Registers 0 to 7 Bit Descriptions

Bit #	15 to 12	11	10	9	8	7	6	5	4	3	2	1	0
Bit Name	Not used	D 11	D 10	D 9	D 8	D 7	D 6	D 5	D 4	D 3	D 2	D 1	D 0
R/W	Read as 0's	R	R	R	R	R	R	R	R	R	R	R	R
Power-up state	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

D11 - D0 : These bits comprise the 12 bit ADC value.

B.3 Control Register

The Control Register contains a single bit that is used to enable/disable the updating of the CVT. Disabling the CVT before reading the ADC channels ensures that all channels' readings represent voltages that are simultaneously sampled by the parallel ADC's. If this bit is set to a '1', the CVT updates are enabled. If this bit is set to a '0', the CVT updates are disabled.

Table B-2 Control Register Bit Description

Bit #	15 to 1	0
Bit Name	Not used	N_UPDIS
R/W	Read as 0's, Writes discarded.	RW
Power-up state	N/A	1 (Run)

N_UPDIS : This is the CVT update enable/disable bit.

B.4 Status Register

This register reflects the state of the data monitor port signals. The data monitor port allows the ADC128F1 device's operation to be checked without going through the IPack logic interface.

Table B-3 Status Register Bit Definitions

Bit #	15 to 12	5	4	3	2	1	0
Bit Name	Not used	N_DATAOEG	DATAOE2	DATAOE1	DATAOE0	N_HUPDIS	RUN
R/W	Read as 0's	R	R	R	R	R	R
Power-up state	N/A	N/A	N/A	N/A	N/A	N/A	N/A

N_ DATAOEG : This bit indicates the state of the N_DATAOEG signal.
N_ DATAOE2 - N_DATAOE0 : These bits indicate the state of the N_DATAOE2 - N_DATAOE0 signals.
N_ HUPDIS : This bit indicates the state of the N_HUPDIS signal.
RUN : This bit indicates the state of the RUN signal.

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

Gold-standard solutions

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

We buy equipment

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

Learn more!

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

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

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

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

ARTISAN® TECHNOLOGY GROUP