Technical Guide
CCRTAICC (WC-ADS6418)

PCIe 64-Channel Analog Input Card (AICC)

Driver	ccrtaicc (WC-ADS6418)	
OS	RedHawk (CentOS or Ubuntu based)	
Vendor	Concurrent Real-Time	
Hardware	PCIe 64-Channel Analog Input Card (CP-ADS6418)	
Author	Darius Dubash	
Date	January 25th, 2021	Rev 2021.1

This page intentionally left blank

Table of Contents

This page intentionally left blank

1. Introduction

This technical guide provides an insight into the workings of the various components of the FPGA card. Several example programs supplied in the installed driver's *test* directory can assist the user in developing their applications. The board is comprised of the following features:

- Analog to Digital (ADC) conversion
- SDRAM *(currently not supported by this card)*
- Clocks
- Calibration
- Serial Prom *(currently not supported by this card)*

2. Analog to Digital (ADC) Conversion

The ADC has 64 channels with 18-bit resolution, controlled by four ADC converters; each can be assigned one of four update clocks, or four inverted update clocks and can have as input either an external signal or calibration bus. Both *single-ended* or *differential* inputs are supported.

ADC to channel association is as follows:

ADC 0 – Channel 0 to 15
ADC 1 – Channel 16 to 31
ADC 2 – Channel 32 to 47
ADC 3 – Channel 48 to 63

Prior to performing any conversion, the ADC converter needs to be activated with the *ccrtAICC_ADC_Activate()* API call. Without this activation, all other ADC calls will fail.

There are two mechanisms implemented by the hardware to enable the user to acquire analog signals. The ADC channels can be read from either 64 channel registers that are updated at the selected clock rates or an ADC FIFO that is 128K samples deep. Each ADC FIFO sample will also contain the channel number associated with the sample. Either of these approaches can be used to acquire digital samples from the channels. The ADC FIFO approach of course captures all the samples at the selected clock rates as long as no overflow condition occurs.

- ADC Channel Registers
- ADC FIFO

Prior to any data being collected, the user needs to configure each ADC in order to select one of 4 individual clocks (0 to 3) as the input signal, they can also select one of 4 inverted clocks (0 to 3) which are the same normal clocks inverted. The input signal can be either external inputs (normal mode), or calibration bus (for debug and calibration). Additionally, the onboard clock generator needs to be programmed with the selected ADC clock(s) at the user desired data collection rate. Each of the four individual ADCs can also be programmed with data format of *offset binary* or *two's complement* and a *bipolar* voltage range of either 5 or 10 volts or a *unipolar* voltage range of either 5 or 10 volts. Additionally, the ADCs can be set to *normal* or *high speed*. Normal speed is when you wish to sample below the 500KSPS clock rate and high speed when you want to select clock speed below 700KSPS. In this case, only even channels of the ADC will be active and odd channels will be unused.

2.1.1 ADC Channel Registers

This mechanism allows the user to *asynchronously* acquire *raw* data for any converted analog channel. Once the clocks have started *(after programming the ADCs and clocks)*, the board will continuously convert the ADC channels and update all the *Channel Registers* that have an active clock at the programmed clock rate. User can then asynchronously read any of the registers to acquire the latest converted *raw* data.

There are various methods available at the disposal of the user to receive the contents of the converted channel registers. Each has its own merit, limitations and performance impact and left to the sole discretion of the user as to the method to use.

a) Advanced users can access these registers directly via memory mapping, and bypassing the API, however, care must be taken in performing synchronization with any other applications accessing the board at the same time, since all safety locking will be bypassed. Failure to do so will result in unpredictable results.

   The memory mapped pointer *local_ptr* can be obtained by using the *ccrtAICC_Get_Library_Info()* call. Once the pointer is available, the channels can be accessed via the *ADC_Data[ ]* array.

   If the user wishes to determine the *floating point* voltages for the *raw* data, they can do so with the help of the *ccrtAICC_DataToVolts()* library call. This call requires as an argument a pointer to the *ccrtaicc_volt_convert_t* structure that holds the current ADC configuration information.

b) Alternatively, the user can use the *ccrtAICC_Fast_Memcpy()* library call to copy a consecutive set of *raw* channel registers contents to a local buffer.

c) Another method to transfer the contents of a consecutive set of *raw* channel registers to a local buffer is to use the *ccrtAICC_Transfer_Data()* library routine. The advantage of this call is that it allows the user to transfer the data via DMA or Programmed I/O. If this call is going to use DMA, then the received user buffer must be a buffer that can allow the board to perform DMA writes. This buffer can be obtained with the help of the *ccrtAICC_MMap_Physical_Memory()* library call.

d) Another approach is for the user to make use of the driver to acquire the contents of the ADC channels. In this case, the user needs to first select the appropriate channel read mode operation *(Programmed I/O or DMA)* with the *ccrtAICC_ADC_Set_Driver_Read_Mode()* library call and then call the *ccrtAICC_Read()* routine to read the *raw* channel registers. ***At present, the driver does NOT support DMA transfers.*** In this case *(i.e. PIO mode)*, any buffer *(not necessarily a DMA capable one)* can be supplied to the *ccrtAICC_Read()* call.

e) Finally, the *ccrtAICC_ADC_Read_Channels()* library call not only allows the user to select individual channels via a channel mask, but also returns the *raw* and *floating point* voltages as determined by the current configuration of ADC converters.

