AD7124 USER GUIDE

with Simblee Microcontroller

NICHOLAS P. NEWMAN, DARTMOUTH COLLEGE

August 18, 2017

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Using the Simblee to interface with the AD7124-4 is somewhat complicated and finicky but these libraries provide an API that allows implementation of higher-level functions that can be used without in-depth knowledge of the underlying processes. The current iteration of the library is designed to work in single-ended, continuous-conversion, 10ksps mode with an internal reference voltage. Documentation can be found in this primer for how best to adjust configuration options and verify correct operation.

1 INSTALLATION

1.1 Library Files

The relevant files can be found at the *AD7124-4_Simblee* GitHub repository. Contributions are more than welcome! Feel free to fork the repository and add your own functions or configurations. The following files should be downloaded and added to an appropriately named folder within the standard Arduino library folder:

- Communication.cpp
- Communication.h
- AD7124.cpp
- AD7124.h
- AD7124_regs.cpp
- AD7124_regs.h
- adspi.cpp
- adspi.h
- keywords.txt

The standard Arduino libraries folder can be found at the default location of the sketchbook repository.

The usual locations for the user Arduino libraries can likely be found at the following locations based on platform. Filepaths may change based on operating system and/or Simblee package iteration. A good overview of how libraries are structured and installed can be found at `https://www.arduino.cc/en/Guide/Libraries`.

1.2 Hardware Files

The Simblee hardware package needs to be installed for the Arduino IDE to successfully compile and link the source code and upload it to the microcontroller. The Arduino website has a good step-by-step manual for how to download generic packages at `https://www.arduino.cc/en/Guide/Libraries`. Simblee provides a good multi-platform quick start guide that can be found at `https://www.simblee.com/Simblee_Quickstart_Guide_v1.1.0.pdf`.

2 MODIFICATIONS

2.1 SPI (Hardware)

The SPI library needs to be modified to include the new function *transfer24()* and the corresponding prototype. Once the Simblee library has been downloaded, and installed through the Arduino interface, navigate to the following directory based on platform:

WINDOWS `C:/Users/User/AppData/Local/Arduino15/Packages/Simblee/hardware/Simblee/1.1.3/libraries/SPI`

MAC `HD/Users/User/Library/Arduino15/Packages/Simblee/hardware/Simblee/1.1.3/libraries/SPI`

LINUX `.../portable/packages/Simblee/hardware/Simblee/1.1.3/libraries/SPI`

Then either download the SPI files (SPI.cpp and SPI.h) from the GitHub repository and substitute them for the old version or add the following items to the

original files:
In SPI.cpp. . .

```
uint32_t SPIClass::transfer24(uint32_t _data)
{
    uint8_t highByte, midByte, lowByte;

    if(spi->CONFIG && 0x01)  //LSB first
    {
        spi->EVENTS_READY = 0;          //clear ready event
        // SPI is triple buffered
        spi->TXD = _data & 0xFF;      //lower 8 bits
        spi->TXD = (_data >> 8) & 0xFF;      // middle 8 bits
        spi->TXD = (_data >> 16) & 0xFF;       // upper 8 bits
        while (spi->EVENTS_READY == 0) ; // wait for ready event
        spi->EVENTS_READY = 0;          //clear ready event
        lowByte = spi->RXD;
        while (spi->EVENTS_READY == 0) ; // wait for ready event
        spi->EVENTS_READY = 0;          //clear ready event
        midByte = spi->RXD;
        while (spi->EVENTS_READY == 0) ; // wait for ready event
        spi->EVENTS_READY = 0;          //clear ready event
        highByte = spi->RXD;
    }
    else                            //MSB first
    {
        spi->EVENTS_READY = 0;          //clear ready event
        // SPI is trpible buffered
        spi->TXD = (_data >> 16);        // upper 8 bits
        spi->TXD = (_data >> 8);        // middle 8 bits
        spi->TXD = _data & 0xFF;      //lower 8 bits
        while (spi->EVENTS_READY == 0) ; // wait for ready event
        spi->EVENTS_READY = 0;          //clear ready event
        highByte = spi->RXD;
        while (spi->EVENTS_READY == 0) ; // wait for ready event
        spi->EVENTS_READY = 0;          //clear ready event
        midByte = spi->RXD;
        while (spi->EVENTS_READY == 0) ; // wait for ready event
        spi->EVENTS_READY = 0;          //clear ready event
        lowByte = spi->RXD;
    }

  uint32_t data = (highByte << 16) | (midByte << 8) | lowByte;

  return data;
}
```

