Release Notes
CCURAOCC_WAVE (WC-DA3218-WAVE)

API	Ccuraocc_wave (WC-DA3218-WAVE)	Rev 6.5
OS	RedHawk	Rev 6.5
Vendor	Concurrent Computer Corporation	
Hardware	PCIe 32-Channel Digital to Analog Output Converter Card (CP-DA3218)	
Date	May 27th, 2014	

concurrent
REAL-TIME

This page intentionally left blank

Table of Contents

This page intentionally left blank

1. Introduction

This document assists the user in installing the Concurrent Computer Corporation's (CCUR's) ***ccuraocc-wave*** library and test programs on a RedHawk OS for use with the CCUR PCIe Digital to Analog Output Converter Card CP-DA3218. This library relies on the ***ccuraocc driver*** being installed. For further information on the driver and the board, refer to the WC-DA3218 Digital to Analog Output Card driver installation documentation by Concurrent Computer Corporation.

2. Wave Generation Description

The AOCC is a 32-channel 18-bit digital to analog converter card with a PCI express interface. It is implemented using Linear Technology LTC2758 dual channel DAC's. The PCI interface utilizes a PLX Technology PEX-8311AA PCI-express-to-local bus bridge. There is a Lattice ECP2M FPGA for control of board functions including registers and storage. An adjustable main clock source is generated by a low jitter PLL. The external synchronizing interface consists of LVDS signaling connected via RJ-12 (6-pin phone) style cabling.

This Concurrent Computer Corporation *ccuraocc_wave* (***WC-DA3218-WAVE)*** package consists of a library and test program that interfaces with the ***ccuraocc*** driver to provide an easy, yet powerful mechanism for users to generate continuous and unique waveforms on any specified channels for the CP-DA3218 cards. These waveforms are supplied by the user in the form of an ASCII file that conforms to a specific format. Various user interface calls have been provided in the API located in the ***libaocc_stream.a*** library.

Once the waves are running, users can perform seamless operations on them, such as varying the frequency and phase of the data samples once the internal buffered samples have been exhausted. Capability is provided to affect these changes on a per-channel basis with a single API call. Other running channels that are not part of the change are not affected in any way. Users can similarly switch to different waves on running channels in a seamless operation with the appropriate API calls. At any time, during wave generation, users have commands available to stop, pause and resume waves on the board. Additionally, applications using this API, can schedule themselves at various priorities on various targeted CPU during the opening of the wave stream.

Capability for internal or external clocking can be accomplished through this API to connect multiple cards and synchronize commencement of waves on all the cards.

There is also no limit to linking new waveform scenarios with running waveforms without affecting the operation of the running waveforms on individual channels. A single API call can then be used to terminate running waveform scenarios and start the next linked scenario.

Users can also provide a calibration file per board; that is used by the API to perform offset/gain calibration for each channel.

Finally, API calls are also provided to enable debugging and printing useful wave streaming information.

3. Requirements

The following is the system requirements and must be installed on the system.

- CCURAOCC PCI express 32-channel, 18-bit Single-Ended or Differential, Digital to Analog Output Converter Card.
- Selected versions of RedHawk Revision 6.5.x only on either i386, i686 or x86_64 platforms. Actual supported versions depend on the individual wave package.
- CCURAOCC driver with its library installed.

4. Installation and Removal

4.1. Hardware Installation

Refer to the *CCURAOCC (WC-DA3218)* driver installation for necessary hardware installation.

4.2. Software Installation

Before installing the ***ccuraocc_wave*** package, you will need to install the ***ccuraocc*** driver and libraries; otherwise, installation will fail.

The ***ccuraocc_wave*** package is supplied in an ***rpm*** format on a CD-ROM.

To install the ***ccuraocc_wave*** package, load the CD-ROM installation media and issue the following commands as the **root** user. The system should auto-mount the CD-ROM to a mount point in the **/media** directory based on the CD-ROM's volume label – in this case **/media/ccuraocc_wave**. Then enter the following commands from a shell window:

```
== as root ==
# cd /media/ccuraocc_wave
# rpm -ivh ccuraocc_wave_RedHawk_driver_6.5.*.rpm
# cd /
# eject
```

On successful installation the source tree for the ***ccuraocc_wave*** package, including the API libraries, and test programs is extracted into the */usr/local/CCUR/drivers/ccuraocc_wave* directory by the rpm installation process, which will then compile and install the various software components.

