AN-1365
应用笔记

One Technology Way • P.O. Box 9106 • Norwood, MA 02062-9106, U.S.A. • Tel: 781.329.4700 • Fax: 781.461.3113 • www.analog.com

AD1933/AD1934/AD1938/AD1939/AD1974引导应用

作者：David M. Thibodeau

简介

AD1933/AD1934/AD1938/AD1939/AD1974系列编解码器、ADC和DAC有一个独立工作模式，可以在无微控制器的情况下使用。遗憾的是，除了ADC可用作主机或从机以外，在独立模式下不存在其它工作模式选项。现代系统更多地需要更快速的采样速率或TDM工作模式。在没有系统微控制器的情况下，人们希望能够让转换器自行引导到这些高级工作模式。本应用笔记详细说明了一种用于将这些器件引导到所需任何工作模式的低成本解决方案。

范例使用Microchip® PIC12(L)F1571/PIC12(L)F1572来引导AD1938，另外还支持静音功能。该器件成本非常低，可通过普通经销渠道获得。它提供多种封装，可满足大部分应用的成本/尺寸需求。

为方便阅读，本应用笔记使用AD1938作为引导应用的例子，但该应用对AD1933/AD1934/AD1938/AD1939/AD1974整个系列都有效。

编程示例利用Custom Computer Services, Inc. (CCS) C语言编译器编写。

本应用笔记详细说明了实现此功能所用的硬件和软件。

目录

简介.....1
修订历史.....2
硬件概述.....3
　引脚功能详解.....3
软件概述.....4
　软件文件.....4
软件详解.....4
编译器信息.....4
SPI端口命名规则.....4
时序详解.....4
代码列表.....6

修订历史

2015年8月—修订版0：初始版

硬件概述

AD1933/AD1934/AD1938/AD1939/AD1974全都有一个SPI端口，用于与系统控制器通信。他们全使用相同的ID地址，该地址已嵌入SPI指令格式中，无需对代码做任何更改便可与该系列中的任意器件通信。顺带提一下，这使得我们可以同时对多个器件编程，因为此程序并不从编解码器读取任何数据。如果需要对多个器件进行不同的编程，用户可以修改程序以包括其它CLATCH(从机选择)输出。

Microchip PIC12(L)F1571/PIC12(L)F1572有6个多功能GPIO引脚和1个内置振荡器模块。只要有电源、接地并且电源引脚上有一个0.1 µF旁路电容，该器件便可工作。另外，建议使用一个上拉电阻以支持静音功能。CLATCH线也需要一个上拉电阻，以防编解码器在微控制器启动之前进入独立模式。此微控制器应用仅需三个无源器件，因此该解决方案的性价比非常高。

PIC12F1571/PIC12F1572是该器件的5 V版本，PICLF1571/PICLF1572是低压版本。AD1938的逻辑端口兼容5 V，因此可使用任何一种器件。不过，建议使用PICLF1571/PICLF1572版本。