   The user has the option to supply a *NULL* pointer instead of the *adc_csr* argument, in which case the *ccrtAICC_ADC_Read_Channels()* call will internally extract the current hardware ADC configuration prior to computing the *floating point* voltage. This would add considerable overhead to the call if it is being called multiple times. Alternatively, the user could first determine the current ADC configuration using the *ccrtAICC_ADC_Get_CSR()* first and then supplying the current configuration to the *adc_csr* argument in the following *ccrtAICC_ADC_Read_Channels()* calls, with the assumption that the ADC configuration is not going to change for the duration of the reads.

2.1.2 ADC FIFO

This mechanism allows the hardware to *synchronously* acquire the *raw* data for any converted analog channel. Once the ADCs and clocks have been programmed and started, the board will continuously convert the selected ADC channels and place them in the *ADC FIFO* at the programmed clock rate. The user can select which channels are to be sampled by the hardware and placed in the *ADC FIFO* with the channel selection mask supplied to the *ccrtAICC_ADC_Set_Fifo_Channel_Select()* call.

User can then asynchronously extract the samples from the *ADC FIFO* via several methods. Care must be taken to ensure that the *ADC FIFO* does not get empty *(underflow)* or go beyond full *(overflow)*, otherwise synchronous data collection will be compromised. At any time, the *ccrtAICC_ADC_Get_Fifo_Info()* call can be invoked to determine the status of the *ADC FIFO*.

Unlike the samples in the *ADC Channel Registers* which only contain the *raw* 18-bit sample data, the *ADC FIFO* samples contain the *raw* 18-bit channel data along with the channel number in the most significant byte associated with the channel in the 32 bit FIFO sample.

If the method to extract samples from the *ADC FIFO* is too slow, the user may consider either selecting fewer channels being scanned or reducing the sample collection clock rate.

This card can support channel clock speeds of up to 700KSPS. When operating at clock speeds between 500KSPS and 700KSPS, the user should set the ADC to *high speed* mode. This is to ensure that the hardware can keep up with the clock rate and this is accomplished by reducing the number of channel from 16 per ADC to 8 per ADC. In this case, only the even channels are active, while the odd channels are not used and not placed in the FIFO even if the user has selected the odd channels in the channel select mask.

Additionally, the users can use this FIFO mechanism to achieve 1MSPS per channel by dedicating two channels for the source signal. What is done in this case is to select two channels from different ADCs, e.g. 0 for ADC0 and 16 from ADC1. The ADCs could be in *normal* or *high speed* mode. The user would then assign a clock to ADC0 and an inversion of the *same* clock to ADC1. Now, when samples are collected, they will alternate between channel 0 and 16 in the FIFO. The user can then extract the samples from the FIFO and merge them into a single FIFO stream achieving the 1MSPS rate for the selected channel. There is no reason why a user could not similarly pair two channels that have both ADCs configured for *high speed* mode operating at the maximum clock speed of 700KSPS. In this case, the merged sample rate of the channels could be as high as 1.4MSPS.

The obvious drawback for operating channels at high speeds is that the software would not be fast enough to extract the FIFO samples and come back in time to extract the next set of samples without overflowing. In that case, the user has the choice of either reducing the number of channels being sampled and/or reducing the sampling frequency.

Prior to collecting the samples, it is recommended to reset the *ADC FIFO* to ensure that FIFO is empty. This can be accomplished by the *ccrtAICC_ADC_Reset_Fifo()* call.

Recommended method of data collection is to start the clocks, let them settle and then reset the FIFO just prior to starting sample collection.

There are various methods available at the disposal of the user to receive the contents of the converted channel samples from the *ADC FIFO*. Each has its own merit, limitations and performance impact and left to the sole discretion of the user as to the method to use.

a) Advanced users can access this register directly via memory mapping, and bypassing the API, however, care must be taken in performing any synchronization with any other applications accessing the board at the same time, since all safety locking will be bypassed. Failure to do so will result in unpredictable results.

   The memory mapped pointer *local_ptr* can be obtained by using the *ccrtAICC_Get_Library_Info()* call. Once the pointer is available, the channels can be accessed via the *ADC_FifoData* FIFO register.