4.3. Software Removal

The ***ccuraocc_wave*** package can be uninstalled and removed. Once removed, the only way to recover the driver is to re-install the ***rpm*** from the installation CDROM:

If any changes have been made to the package installed in ***/usr/local/CCUR/drivers/ccuraocc_wave*** directory, they need to be backed up prior to invoking the removal; otherwise, all changes will be lost.

```
=== as root ===
# rpm -e ccuraocc_wave     (uninstall and delete package)
```

If, for any reason, the user wishes to uninstall the package but not remove it, they can perform the following:

```
=== as root ===
# cd /usr/local/CCUR/drivers/ccuraocc_wave
# make uninstall        (uninstall the package)
```

In this way, the user can simply issue the ***'make install'*** in the ***/usr/local/CCUR/drivers/ccuraocc_wave*** directory at a later date to re-install the package.

5. Associate Device Names to Cards

By default, the ***ccuraocc*** driver creates two device names for each board found in the system. The name of the devices are */dev/ccuraocc<bno>* and */dev/ccuraocc_wave<bno>* where <bno> corresponds the card number found in the system. This *aocc_stream* library only uses the */dev/ccuraocc_wave* device name to communicate with the driver.

Refer to the ***ccuraocc*** driver installation document for information of associating device names to cards.

6. Building and Running the Tests

Build and run the wave test programs *(for further information, refer to Section 10):*

```
   === as user or root ===
# cd /usr/local/CCUR/drivers/ccuraocc_wave/test
# make
# ./aocc_stream            (run aocc_stream  wave generator using AOCC_STREAM api)
```

7. Re-building the Library and Tests

If for any reason the user needs to manually rebuild and load an *installed **rpm*** package, they can go to the installed directory and perform the necessary build.

To build the library and tests:

```
=== as root ===
   # cd /usr/local/CCUR/drivers/ccuraocc_wave
   # make clobber      (perform cleanup)
   # make              (make package and build the library and tests)
```

After the package is built, you will need to install it.

```
=== as root ===
# cd /usr/local/CCUR/drivers/ccuraocc_wave
# make install      (install the library and man page)
```

8. AOCC_Stream API

This API provides an easy yet powerful mechanism for users to generate continuous and unique waveforms on any specified channels for CP-DA3218 cards. This API is located in the ***libaocc_stream.a*** library. These wave forms are supplied by the user in the form of an ASCII file that conforms to a specific format. A test program aocc_stream.c located in the test directory is a useful starting point to become familiar with the API.

8.1. User Interface Calls

```
AOCC_ActionChangeChanFreq()
AOCC_ActionChangeChanPhase()
AOCC_ActionChangeChanWave()
AOCC_ActionLinkWave()
AOCC_ActionRemoveWaves()
AOCC_ActionSelectNextWave()
AOCC_CloseAOStream()
AOCC_CommandAOStream()
AOCC_DebugAOStream()
AOCC_ErrorAOStream()
AOCC_InfoAOStream()
AOCC_OpenAOStream()
AOCC_PrintFifoStats()
AOCC_PrintInfoAOStream()
AOCC_PrintTimingAOStream()
```