In SPI.h. . . in public prototype declarations:

```
uint32_t transfer24(uint32_t data);
```

2.2 AD7124 (No-OS)

The *Communcation* and *AD7124* libraries are provided as-is by Analog Devices and are available at https://wiki.analog.com/resources/tools-software/uc-drivers/ad7124. The *Communication* library functions are entirely populated by N. Newman and are designed specifically for the Simblee. That is, several of the configuration options and arguments that are available in the original file are omitted for conciseness. For other microcontrollers, this library may need to be adjusted to either include these arguments or change hard-coded configuration options specific to the Simblee.

The AD7124 remains functionally unmodified to allow users who intend to develop additional library use cases to remain on the same page as other developers. Additional functionality is provided by the adspi library. However, the current Arduino compiler (*avr-g++*) does not support implicit iteration through enums, an operation that the No-OS software uses several times. The AD7124 library provided in the GitHub repository includes a fix for this by adding a global iteration variable that is statically cast as an enum to allow indexing.

The changes are detailed explicitly below:

```
for(regNr = AD7124_Status; (regNr < AD7124_Offset_0) && !(ret < 0){
  regNr++
  ...
```

becomes

```
for(regInt = AD7124_Status; (regInt < AD7124_Offset_0) && !(ret < 0); regInt
    ++)
  {
    ad7124_registers regNr = static_cast<ad7124_registers>(regInt);
    ...
```

3 IMPLEMENTATION

This guide is written specifically for developers using the Arduino IDE. However, the hardware package from Simblee can be used with few modifications to work with command line tools. The following examples are Arduino-specific, as the adspi library relies on several functions from the standard *Arduino.h* library.

Important preliminary steps:

1. Configure SPI pins (MOSI, MISO, etc.)

   - The pin definitions are located in the *variant.h* file, which itself is located in the SPI hardware package (detailed in 2.1), at /variants/Simblee. You can changed the default defitions:

   ```
   #define PIN_SPI_SCK    (4u)
   #define PIN_SPI_SS     (6u)
   #define PIN_SPI_MOSI   (5u)
   #define PIN_SPI_MISO   (3u)
   ```

     by editing any of the (Xu) pin numbers. Further information can be found in the Simblee User Guide located at https://www.simblee.com/Simblee%20User%20Guide%20v2.06.pdf.

   - In *adspi.h* the ADSPI_CS pin is declared with a backup #ifndef, which is used to ensure the definition has a value. The CS (Chip Select) pin requires very specific timing and behavior for several functions, and therefore is declared a 'digital out' and is manually toggled by several functions in *adspi.cpp*.

2. Decide on a Serial baud rate

   - The Simblee supports standard UART baud rate values ranging from 300 to 115200. During testing when timing is critical and serial writes can occupy a substantial amount of available clock pulses, it is recommended to use the highest baud rate possible. This matters less during final implementation when the UART will likely go unused. (Examples will, however, use Arduino standard 9600, as example applications with Serial prints are not time critical.)

3. Configure the AD7124 registers

   - The registers on the AD7124 chip dictate how it operates. In-depth explanation and examples will be provided in 3.1.