   If the user wishes to determine the *floating point* voltages for the *raw* data, they can do so with the help of the *ccrtAICC_DataToVolts()* library call. This call requires as an argument a pointer to the *ccrtaicc_volt_convert_t* structure that holds the current ADC configuration information.

b) Another method to transfer the samples collected in the *ADC FIFO* to a local buffer is to use the *ccrtAICC_Transfer_Data()* library routine. The advantage of this call is that it allows the user to transfer the data via DMA or Programmed I/O. If this call is going to use DMA, then the received user buffer must be a buffer that can allow the board to perform DMA. This buffer can be obtained with the help of the *ccrtAICC_MMap_Physical_Memory()* library call.

c) Another approach is for the user to make use of the driver to extract the contents of the samples from the *ADC FIFO*. In this case, the user needs to first select the appropriate channel read mode operation *(Programmed I/O or DMA)* with the *ccrtAICC_ADC_Set_Driver_Read_Mode()* library call and then call the *ccrtAICC_Read()* routine to read the *raw* channel samples. ***At present, the driver does NOT support DMA transfers.*** In this case *(i.e. PIO mode)*, any buffer *(not necessarily a DMA capable one)* can be supplied to the *ccrtAICC_Read()* call.

2.1.3 ADC Input Options

Each of the four ADC's has the option of selecting its inputs either from the external lines *(normal mode)* or from the *calibration bus* with the *ccrtAICC_ADC_Set_CSR()* call. If external lines are selected for an ADC, all 16 ADC channels will return the *raw* digital values for the 16 inputs lines. If calibration bus is selected, then all ADCs can receive one of the following with the *ccrtAICC_Set_Calibration_CSR()* call:

- Calibration Ground
- Calibration Positive 10 Volt Reference Voltage *(really 9.91 volts)*
- Calibration Negative 10 Volt Reference Voltage *(really 9.91 volts)*
- Calibration Positive 5 Volt Reference Voltage *(really 4.95 volts)*
- Calibration Negative 5 Volt Reference Voltage *(really 4.95 volts)*
- Calibration 2 Volt Reference *(really 2.048 volts)*

The calibration connections are used for calibrating the ADCs. ***Note that all channels will display the same Calibration reference voltage, depending on the calibration bus selection.***

3. Reading and Writing to the card

This card has the ability to perform reads and writes to the card in four ways.

1. Programmed I/O
2. Basic DMA *(Direct Memory Access)*
3. Modular Scatter-Gather DMA

Of the four approaches, the programmed I/O is the slowest, however, it gives the user the ability to access any region on the card to read and write to it. The restrictions are of course, if a region is a read-only region then writes to it will not take place and vice versa. This approach also utilizes the most CPU and PCI bandwidth. It is good for small size read or write operations.

Basic DMA is faster than programmed I/O when a larger region is read or written to. It also has the advantage of reducing the CPU bandwidth since once the DMA operation commences, entire transfer occurs between the card and memory without CPU intervention. Since there is a finite setup time to initialize the DMA, it is only useful for large transfers as the overhead of setup would offset any gains for smaller transfers. Each call to the Basic DMA engine causes a single transfer read or write operation. You can also use interrupts to determine the end of transmission instead of polling. In latter case is faster response while the former uses less CPU overhead.

Modular Scatter-Gather DMA is similar to the Basic DMA with two differences. It is a lot faster that the Basic DMA and the user has the ability to setup multiple DMA accesses with a single call.

The following calls can assist the user in performing the I/O:

1. Programmed I/O
   - ccrtAICC_Fast_Memcpy()
   - ccrtAICC_Fast_Memcpy_Unlocked()
   - ccrtAICC_Fast_Memcpy_Unlocked_FIFO()
   - ccrtAICC_Transfer_Data()
   - ccrtAICC_Get_Mapped_Local_Ptr() // pointer to the card local memory - advanced users only
   - ccrtAICC_Read() // for reading ADC channels
   - ccrtAICC_ADC_Read_Channels() // for reading ADC channels
   - ccrtAICC_ADC_Read_Channels_Calibration() // for reading ADC channel calibration values
   - ccrtAICC_Get_Value() // to read specific values on the board registers
   - ccrtAICC_Set_Value() // to write specific values to the board registers

2. Basic DMA
   - ccrtAICC_Transfer_Data()

- ccrtAICC_DMA_Configure()
  ccrtAICC_DMA_Fire()

3. Modular Scatter-Gather DMA
   - ccrtAICC_Transfer_Data() // single MsgDma transfer
   - ccrtAICC_MsgDma_Seize() // single MsgDma transfer
     ccrtAICC_MsgDma_Configure_Single()
     ccrtAICC_MsgDma_Fire_Single()
     ccrtAICC_MsgDma_Release()
   - ccrtAICC_MsgDma_Seize() // multiple MsgDma transfer
     ccrtAICC_MsgDma_Configure_Descriptor()
     ccrtAICC_MsgDma_Setup()
     ccrtAICC_MsgDma_Fire()
     ccrtAICC_MsgDma_Release()

4. SDRAM *(currently not supported)*

Currently, this card does NOT support SDRAM.