图1显示了微控制器的GPIO和其它功能。图2显示了示例解决方案中分配给这些引脚的功能。

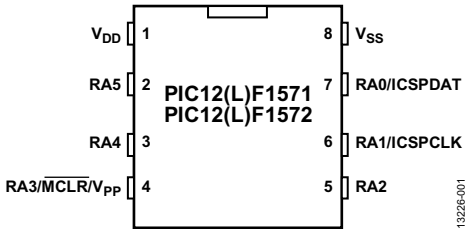

图1. PIC12(L)F1571/PIC12(L)F1572引脚功能

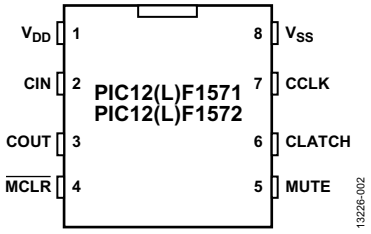

图2. 分配给GPIO引脚的功能

引脚功能详解

下面是PICLF1571/PICLF1572引脚功能的详细说明。

- CIN：此引脚用于将AD1938 SPI输出的数据输入微控制器。此引脚连接到AD1938的COUT引脚。
- COUT：此引脚为微控制器的数据输出引脚，连接到AD1938的CIN引脚。
- $\overline{\text{MCLR}}$: 此引脚是微控制器的主机清零功能。此引脚内置弱上拉电阻和噪声/尖峰滤波器。拉低此引脚会使器件处于复位状态，让该引脚浮动变回高电平状态时，器件复位并从头启动应用程序。这使得开关或系统控制器可以启动复位。如果不使用，此引脚应浮空或连接到V_{DD}。
- MUTE：此引脚需要外部上拉电阻。初始化后，此引脚有效。拉低该引脚会使所有DAC输出静音并保持该状态，直至该引脚回到高电平状态，然后发生取消静音命令。
- CLATCH：此引脚需要连接到AD1938的$\overline{\text{CLATCH}}$引脚。此引脚需要一个弱上拉电阻，使其在施加电源后的启动期间处于高电平。这可以防止AD1938在施加电源后立即进入独立模式。
- CCLK：此引脚为串行时钟引脚，需要连接到AD1938的CCLK引脚。

注意，ADI公司对SPI端口功能使用了不同的名称。详情参见“SPI端口命名规则”部分。

微控制器为SPI主机。

软件概述

软件程序流很简单。

1. 设置微控制器内部振荡器和端口方向。
2. 将$\overline{\text{CLATCH}}$引脚设置为高电平。
3. 等待AD1938完成初始化和校准。
4. 调用发送SPI消息并将编解码器配置为所需工作模式的例程。
5. 进入一个永久循环。
   a. 进入睡眠状态，但设置一个唤醒中断，当MUTE引脚变为低电平时触发该中断。
   b. 唤醒时，发送DAC静音SPI消息，然后等待该引脚变回高电平。
   c. 当MUTE引脚变为高电平时，发送SPI消息使DAC取消静音，然后回到睡眠状态。

几个中断管理步骤详见程序代码。

软件文件

此程序有三个源代码文件。

12LF1572.h：这是此微控制器专用的头文件。此文件包含许多有用的定义和内置函数原型。此文件随同编译器提供。

AD1938 Self Boot main.h：此文件包含编译器指令和此编码器专用的有用定义。

编码器指令用于包括上述头文件，设置器件中的一些保险丝，设置所用内部振荡器频率的延迟时间，然后定义SPI功能。此程序采用软件定义的SPI方案，因为该器件没有硬件SPI端口。有关本例所用编译器的详细信息，请参阅“编译器信息”部分。

为该编码器提供的定义用于设置对编解码器和寄存器地址的读或写命令。

软件详解

所用的数据结构为两个数据类型的联合：一个32位整数和一个包括4个8位整数的结构。这种方法允许以两种不同方式对同一数据进行寻址。一种是SPI消息函数调用所用的32位大整数。此调用需要一个整型变量。另一种方式是采用由4个整型变量组成的结构，以便程序能够对一条SPI消息的各个部分进行寻址。AD1938的SPI消息格式为24位。软件SPI功能允许编程人员定义一条消息要传输的位数。问题是C编译器没有24位整型变量，因此使用32位。该结构有一个称为unused_byte的变量，用于补充这些额外的位。仅传输实际32位变量中的前24位。

此程序不会读取任何寄存器。软件能够从编解码器读取数据。要读取，须使用读取数据的定义；读取后，数据存储在消息结构的Data变量中。代码注释(参见“代码列表”部分)给出了一个关于如何读取数据的例子。

编解码器上电时，所有寄存器清零。数据手册在“寄存器详解”部分中将该值列为第一个值。任何为0或在“寄存器详解”中第一个出现的期望寄存器值，都无需在器件上电时写入。这样可以减少设置器件为期望模式所需的寄存器写操作次数，从而简化程序。

编译器信息

本例使用的编译器可从CCS获得。该编译器既有集成开发环境可供购买，也提供命令行形式。命令行编译器的成本非常低，而且可以集成到Microchip, Inc提供的免费开发环境MicroChip MPLAB®中。

选择该编译器的原因是其产生的文件很小，并且提供许多内置函数。本例使用SPI内置函数，其可在没有硬件SPI端口的器件上实现软件SPI解决方案。

SPI端口命名规则

ADI公司对SPI端口功能使用了不同的引脚名称。变化详情参见表1。

表1. SPI端口引脚命名规则

旧引脚名称	新引脚名称
CDATA	MOSI
COUT	MISO
CCLK	SCLK
$\overline{\text{CLATCH}}$	$\overline{\text{SS}}$

旧名称中的C表示通信。业界其它地方使用的新名称意义如下：

- MOSI：主机输出、从机输入
- MISO：主机输入、从机输出
- SCLK：串行时钟
- $\overline{\text{SS}}$: 从机选择，一般是低电平有效