3.1 Setup

3.1.1 *Register Configuration and Wake-Up*

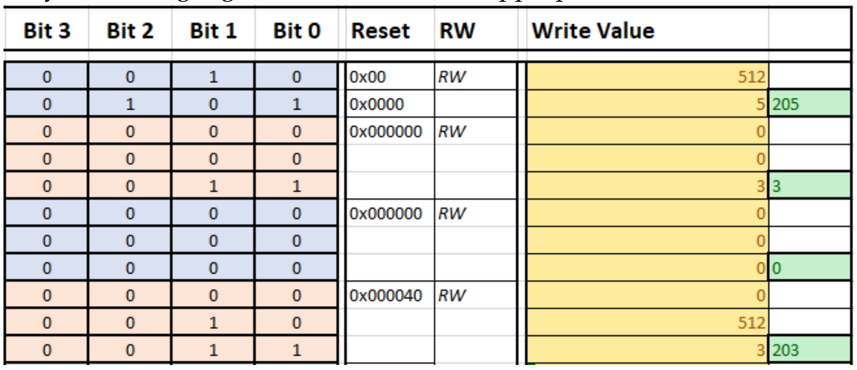
The first function to call in the *setup* section of a sketch is adspi.begin(), which sets up the SPI channel, initializes all AD7124 registers, and tells the ADC to start producing conversions.

The registers on the AD7124 dictate every facet of operation, including ODR, input channels, filering, and gain. The function of each register and how to configure them for certain options is detailed in the AD7124-4 datasheet, located at `http://www.analog.com/media/en/technical-documentation/data-sheets/AD7124-4.pdf`. Several example register setups can be found on the GitHub repository, but it is certainly worth knowing how to configure them individually. This is the singular instance where the end-user is required to modify the contents of a library file. The No-OS software combined with the tools provided by N. Newman make this a relatively painless process.

EXAMPLE REGISTER CONFIGURATION The following steps will detail a hypothetical situation where a user would like to manually change the analog inputs from {AIN1, AIN2} to {AIN3, AIN4} and the IC mode from continuous-conversion to standby.

1. (optional) Download the excel macro tool *RegEditTool* from the GitHub repository.

   - This macro tool allows for rapid register configuration changes by automatically converting register bit values to the appropriate hexadecimal write value.



Bit 3	Bit 2	Bit 1	Bit 0	Reset	RW	Write Value	
0	0	1	0	0x00	RW	512	
0	1	0	1	0x0000		5	205
0	0	0	0	0x000000	RW	0	
0	0	0	0			0	
0	0	1	1			3	3
0	0	0	0	0x000000	RW	0	
0	0	0	0			0	
0	0	0	0			0	0
0	0	0	0	0x000040	RW	0	
0	0	1	0			512	
0	0	1	1			3	203

2. Open the *AD7124_regs.cpp* file.

3. Calculate required values for register writes. In this example, we want to change the analog inputs from {AIN1, AIN2} to {AIN3, AIN4}. Looking at the datasheet shows we need to change the CHANNEL register bits [9:0] from `0b0000000001` to `0b0000110100` (hex `0x001` to `00x64`). The same process shows we need to change the ADC_CONTROL register bits [5:2] from `0x0` to `0x2`.

4. Substitute existing write values in the struct (second column) with the values calculated by the tool. The *adspi.begin()* function automatically writes all values in this struct to the ADC registers.

NOTE On-the-fly register configurations not included in the adspi library can be acheived in two ways: the *adspi.command()*, which bypasses the register struct provided by the AD7124 lib and writes to the register directly. The *adspi.regEdit()* function modifies the contents of the struct, which can then be written with the *adspi.updateRegs()* function. It is important to note that adspi library register edit commands bypass the register struct, meaning upon restart and rewrites, those changes will be lost.

3.1.2 *Bipolar & Unipolar Operation*

The AD7124 can be configured to accept both unipolar and bipolar input voltages. Regardless, the chip cannot accept negative voltages with respect to chip ground (V_{ss}).

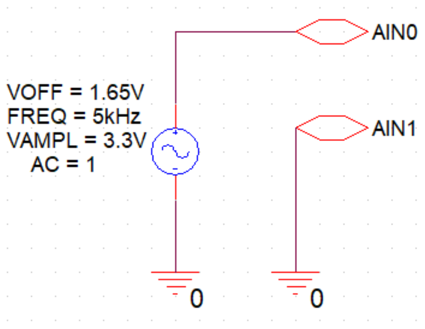


Figure 1: Unipolar Channel Diagram

UNIPOLAR Unipolar operation will read single-ended inputs and output positive-only conversions. Because the DC offset of the input signal is included, the PGA cannot be used as effectively, as any gain will also apply to the offset, usually saturating the ADC. This works best for PGA set to 1 (low-power, low-current), but as the outputs of some sensors are differential signals centered around some DC voltage, a bipolar input would be more appropriate. Unipolar conversions are recorded as

$$\text{Code} = \frac{(2^N \times A_{IN} \times \text{Gain})}{V_{ref}}$$

which corresponds to the decoding function implemented in the adspi library:

```
double adspiClass::D2V_Sing(int32_t val)
{
  double bits = pow(2,24);
  double V_ref = 3.3;
  double gain = 1;

  return ((val * V_ref) / (bits * gain));
}
```