This card includes a 256 Mega-Word SDRAM. Currently, no memory has been reserved for internal use.

Clock 7 is internally assigned to SDRAM by the hardware and it needs to be programmed and running at 10MHz prior to any SDRAM operation.

Once clock 7 is programmed and running, the SDRAM needs to be activated with the *ccrtAICC_SDRAM_Activate()* API call. Without this activation, all other SDRAM calls will fail.

The user can read or write to any word within the SDRAM with the use of the *ccrtAICC_SDRAM_Read()* and *ccrtAICC_SDRAM_Write()* calls respectively. All operations are word oriented.

4.1.1 SDRAM Read *(currently not supported)*

Currently, this card does NOT support SDRAM.

Typically a read operation consists of reading a set of words from a given word offset within the SDRAM. To perform this operation, first ensure that the SDRAM is in the read *incrementing* mode by setting the *read_auto_increment* argument in the *ccrtAICC_SDRAM_Set_CSR()* call to *CCRTAICC_SDRAM_READ_AUTO_INCREMENT_ENABLE*. This call need only be done once. The user can then issue the *ccrtAICC_SDRAM_Read()* with the word offset specified in *Offset* and the word size in *Size.*

Though the hardware allows the user to disable the auto incrementing of the read address, it is not normally used in this mode. If read auto incrementing is disabled, the same word will be read repeatedly.

4.1.2 SDRAM Write *(currently not supported)*

Currently, this card does NOT support SDRAM.

Typically a write operation consists of writing a set of words to a given word offset within the SDRAM. To perform this operation, first ensure that the SDRAM is in the write *incrementing* mode by setting the *write_auto_increment* argument in the *ccrtAICC_SDRAM_Set_CSR()* call to *CCRTAICC_SDRAM_WRITE_AUTO_INCREMENT_ENABLE*. This call need only be done once. The user can then issue the *ccrtAICC_SDRAM_Write()* with the word offset specified in *Offset* and the word size in *Size.*

Though the hardware allows the user to disable the auto incrementing of the write address, it is not normally used in this mode. If write auto incrementing is disabled, all the words will be written to the same offset within the SDRAM.

5. Clocks

This FPGA supports a total of ten clock generators Clock 0 to Clock 9. Following are their assignments:

- Clock 0 to 3 – for ADC
- Clock 7 – for SDRAM *(currently the card does not support SDRAM)*
- Clock 4, 5, 6, 8 and 9 – Reserved

Currently, users can select any of the four clocks (Clock 0 to 3) for ADC. They operate in either normal mode or inverted mode. Users can assign any combinations of the above four clocks *(normal or inverted)* for any of the four ADCs.

Clock 7 is only used by the SDRAM and must be programmed and running at 10MHz prior to performing any SDRAM operations. This is automatically done when the clock creating API is called.

Though there are several API calls to control the clock generator, it is recommended that they be left to the advanced users to control as they require in depth knowledge of the internals of the hardware and workings of the clock generator. For most users, the following API calls should suffice to handle most situations:

- *ccrtAICC_Reset_Clock()*
- *ccrtAICC_Compute_All_Output_Clocks()*
- *ccrtAICC_Program_All_Output_Clocks()*
- *ccrtAICC_Clock_Get_Generator_Info()*

Due to the complexity of programming the clock generator and due to hardware limitations *(different clocks sharing same resources)*, a user cannot *append* to or *change* already running clocks. If multiple clocks are to be used, then the user needs to program all the clocks with the single call prior to commencing. Additionally, the software makes all attempts to program the clocks with the user desired frequency. There may be times when the desired frequencies are so mismatched that it will be impossible for the clock chip to be programmed for those exact frequencies. In that case, the user has two choices (1) change the clock frequencies slightly (2) increase the supplied tolerance to the API call. In the latter case, the call will attempt to program the frequencies closest to what the hardware will allow.

5.1.1 Reset All Clocks

This call simply resets and disables *all* the clocks on the board. Not much can be done with the card until the clocks are programmed and running.

5.1.2 Compute All Output Clocks

Any of the ten clocks can be selected to be programmed with any frequency ranging from 1 Hz to 250 MHz. Since the clocks are sharing hardware resources, there may be certain frequency and clock combinations that will make programming the board impossible due to clock chip limitations. In this case, the user has the option to select fewer clocks, change the frequencies or increase the acceptable tolerance for desired frequencies.

The user can use the *ccrtAICC_Compute_All_Output_Clocks()* call to see if their combination of clock programming is going to work. No actual programming of the hardware takes place and therefore it should not interfere with any other hardware operation. If the call succeeds, it returns detailed information in the *AllClocks* argument for each of the clocks. Users can decide whether to program the clock generator with the same information using the *ccrtAICC_Program_All_Output_Clocks()* call.

