AN-1365
APPLICATION NOTE

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 Boot Up Application

by David M. Thibodeau

INTRODUCTION

The AD1933/AD1934/AD1938/AD1939/AD1974 family of codecs, ADCs, and DACs have a standalone mode of operation that allows them to be used without a microcontroller. Unfortunately, with the exception of the ADCs being a master or a slave, there are no options for different modes of operation in the standalone mode. Modern systems require faster sample rates, or TDM modes of operation, more frequently. It is desirable to be able to have converters boot up on their own in these advanced modes of operation where a system microcontroller is not present. This application note details a low cost solution for booting up these parts in any desired mode of operation.

For this example, a Microchip® PIC12(L)F1571/PIC12(L)F1572 is used to boot up the AD1938 and also provides for a mute function. This part is very low cost and is available through normal distribution channels. It is offered in a variety of packages to meet the cost/size needs of most applications.

For ease of reading, this application note uses the AD1938 as an example in the boot up application, but the application is functional with the entire AD1933/AD1934/AD1938/AD1939/AD1974 family.

The programming examples were written using the Custom Computer Services, Inc. (CCS) C language compiler.

This application note details the hardware and software used to provide this functionality.

TABLE OF CONTENTS

REVISION HISTORY

8/15—Revision 0: Initial Version

HARDWARE OVERVIEW

The AD1933/AD1934/AD1938/AD1939/AD1974 all have an SPI port for communicating with a system controller. They all use the same ID address embedded in the SPI command format so no changes to the code are required to communicate with any part in this family of parts. As a side note, this makes it possible to program multiple parts simultaneously because this program does not read any data from the codec. If there is a need for multiple parts programmed differently, then the user can modify the program to include an additional CLATCH (slave select) output.

The Microchip PIC12(L)F1571/PIC12(L)F1572 has six multifunction GPIO pins and a built-in oscillator block. All that is required for the part to function is power, ground, and a 0.1 µF bypass capacitor on the power pins. A pull-up resistor for the mute function is also recommended. The CLATCH line also requires a pull-up resistor to prevent the codec from entering the standalone mode prior to the microcontroller booting up. Only three passive components are required for this microcontroller application to function making it a very cost effective solution.

The PIC12F1571/PIC12F1572 is the 5 V version of the part and the PICLF1571/PICLF1572 is the low voltage version. Because the AD1938 is 5 V tolerant on the logic ports then either part can be used. However, it is recommended to use the PICLF1571/PICLF1572 version.

Figure 1 shows the GPIO and other functions of the microcontroller. Figure 2 shows the functions assigned to these pins in this example solution.

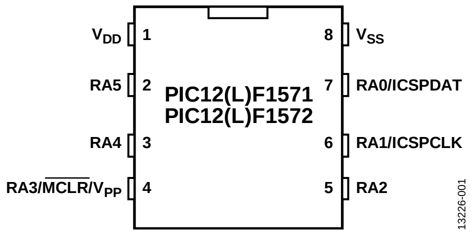

Figure 1. PIC12(L)F1571/PIC12(L)F1572 Pin Functions

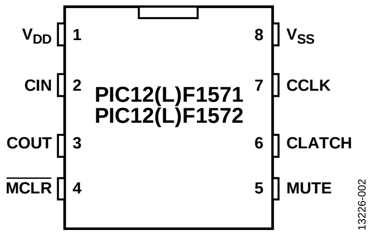

Figure 2. Functions Assigned to the GPIO Pins

PIN FUNCTION DETAILS

A list of the PICLF1571/PICLF1572 pin function details follows.

- CIN: This pin is a data input to the microcontroller from the AD1938 SPI output. Connect this pin to the COUT pin of the AD1938.
- COUT: This pin is the data output from the microcontroller and is connected to the CIN pin of the AD1938.
- $\overline{\text{MCLR}}$: This pin is the master clear function of the microcontroller. This pin has an internal weak pull-up and has a noise/spike filter built in. Pulling this pin down holds the part in reset and allowing the pin to float back high resets the part and starts the application program from the beginning. This allows a reset to be initiated by either a switch or a system controller. If unused, then leave this pin floating or tie it to V_{DD}.
- MUTE: This pin needs to be pulled high externally. This pin is active after initialization. Pulling it low mutes all the DAC outputs and remains that way until the pin returns to the high state. Then it transmits the unmute command.
- CLATCH: This pin needs to connect to the $\overline{\text{CLATCH}}$ pin of the AD1938. This pin needs to have a weak pull-up so that it is high during the start-up time right after power is applied. This prevents the AD1938 from entering standalone mode right after power is applied.
- CCLK: This pin is the serial clock. It needs to be tied to the CCLK pin of the AD1938.