时序详解

施加电源后，在代码开始执行之前，微控制器有大约65 ms的唤醒时间。这远早于编解码器准备就绪可进行编程，因此，微控制器再等待265 ms。结果，从上电到发生SPI消息的时间约为330 ms。详情参见图3。图4至图8显示了其它信号时序。

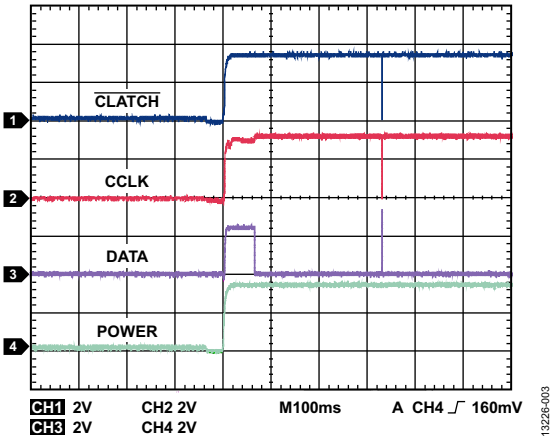

图3. 从上电到SPI消息传输的启动时间

SPI时钟频率约为144 kHz，如图4所示。

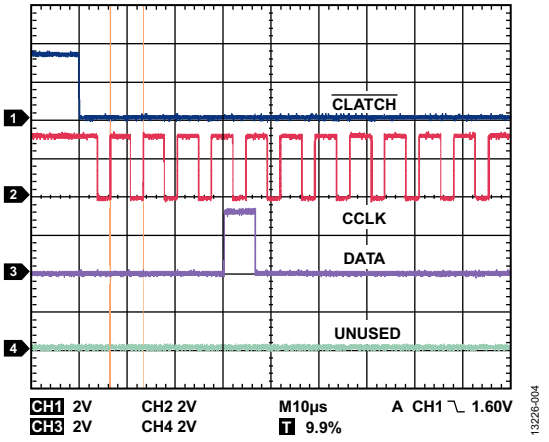

图4. SPI时钟频率 = 144 kHz

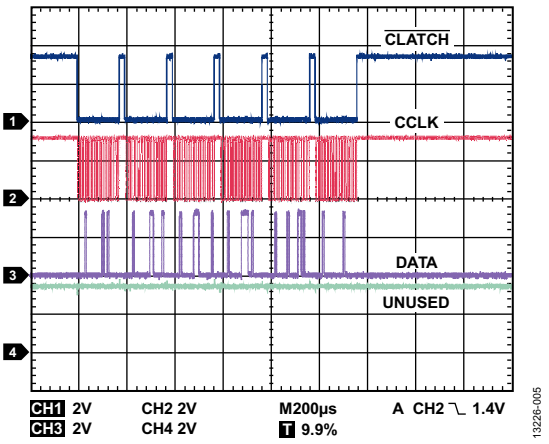

图5. 送出6条SPI消息用于配置编解码器

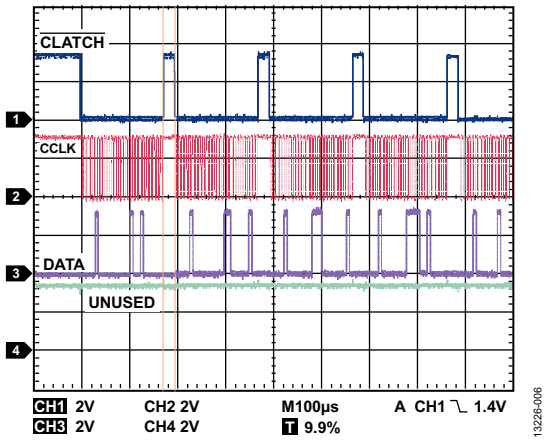

图6. 消息之间的时间约为25 µs

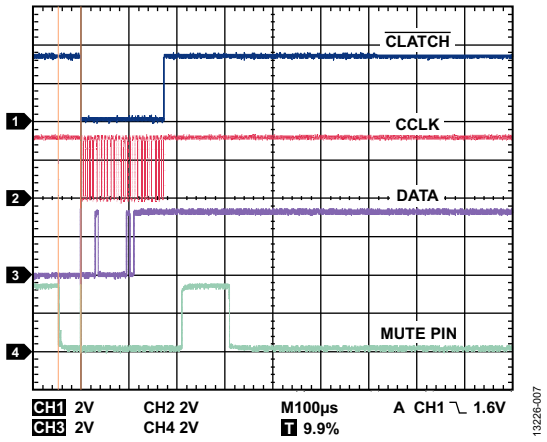

图7. 从MUTE引脚变为低电平和静音消息传输开始的时间为46 µs

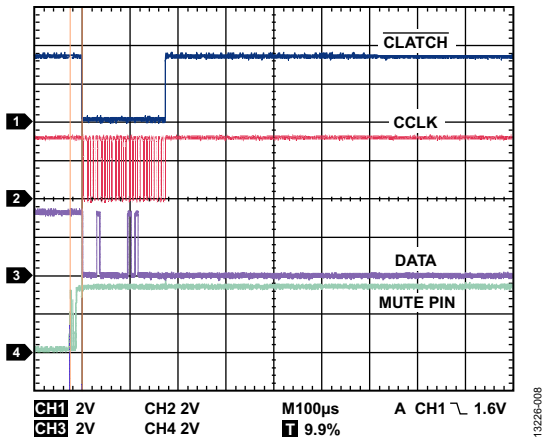

图8. 从MUTE引脚变为高电平和取消静音消息传输开始的时间为25 µs

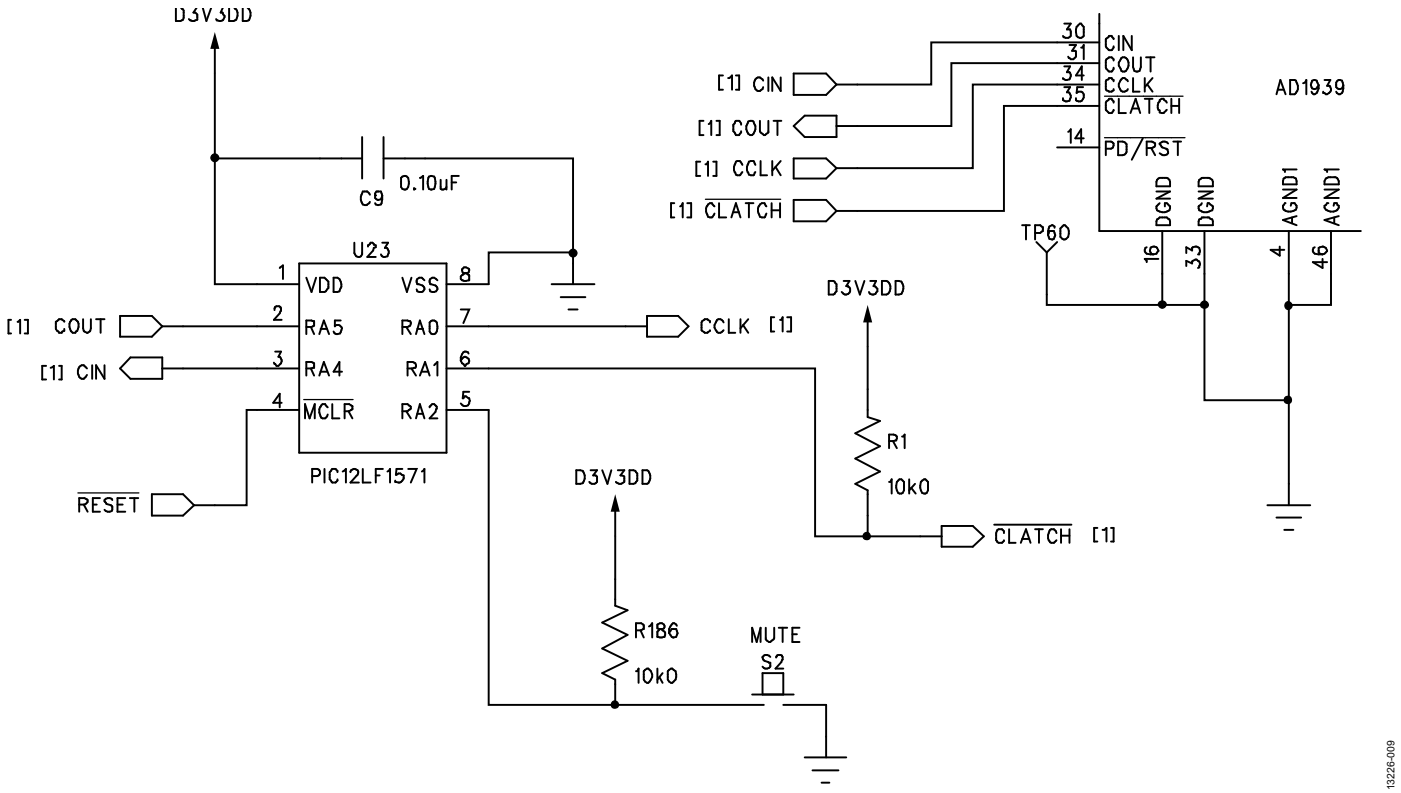

图9. 范例原理图

代码列表

主程序

为了节省引导时间，有必要知道AD1938器件退出复位时所有寄存器皆为0。数据手册显示，默认值为“寄存器详解”中列出的第一个值，这些默认值是0设置。因此，针对所需的工作模式，本程序只需更改不应当为0的寄存器。

本程序的另一个限制是：当$\overline{\text{CLATCH}}$线保持低电平时，内置软件SPI例程不支持大块写操作。这样就不能使用器件支持自动地址递增的特性。

回读数据

下例显示了如何从器件回读数据：

```
#include <AD1938-Self-Boot-main.h>
/*


SPI_Message_In.Full_SPI_Msg = spi_xfer(SPI_Message_Out.Full_SPI_Msg); // Send out the SPI
message with the first byte set to read.
*/


/*
Global Variables
*/
boolean First_Time_Sleep;
union SPI_Message_Type {
 int32 Full_SPI_Msg;
 struct SPI_Bytes_Type {
 int8 Data;
```

```
 int8 Reg_Addr;
 int8 Read_Write;
 int8 Unused_Byte;
 } SPI_Byte;
} SPI_Message_Out; //, SPI_Message_In;

void Configure_Codec_for_TDM8_ADC_Master_MCLKIn()
{
 /*
 This routine configures the codec for TDM8 with the ADC clocks as a master and the DAC clocks
as a slave. MCLK comes from an external crystal source at 12.288 MHz. 32 kHz/48 kHz sampling
rate.









 */
 SPI_Message_Out.SPI_Byte.Unused_Byte = 0; // This will stay the same for all messages so set
this only once.
 SPI_Message_Out.SPI_Byte.Read_Write = Global_Address_Write; // This also stays the same for the
initial bank of writes.

 SPI_Message_Out.SPI_Byte.Reg_Addr = DAC_Control_0;
 SPI_Message_Out.SPI_Byte.Data = 0x40; // This sets the DAC to TDM mode
 spi_xfer(SPI_Message_Out.Full_SPI_Msg); // Send out the SPI message.