5.1.3 Program All Output Clocks

This call first resets all the clock generators and then programs them with the desired frequencies supplied to the call. If any components *(e.g. ADC, or SDRAM)* are operational, they will no longer work until the corresponding clocks have been re-programmed. It is recommended to stop all components that are using the clocks prior to reprogramming the clock generators; otherwise, the component operation will be compromised.

5.1.4 Get Clock Generator Information

This call provides detailed information for any of the selected clock generators in the *CgInfo* argument of the *ccrtAICC_Clock_Get_Generator_Info()* call.

6. Calibration

For accurate representation of samples, users must perform calibration of ADC channels prior to sampling. ADC calibration makes use of either the on-board reference voltage or an external input.

6.1.1 ADC Calibration

The simplest way to calibrate all or a selected set of channels using the internal reference voltages, is to use the single call *ccrtAICC_ADC_Perform_Auto_Calibration()*. The calibration values assigned to channels will be directly impacted by the clock frequency, voltage range and normal/high speed of the ADCs. It is therefore required that the user set the individual ADCs properly prior to commencing calibration. The call requires a channel start and stop range, therefore individual channels can be calibrated without disturbing the calibration of other channels if so desired. When the call is successful, the offset, positive and negative calibration values for the selected channels have been calibrated.

External ADC calibration is more involved as the user needs to interactively supply the appropriate input signals. The user can perform external calibration by supplying zero volts signal to the selected channels and using the *ccrtAICC_ADC_Perform_External_Offset_Calibration()* call. Next, they can perform positive calibration by supplying an external positive signal to the selected channels and using the *ccrtAICC_ADC_Perform_External_Positive_Calibration()* call with the *ReferenceVoltage* argument set to the value of the external input signal and finally supplying a negative signal to the selected channels and using the *_ccrtAICC_ADC_Perform_External_Negative_Calibration()* call with the *ReferenceVoltage* argument set to the negative signal supplied.

If users prefer that the hardware not perform any calibration for specific channels, one can do that with the use of the *ccrtAICC_ADC_Set_Offset_Cal()* call with 0 volts offset and a gain of 1 for the *ccrtAICC_ADC_Set_Positive_Cal()* and *ccrtAICC_ADC_Set_Negative_Cal()* calls. Users can skip calibration data for channels being update by setting the corresponding channel with the *CCRTAICC_DO_NOT_CHANGE* flag instead. They can also use the *ccrtAICC_ADC_Reset_Calibration()* call to reset the calibration for all the channels.

7. Serial PROM *(currently not supported)*

Currently, this card does NOT support Serial PROM.

The board contains a *Serial Prom* that is 1024 short words (2048 bytes) deep. Information written to the *Serial Prom* is preserved and contains vital board information and should not be erased or changed by the user.

8. Mult-Board Clock Synchronization

Multi-Board clock synchronization provides the ability to connect several *CCRTAICC* analog input cards so that they can all be driven by a single input clock, thus achieving clock synchronization for all the connected cards. There are several configurations available to the user.

- A master board using one of its internal clocks as input to drive the clock signal to its clock output connector. This master clock output connector is then physically connected to the next boards clock input connector using a standard *shield* Cat5 Ethernet cable. Several cards can be connected in a daisy chain fashion re-driving its input clock to its output clock connector.
- Similar to the above hookup, except that instead of using the internal clock, the master board's input clock could be received from an external clock generator source.
- A clock generator source that has multiple LVDS outputs could connect to each cards input clock connector.

The two API calls that can be used to provide a software control to the synchronization are:

1. *ccrtAICC_Set_External_Clock_CSR()*
2. *ccrtAICC_ADC_Set_CSR()*

Although this card has the capability of being programmed with different clocks for each of the four ADCs *(16 channels per ADC)*, multi-board clock synchronization has only one physical line available to propagate a single clock to the next card. It is for this reason, that only one clock *(normal or inverted)* speed can be used to drive all the cards. Due to this restriction, the maximum clock speed when using *high-speed* channel setup will be limited to 700KSPS for a maximum of 32 channels and when using *normal-speed* channel setup will be limited to 500KSPS for a maximum of 64 channels. This is due to the fact that double the above speeds can only be achieved by driving different ADCs with a non-inverted clock on one ADC and its inverted clock on the other ADC, i.e. requiring *two* clocks to be propagated to all the cards.

The user can use any one of the four onboard clocks as input to the master board or an external clock. When using an external clock, there is no capability to disable/enable external input clocking as that would be an asynchronous event independent of the clock cycle. Thus, there would be no guarantee to the position of the clock when the sampling starts for each card. For this reason, when synchronizing multiple boards with a master board using an external input clock, the user needs to first setup all three cards so that they are all waiting for samples and then a clean clock signal starting with a low to high transition needs to be applied. This will ensure that all three cards are properly synchronized.