I recommend using unipolar mode in single-ended configuration with low-frequency, no-gain applications. It works well for low-power systems and systems that do not require AGC.

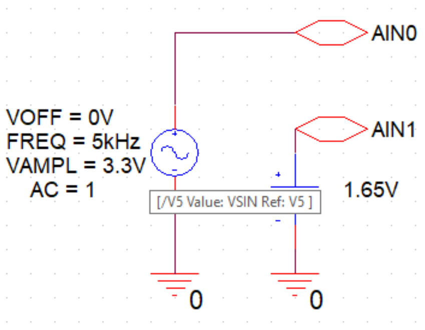


Figure 2: Bipolar Channel Diagram

BIPOLAR In systems that require AGC or have offset analog signals, bipolar mode is likely the best option. The negative input (AIN1) should be connected to the offset voltage (in a 3.3V system, 1.65V) and the positive input (AIN0) to the input

voltage. To avoid offset errors, the user should provide a stable negative input, and not use the internal reference of the AD7124. The chip itself *does* provide the option to connect an external input to an internal source (such as $\frac{V_{dd}}{2}$), but offset error then becomes an issue. See the channel register configuration options for more information.
The bipolar input configuration makes conversions using the differential code system, namely:

$$\text{Code} = 2^{N-1} \times ((A_{IN} \times \frac{\text{Gain}}{V_{ref}}) + 1)$$

and uses the decoding function in the adspi library:

```
double adspiClass::D2V_Diff(int32_t val)
{
  double bits = pow(2,23);
  double V_ref = 3.3;
  double gain = 1;

  return ((val/bits) - 1)*(V_ref / gain);
}
```

3.1.3 *External Inputs*

The AD7124 accepts override values for several reference voltages and signals that can be provided by the developer.

POWER SOURCE AND REFERENCE VOLTAGES The AD7124 V_{dd} and V_{ss} voltages are determined by connecting the appropriate terminals to the power sources. Split power supply operation is possible and produces better noise reduction and can accept truly bipolar inputs. However, for simplicity, both the analog and digital portions of the chip will use a single power supply in this example. The AD7124 can provide several different reference voltages for performing conversions. Low-power applications in single-ended mode may consider using the 2.5V internal source provided by the chip. Most setups in the examples default to using the internal V_{dd} reference. The AD7124 also accepts external reference voltages via the REFIN1 and REFIN2 pin inputs. The configuration bits for these options are [4:3] in the configuration register.

CLOCK AND SPI The adspi library provides a register level function for initializing an external clock for the AD7124. The benefits of using an external clock are reduced power consumption, reduced jitter, reduced clock frequency error, and the ability to chain several ADCs together for multiple conversions. The AD7124 divides an external clock signal frequency down by four, so the adspi *start_exclk(int pin)* function produces a 4MHz signal that is divided down to 1MHz internally. The internal clock of the AD7124 is a 614.4kHz clock with a ±5% tolerance. This clock is recommended for high-power mode operation and sampling rates greater than 1ksps. The external clock pin on the IC should be connected to the appropriate pin on the microcontroller.
The Simblee SPI library is automatically configured to run at 4MHz and the GitHub files already have correct default values. If beginning with fresh SPI files, make sure to set the data mode to 3 in the *SPI.begin()* function.

MULTIPLE CHANNELS & SETUPS Multiple channels can be configured to run simultaneously by the AD7124. Up to eight different setups (analog inputs, gain, filter, data rate, etc.) can be configured and stored via register editing, and the ADC can step through four of them simultaneously. Because this *is* an multiplexed system, the total output data rate can never exceed 19.2ksps. The examples provided in the Git repository use only one setup and channel. Different setup configurations can be accessed by using one of the example AD7124_reg.cpp files or modifying one yourself. See figures 71 & 72 in the datasheet for more information.