 SPI_Message_Out.SPI_Byte.Reg_Addr = DAC_Control_1;
 SPI_Message_Out.SPI_Byte.Data = 0x04; // This sets the DAC to 256 BCLKs per frame which is
8 channels in slave mode
 spi_xfer(SPI_Message_Out.Full_SPI_Msg); // Send out the SPI message.

 SPI_Message_Out.SPI_Byte.Reg_Addr = ADC_Control_0;
 SPI_Message_Out.SPI_Byte.Data = 0x02; // This turns on the high-pass filter for the ADCs which
is a good idea. It removes the DC offsets.
 spi_xfer(SPI_Message_Out.Full_SPI_Msg); // Send out the SPI message.

 SPI_Message_Out.SPI_Byte.Reg_Addr = ADC_Control_1;
 SPI_Message_Out.SPI_Byte.Data = 0x20; // This sets the ADC to TDM mode.
 spi_xfer(SPI_Message_Out.Full_SPI_Msg); // Send out the SPI message.

 SPI_Message_Out.SPI_Byte.Reg_Addr = ADC_Control_2;
 SPI_Message_Out.SPI_Byte.Data = 0x68; // This sets the ADC to master with 256 BCLKs per frame,
8 channels.
 spi_xfer(SPI_Message_Out.Full_SPI_Msg); // Send out the SPI message.

 SPI_Message_Out.SPI_Byte.Reg_Addr = PLL_Control_0;
 SPI_Message_Out.SPI_Byte.Data = 0x80; // This enables the internal MCLK to start up the part
 spi_xfer(SPI_Message_Out.Full_SPI_Msg); // Send out the SPI message.
```

```
}


void Initialize()
{
 setup_oscillator( OSC_16MHZ);
 set_tris_a( 0b11101100 );
 //enable_interrupts(GLOBAL);
 /*
 Note about global interrupts:
 Do not enable the global interrupts for the interrupt on a pin change to wake the processor
from sleep. It is actually desired to not enable them because then the processor just wakes from
sleep and continues program execution from the instructions following the sleep instruction.
 Should the global interrupts be enabled then the processor wakes up from sleep and attempts to
execute the ISR for the interrupt. In the case of this program, there is no ISR; therefore, it
starts executing from Instruction 0 which is the start of the program, basically performing a
reset.