Another point to note is that after the onboard FIFO is reset, the first ***two*** clock cycles *(whether using the on board clock or an external clock)* are ***discarded*** before sample connection starts.

8.1 Example setup and display

This example demonstrates multi-board clock synchronization:

Three cards are connected with the output clock on the master board 0 being connected to the input clock on the first slave board 1, the output of the slave board 1 is connected to the input of slave board 2. A precision signal generator is connected to channel 14 of all three cards generating a 14KHz, +/-9.5V Sine wave. A second output from the signal generator is connected to channel 16 of all three cards also generating a 14KHz, +/-9.5V Square wave phase shifted by 180 degrees.

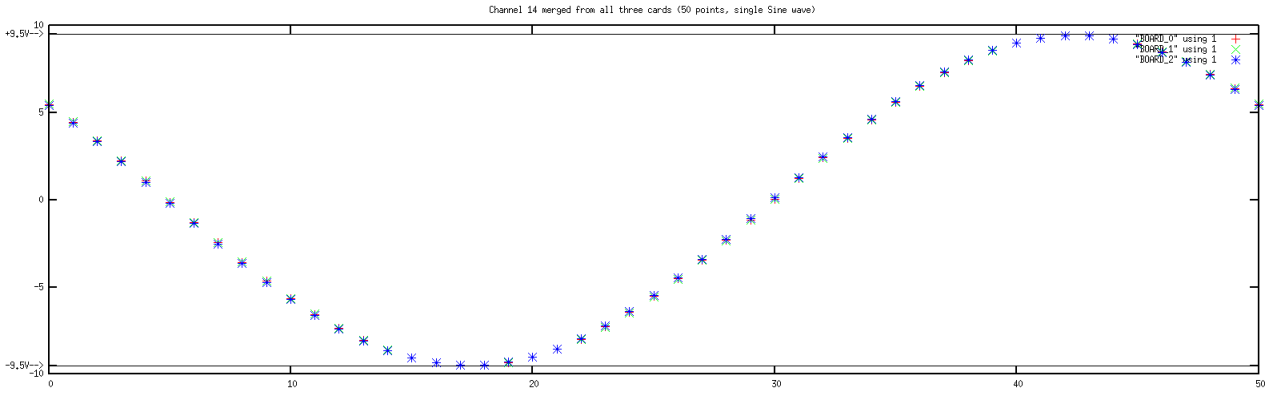
The master board 0 uses its internal clock 0 to drive all three cards at a maximum clock speed of 700,000SPS. All cards are first calibrated with this clock speed, +/-10 V range, high-speed channel selection and all output clocks set to "no clocks"

Next, slave board 2 is started as a background task with its input clock set to external input. The slave board 1 is then started as a background task with its input clock set to external input and its output clock set to pass-through its external input clock to its output *(redrive)*

The script sleeps for one second before starting the board that is going to generate the master clock so as to give time for the boards 1 and 2 to be initialized and waiting for external clock to be received.

Finally, board 0 with the master clock is started with its internal clock 0 being sent to the output clock line which should be received by boards 1 and 2

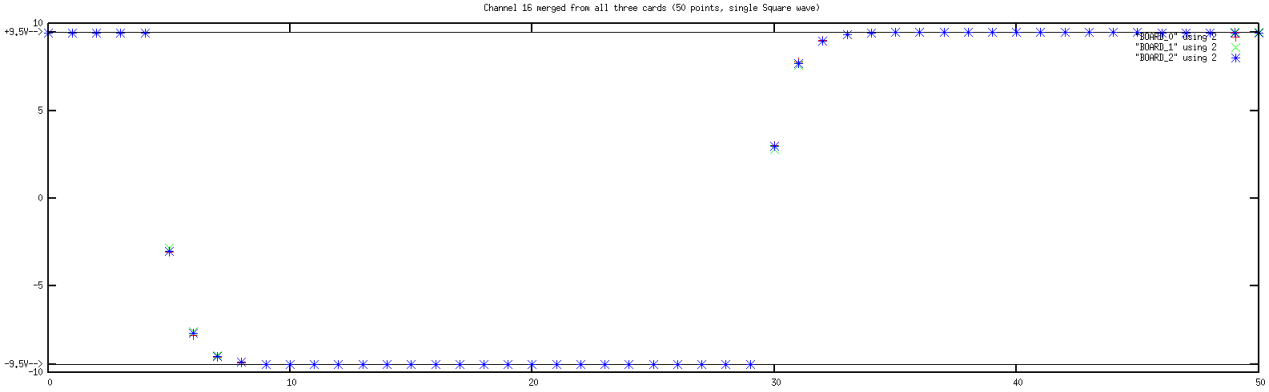
##
The following plot contains 50 sample points of channel 14 from all three cards. You should see a single Sine wave from each of the three cards, and these wave points should overlap if the clocks on all three cards are properly synchronized
##