Note that Analog Devices, Inc. has used different nomenclature for SPI port functions. See the SPI Port Naming Conventions sections for details.

The microcontroller is the SPI master.

SOFTWARE OVERVIEW

The software program flow is simple.

1. Setup the microcontroller internal oscillator and the port directions.
2. Set the $\overline{\text{CLATCH}}$ pin high.
3. Wait for the AD1938 to finish its initialization and calibration.
4. Call the routine that sends the SPI messages and configures the codec for the desired mode of operation.
5. Enter a forever loop.
   a. Go to sleep but set up a wake-up interrupt that triggers when the MUTE pin goes low.
   b. Upon wake-up, send the DAC mute SPI message then wait for the pin to go back high.
   c. When the MUTE pin goes high, send out the SPI message to unmute the DACs and then go back to sleep.

A few interrupt housekeeping steps are detailed in the code.

SOFTWARE FILES

There are three source code files for this program.

12LF1572.h: this is the header file specific to this microcontroller. This file has many useful definitions and built-in function prototypes. This file is supplied with the compiler.

AD1938 Self Boot main.h: this file holds compiler directives and useful definitions specific to this codec.

The compiler directives are to include the header file described previously, set some fuses in the part, set the delay time for the internal oscillator frequency being used. Then define the SPI functions. This program uses a software defined SPI implementation because the part does not have a hardware SPI port. See the Compiler Information section for details on the compiler used for this example.

The definitions provided for the codec are to set the command for reading or writing to the codec and register addresses.

SOFTWARE DETAILS

The data structure used is a union of two data types: a 32-bit integer and a structure of four 8-bit integers. This implementation allows for the addressing of the same data in two different ways. One way is a large 32-bit integer used by the SPI message function call. This call requires an integer variable. The second way is with a structure of four integer variables to allow the program to address the individual parts of a single SPI message. The SPI message format for the AD1938 is a 24-bit message. The software SPI function allows the programmer to define the number of bits transmitted in a message. The problem is that the C compiler does not have a 24-bit integer variable type so a 32-bit was used. The structure has one variable called unused_byte, which is a placeholder for these extra bits. Only the first 24 bits of the actual 32-bit variable are transmitted.

This program does not read any registers. The software is capable of reading data from the codec. To read, use the definition for reading data and once it is read the data is stored in the Data variable of the message structure. An example of how to read data is shown in the code comments (see the Code Listing section).

The codec powers up with all the registers set to zero. The data sheet lists this value first in the register details section. Any desired register setting that is a zero or appears first in the register details does not need to be written upon powering up the part. This simplifies the program by limiting the number of register writes required to set up the part in the desired mode.

COMPILER INFORMATION

The compiler used in this example is available from CCS. The compiler can be purchased with an integrated development environment but it is also offered in a command line form. The command line compilers are a very low cost solution that can be integrated into the MicroChip MPLAB® which is a free development environment offered by Microchip, Inc.

The choice of this compiler is due to the small file sizes produced by the compiler and the availability of many built in functions. This example uses the SPI built-in function, which implements a software SPI solution on parts where hardware SPI ports do not exist.

SPI PORT NAMING CONVENTIONS

Analog Devices has used different pin names (mnemonics) for SPI port functions. See Table 1 for details of the changes.

Table 1. SPI Port Pin Naming Conventions

Legacy Pin Mnemonic	New Pin Mnemonic
CDATA	MOSI
COUT	MISO
CCLK	SCLK
$\overline{\text{CLATCH}}$	$\overline{\text{SS}}$

The C in the legacy names stands for communication. The new names that are used elsewhere in the industry stand for:

- MOSI: Master output slave input
- MISO: Master input slave output
- SCLK: Serial clock
- $\overline{\text{SS}}$: Slave select which is normally an active low.

TIMING DETAILS

When power is applied, there is approximately a 65 ms wake-up time for the microcontroller before its code begins execution. This occurs well before the codec is ready to be programmed so the microcontroller waits for an additional 265 ms. The resulting time from power-up to when the SPI messages are sent is around 330 ms. See Figure 3 for details. Figure 4 to Figure 8 detail other signal timings.