 If the user changes this program and enables the global interrupts, then write an ISR for the
pin change.
 */
 output_high(PIN_A1); // Set the CLATCH pin high upon power up to avoid setting the codec into
standalone mode.
 // There still needs to be a weak pull up on this pin to keep the pin high before the program
executes.
 delay_ms(265); // Insert a delay to allow the PLL of the codec to lock and for initialization
of the codec.
}



void main()
{
 Initialize();
 // Send out the CODEC configuration messages
 Configure_Codec_for_TDM8_ADC_Master_MCLKIn();
 First_Time_Sleep = TRUE; // Do not unmute the DAC the first time the loop is entered and the
part goes to sleep
 // Loop for testing of the mute function.
 while(TRUE)
 {
 //Test if the MUTE pin is high. If it is then go into sleep mode
 if (input(PIN_A2)).
 {
 if (!First_Time_Sleep)
 { // Unmute the codec.
 SPI_Message_Out.SPI_Byte.Read_Write = Global_Address_write;
 SPI_Message_Out.SPI_Byte.Reg_Addr = DAC_Mutes;
 SPI_Message_Out.SPI_Byte.Data = 0x00;
 SPI_Message_Out.SPI_Byte.Unused_Byte = 0;
 spi_xfer(SPI_Message_Out.Full_SPI_Msg); // Send out the SPI message.
 };
 // Prepare to sleep.
```

```
 enable_interrupts( INT_RA2_H2L); // This interrupt brings the device out of sleep
 // Go into sleep mode.
 sleep();
 disable_interrupts( INT_RA2_H2L); // Disable this interrupt so it does not continue to trigger
while sending out the SPI messages.
 clear_interrupt(INT_RA2_H2L); // Clear the interrupt.
 First_Time_Sleep = False; // Mute the DAC when the pin goes back high.
 // Now send out the mute message.
 SPI_Message_Out.SPI_Byte.Read_Write = Global_Address_write;
 SPI_Message_Out.SPI_Byte.Reg_Addr = DAC_Mutes;
 SPI_Message_Out.SPI_Byte.Data = 0xFF;
 SPI_Message_Out.SPI_Byte.Unused_Byte = 0;
 spi_xfer(SPI_Message_Out.Full_SPI_Msg); // Send out the SPI message.
 delay_ms(3); // Insert a delay for switch debouncing. If the MUTE pin is tied to a
microcontroller then this is not needed.
 };
 }
}
```