##########

The following plot contains 50 sample points of channel 16 from all three cards. You should see a single Square wave from each of the three cards, and these wave points should overlap if the clocks on all three cards are properly synchronized

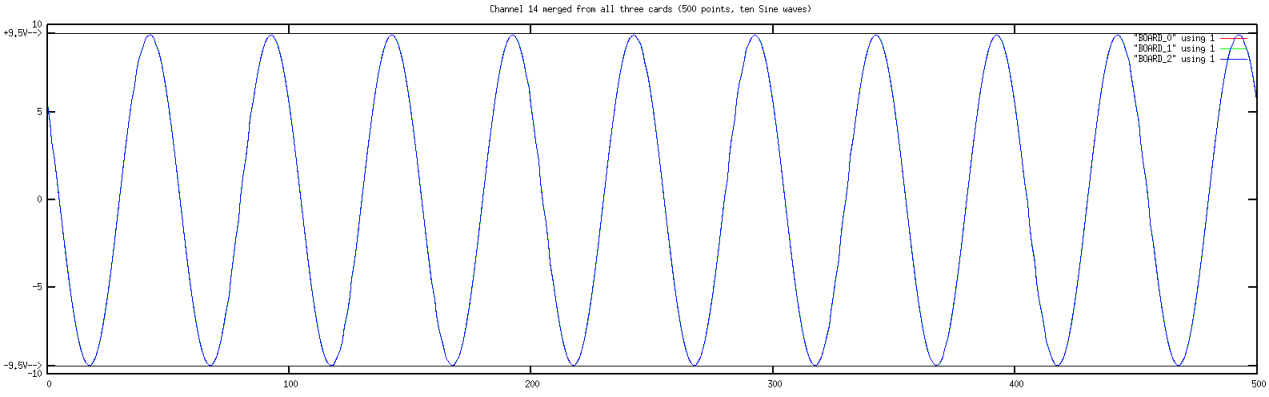
##########



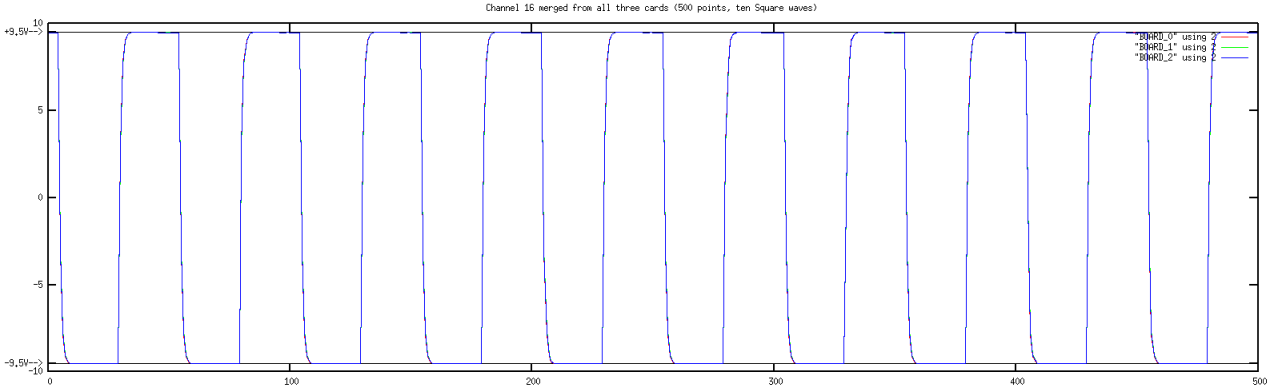
##########

The following plot contains 500 sample points of channel 14 from all three cards. You should see ten Sine waves from each of the three cards, and these wave should overlap if the clocks on all three cards are properly synchronized