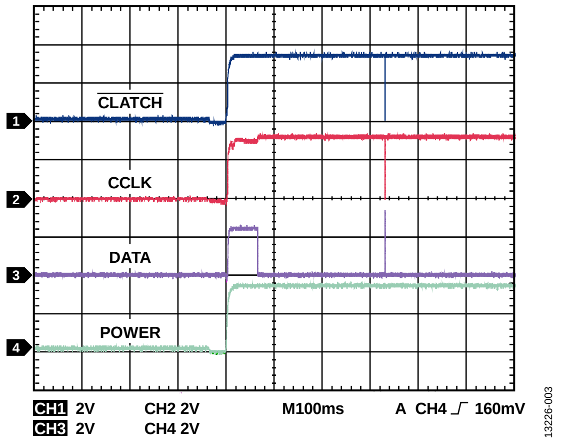

Figure 3. Start-up Time from Power Up to SPI Message Transmission

The SPI clock frequency is approximately 144 kHz as shown in Figure 4.

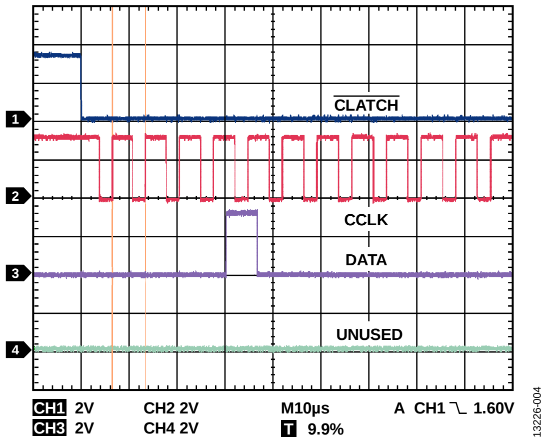

Figure 4. SPI Clock Frequency = 144 kHz

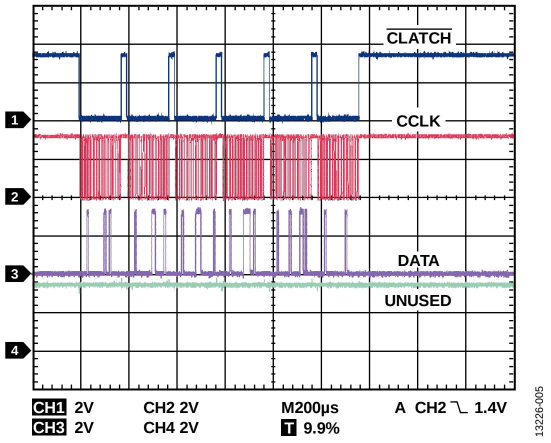

Figure 5. Six Outgoing SPI Messages Used to Configure the Codec

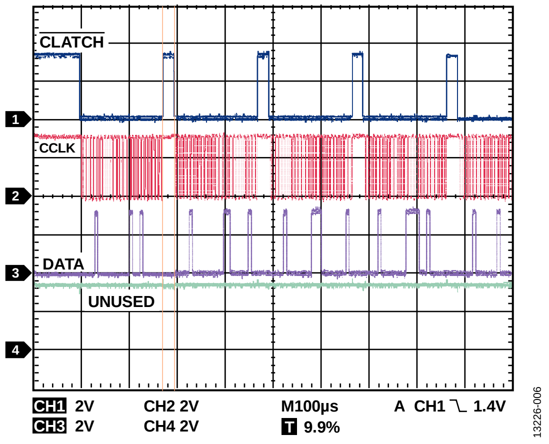

Figure 6. Approximately 25 µs of Time Between Messages

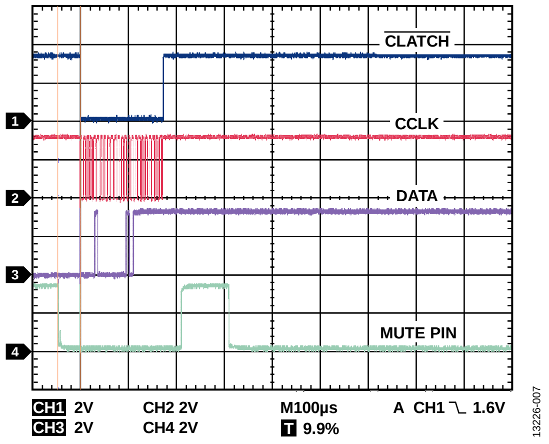

Figure 7. Time from MUTE Pin Going Low and Mute Message Transmission = 46 µs

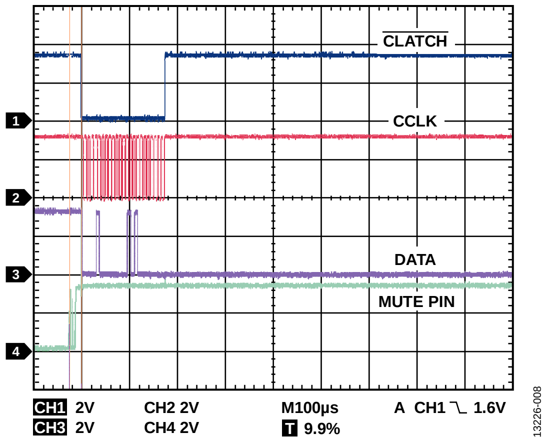

Figure 8. Time from MUTE Pin Going High and Unmute Message Transmission is 25 µs

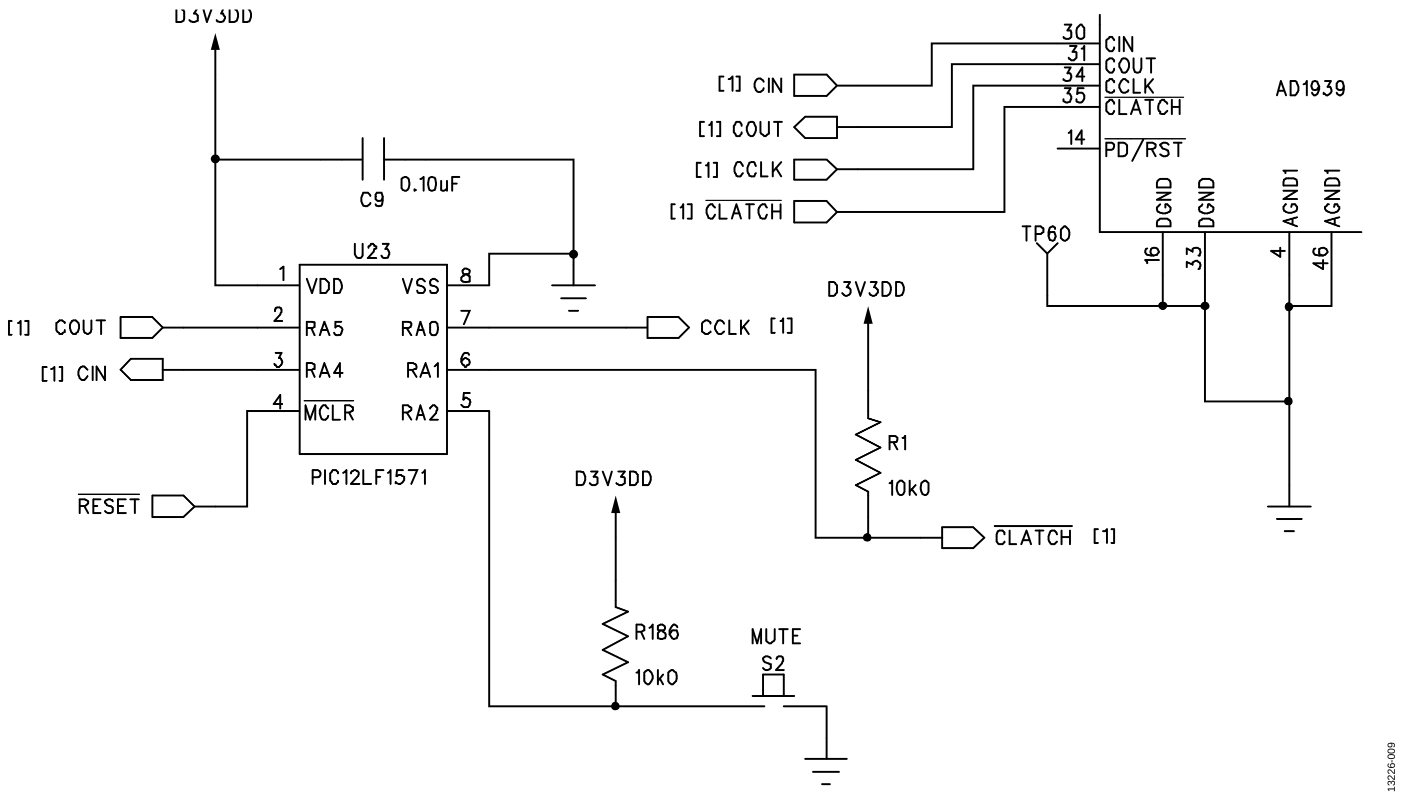

Figure 9. Example Schematic Implementation

CODE LISTING

Main Program

To save boot-up time, it is good to know that the AD1938 parts come out of reset with all the registers at zero. The data sheet shows that the default values as the first value listed in the register details; these defaults are the zero settings. Therefore, this program only needs to change the registers that should not be zero for the desired mode of operation.

Another limitation of this program is that the built-in software SPI routines do not allow large block writes while the CLATCH line is held low. So the feature of the part that allows automatic address incrementation cannot be used.

Reading Data Back

The following example shows how to read data back from the part:

```
#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.
 };
 }
}
```

Self-Boot Program Header File

```
#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
```