自引导程序头文件

```
#include <12LF1572.h>
//#device ADC=16
#FUSES NOWDT //No Watch Dog Timer
#FUSES NOBROWNOUT //No brownout reset
#FUSES NOLVP //No low voltage programing, B3(PIC16) or B5(PIC18) used for I/O
#use delay(internal=16000000)
// use software SPI
#use spi(DI=PIN_A5, DO=PIN_A4, CLK=PIN_A0, ENABLE=PIN_A1, BITS=24, IDLE=1)
// The "USE" statement sets up the software SPI built in compiler functions.
// It sets up the pins used and for this case we need 24 bits for each SPI transmission.
// It is important that the enable (CLATCH) remain low until all 24 bits are transmitted.
// Then the "IDLE" setting sets the state of the clock when at idle. For this part it
// wants to have the clock be high at idle so that is why it is set to 1.
#define Global_Address_Read 0x09 // This is the chip address 4 shifted left one bit plus the R/W
bit set to read.
#define Global_Address_Write 0x08 // This is the chip address 4 shifted left one bit plus the
R/W bit set to write.
// Register Addresses:
#define PLL_Control_0 0
#define PLL_Control_1 1
#define DAC_Control_0 2
#define DAC_Control_1 3
#define DAC_Control_2 4
#define DAC_Mutes 5
#define DAC_Vol_L1 6
#define DAC_Vol_R1 7
#define DAC_Vol_L2 8
#define DAC_Vol_R2 9
#define DAC_Vol_L3 10
#define DAC_Vol_R3 11
#define DAC_Vol_L4 12
#define DAC_Vol_R4 13
#define ADC_Control_0 14
#define ADC_Control_1 15
#define ADC_Control_2 16
```


AN13226sc-0-8/15(0)