3.2 Arduino IDE

BASIC FUNCTION CALLS There are several crucial functions that must be used for basic operation of the AD7124. These include functions required for setup, configuration, and other standard operations. These functions should not be used within continuous-read operation.

Serial.begin() This function is required for any communication with the computer itself for debugging and data output purposes. Unlikely to be used in final application. This belongs in the setup method.

adspi.begin() This function initializes everything required for operating the AD7124. It sets up the SPI channel, writes the values in AD7124_regs.cpp to the on-chip registers, and tells the ADC to begin producing conversions. After major changes to the registers without actually restarting the chip, it is recommended to use the *RegEdit()* and *UpdateReg()* in concert instead of the *adspi.begin()* function. This belongs in the setup method.

adspi.print_regs() This function reads all of the values written to the AD7124 registers and prints them out via the serial port. It is a good way to ensure the correct values were written and check the current state of the IC.
NOTE! Do not use this function while conversions are being performed. First set the chip to standby mode, otherwise the rapid register reads will conflict with conversions, resulting in a CRC error. A good way to use is immediately after the emphadspi.begin() function.

adspi.print_data_wStatus() This function is a simple way to check low data-rate conversions in continuous-conversion mode. It monitors the status register until a conversion is ready to be read, and then reads the data register. Then it outputs the calculated voltage via the serial monitor.
NOTE! This function uses the serial interface, meaning the very slow Serial.fxs are called, dramatically reducing the effective output data rate. This can be mitigated by increasing the serial baud rate, but do not rely on this function for final operation. Best used for development and verifying correct conversion operation.

adspi.SetPGA() This functions sets the gain of the system. Using the on-board PGA, gains can be set in increments of 2^n for $0 < n < 7$: (1 - 128). Factory reset of the gain is 1. Gains other than 1 should only be used with bipolar (differential mode) as otherwise gaining the DC offset of a single-ended input would likely result in saturation.

adspi.Standby() TODOdebug This function puts the AD7124 in standby mode. Standby mode is ready for further SPI communication but does not perform conversions and turns off most of the resources in the IC. Standby mode is a very low power mode.

adspi.PowerDown() TODOdebug Powers down the AD7124. All resources, including current, voltage, and other reference sources are powered down. Will not respond to standard adspi library functions until the adspi.begin function is called again.

adspi.Reset() Resets the device to factory standard. This function rewrites all the registers to the factory standard (visible in the AD7124 datasheet).

4 ERROR REPORTING

4.1 No-OS Error Checking

The AD7124 includes support for numerous types of error checking, and the No-OS software by Analog automatically checks one (CRC checksum error) as well as two additional checks specific to the No-OS software. The adspi library is designed to print error codes as they arise. The three error codes you may encounter are:

1. Error Code: -1

   - Invalid Argument. You should never see this error, as the adspi library wraps the AD7124 library and manages all the inputs internally. Seeing this means an adspi function has a bug.

2. Error Code: -2

   - CRC Checksum Error. This is the only error you may expect to see during regular operation. It occurs when the number of clock pulses from the microcontroller SPI communication does not match the number required for the expected operation. This usually occurs when the user tries to access registers during continuous-read mode without first exiting the mode. It can also happen if a conversion is being produced and the user does a full-system register reset. The best way to avoid this is to enter standby mode before full-system calibrations, large-scale register reads/writes, and before and after continuous-read mode.

3. Error Code: -3

   - Timeout Error. The microcontroller will automatically attempt to execute any function 1000 times before returning a timeout error. This will occur if the device is powered down (different from power-down mode, literally *without power*). Incorrect pin connections (SPI, power, external clock) can produce this error as well.

4.2 Additional Error Checking

In order to configure additional error checking, use the register edit tool and the AD7124_regs.cpp file to enable different types of error checks. Then monitor the status register using the *adspi.status()* function. If the error bit is flipped, use the *adspi.error()* function to read the error register and see which bit has flipped, corresponding to an error. TODO *adspi.getError()*.