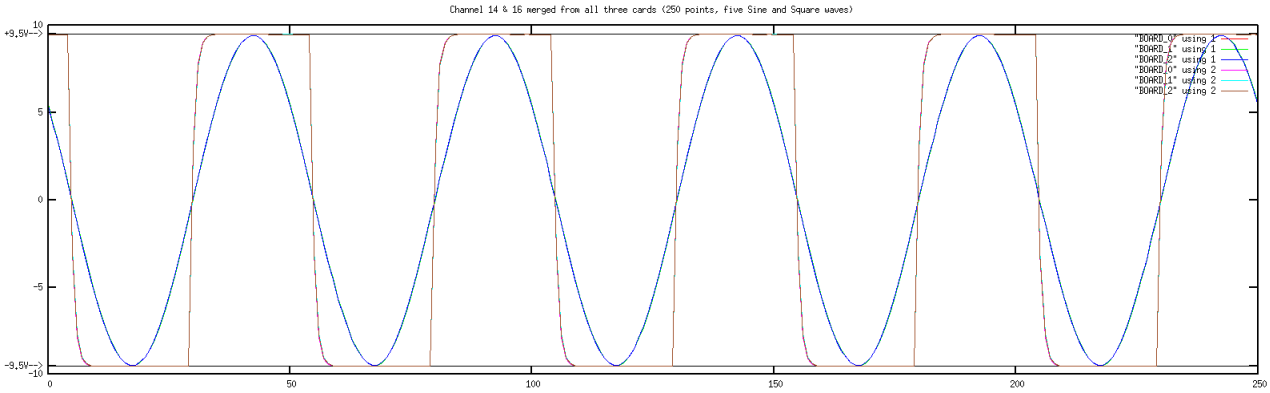
##########



##
The following plot contains 500 sample points of channel 16 from all three cards. You should see ten Square waves from each of the three cards, and these wave should overlap if the clocks on all three cards are properly synchronized
##



##
The following plot contains 250 sample points of channel 14 & 16 from all three cards. You should see five Sine and five Square waves from each of the three cards, and these wave should overlap if the clocks on all three cards are properly synchronized
##



9. Modular Scatter-Gather DMA (MsgDma)

In addition to the regular *basic* DMA engines, the new firmware for this board also supports a single Modular Scatter-Gather DMA engine *(MsgDma)*. Basically, this *MsgDma* engine operates similar to the *basic* DMA engines, but is considerably faster, however with some restrictions, due to the inherent nature of this engine.

This new *MsgDma* engine requires a *configuration/setup* stage to perform a one-time scatter-gather configuration followed by a repeated *firing* stage that performs the actual I/O. Since this is a two stage operation *(unlike that of basic DMA)*, the *MsgDma* configuration must not be modified by another application while it in use, otherwise it could lead to unpredictable and possibly damaging outcome. To avoid the possibility of another application accessing the *MsgDma* engine while in use, two API calls have been provided to *Seize* and *Release* this operation. Following are a list of the available *MsgDma* API calls available to the user:

- ccrtAICC_MsgDma_Configure_ADC_Fifo()
- ccrtAICC_MsgDma_Configure_Descriptor()
- ccrtAICC_MsgDma_Configure_Single()

- ccrtAICC_MsgDma_Fire()
- ccrtAICC_MsgDma_Fire_ADC_Fifo()
- ccrtAICC_MsgDma_Fire_Single()
- ccrtAICC_MsgDma_Free_Descriptor()
- ccrtAICC_MsgDma_Get_Descriptor()
- ccrtAICC_MsgDma_Get_Dispatcher_CSR()
- ccrtAICC_MsgDma_Get_Prefetcher_CSR()
- ccrtAICC_MsgDma_Release()
- ccrtAICC_MsgDma_Seize()
- ccrtAICC_MsgDma_Setup()

The above API calls are grouped according to the *MsgDma* operation needing to be performed:

1. Performing memory to board or board to memory transfer for a *single* location and size
    - ccrtAICC_MsgDma_Seize()
    - ccrtAICC_MsgDma_Configure_Single()
    - ccrtAICC_MsgDma_Fire_Single() - *this step is repeated multiple times for each I/O*
    - ccrtAICC_MsgDma_Release()
2. Performing memory to board or board to memory transfer for *multiple* locations and sizes
    - ccrtAICC_MsgDma_Seize()
    - ccrtAICC_MsgDma_Free_Descriptor()
    - ccrtAICC_MsgDma_Configure_Descriptor() - *this step is repeated multiple times for each location and size*
    - ccrtAICC_MsgDma_Setup() - *this step is performed once to complete the above configuration*
    - ccrtAICC_MsgDma_Fire() - *this step is repeated multiple times for each I/O*
    - ccrtAICC_MsgDma_Release()
3. Performing ADC Fifo reads from the board for a specified number of samples. The ADC Fifo needs to hold at least the specified number of samples before issuing the *ccrtAICC_MsgDma_Fire_ADC_Fifo()* call.
    - ccrtAICC_MsgDma_Seize()
    - ccrtAICC_MsgDma_Configure_ADC_Fifo()
    - ccrtAICC_MsgDma_Fire_ADC_Fifo() - *this step is repeated multiple times for each I/O*
    - ccrtAICC_MsgDma_Release()

    The number of samples to be extracted from the ADC Fifo is specified in the *ccrtAICC_MsgDma_Configure_ADC_Fifo()* call. If the user desires to collect a different number of samples from the FIFO, they need to re-issue this configuration API call with the new sample count. This call has a fair amount of overhead due to reconfiguring the *MsgDma* engine with the new size. It is therefore highly recommended that the user select a sample size that should not be changing during the sample collection otherwise any performance improvements achieved by this *MsgDma* operation will be lost.

For more information on these API calls, refer to the *ccrtaicc_software_interface* document. Additionally, you can get more information on programming the card by referring to the various library tests supplied with the driver.

This page intentionally left blank