8.2. Error Codes

```
typedef enum {
    AOCC_ERROR_NONE                             = (0),

    AOCC_ERROR_BOARD_ALREADY_OPENED             =(-10),
    AOCC_ERROR_BOARD_OPEN_FAILED                =(-11),
    AOCC_ERROR_CHANNEL_NOT_ACTIVE               =(-12),
    AOCC_ERROR_COMMAND_NOT_ALLOWED              =(-13),
    AOCC_ERROR_DUPLICATE_CHANNELS               =(-14),
    AOCC_ERROR_GET_DRIVER_INFO_FAILED           =(-15),
    AOCC_ERROR_GET_BOARD_INFO_FAILED            =(-16),
    AOCC_ERROR_MMAP_FAILED                      =(-17),
    AOCC_ERROR_SAMPLE_RATE_PROGRAM_FAILED       =(-18),
    AOCC_ERROR_THREAD_INIT_FAILED               =(-19),
    AOCC_ERROR_UNSUPPORTED_CHANNEL              =(-20),
    AOCC_ERROR_WRITE_COUNT_MISMATCH             =(-21),

    AOCC_ERROR_INVALID_ARGUMENT                 =(-100),
    AOCC_ERROR_INVALID_BOARD_FREQUENCY          =(-101),
    AOCC_ERROR_INVALID_BOARD_HANDLE             =(-102),
    AOCC_ERROR_INVALID_BOARD_NUMBER             =(-103),
    AOCC_ERROR_INVALID_CALIBRATE_ARGUMENT       =(-104),
    AOCC_ERROR_INVALID_CALIBRATION_FILE         =(-105),
    AOCC_ERROR_INVALID_CHANNEL_FORMAT           =(-106),
    AOCC_ERROR_INVALID_COMMAND                  =(-107),
    AOCC_ERROR_INVALID_DEBUG_OPTION             =(-108),
    AOCC_ERROR_INVALID_INPUT_CLOCK_SELECTION    =(-109),
    AOCC_ERROR_INVALID_KSAMPLES_PER_WRITE       =(-110),
    AOCC_ERROR_INVALID_NUM_OF_MUX_BUFS          =(-111),
    AOCC_ERROR_INVALID_NUM_OF_WRITE_BUFS        =(-112),
    AOCC_ERROR_INVALID_OUTPUT_CLOCK_SELECTION   =(-113),
    AOCC_ERROR_INVALID_PHASE                    =(-114),
    AOCC_ERROR_INVALID_SAMPLE_FILE              =(-115),
    AOCC_ERROR_INVALID_SCHEDULING_POLICY        =(-116),
    AOCC_ERROR_INVALID_THREAD_PRIORITY          =(-117),
```

```
    AOCC_ERROR_INVALID_VOLTAGE_RANGE            =(-118),

    AOCC_ERROR_NO_ACTIVE_WAVES                  =(-200),
    AOCC_ERROR_NO_CHANNELS_SPECIFIED            =(-201),
    AOCC_ERROR_NO_FILELIST                      =(-202),
    AOCC_ERROR_NO_MEMORY_AVAILABLE              =(-203),

    AOCC_ERROR_FIFO_UNDERFLOW                   =(-300),
    AOCC_ERROR_FIFO_OVERFLOW                    =(-301),

    AOCC_ERROR_TIMEOUT                          =(-999999),

    AOCC_ERROR_END_OF_TABLE                     =(0xbaad),
} _error_number_t;
```

8.3. AOCC_ActionChangeChanFreq()

Function: int AOCC_ActionChangeChanFreq(void *handle, AOCC_aos_file_list_t *flist, uint immediate)

Return: good return - AOCC_ERROR_NONE
Error - one of various errors

Description: Change the *frequency* of a set of active channels that have been previously running. User specifies a list of *channel masks* and only *frequency,* via the *flist* structure. All other fields in this structure are ignored. If the *immediate* value is set to '0', change of frequency will occur seamlessly once the internal buffered samples and samples in the hardware FIFO have been exhausted. If the user enters a negative frequency, no frequency adjustment is applied to the basic wave. The wave is run at its natural boards sampling rate, i.e. board sampling frequency divided by the number of samples in the wave. If the frequency is zero, a steady voltage results on the output.

Since this seamless operation requires changes to be "appended" to the already processed samples for all the active channels, the actual *frequency* change on the output will be delayed by the samples that have been buffered for the active channels. If for some reason, the user wishes to shorten the duration when the change is to occur, they may consider setting the *immediate* value to '1'. In this case, the call flushes the entire queued samples (for all active channels) in both the internal buffers and the hardware FIFO and re-primes the FIFO with the new data prior to commencing generation of the change. Seamless operation cannot be achieved in this case. The result is that waves on ALL the active channels will abruptly stop for a short duration before they all resume once again from their initial starting point (first sample in the individual waveform files). Additionally, those waves that requested a *frequency* change will resume at the new frequency. All waves on all the active channels will commence concurrently at the same instant.

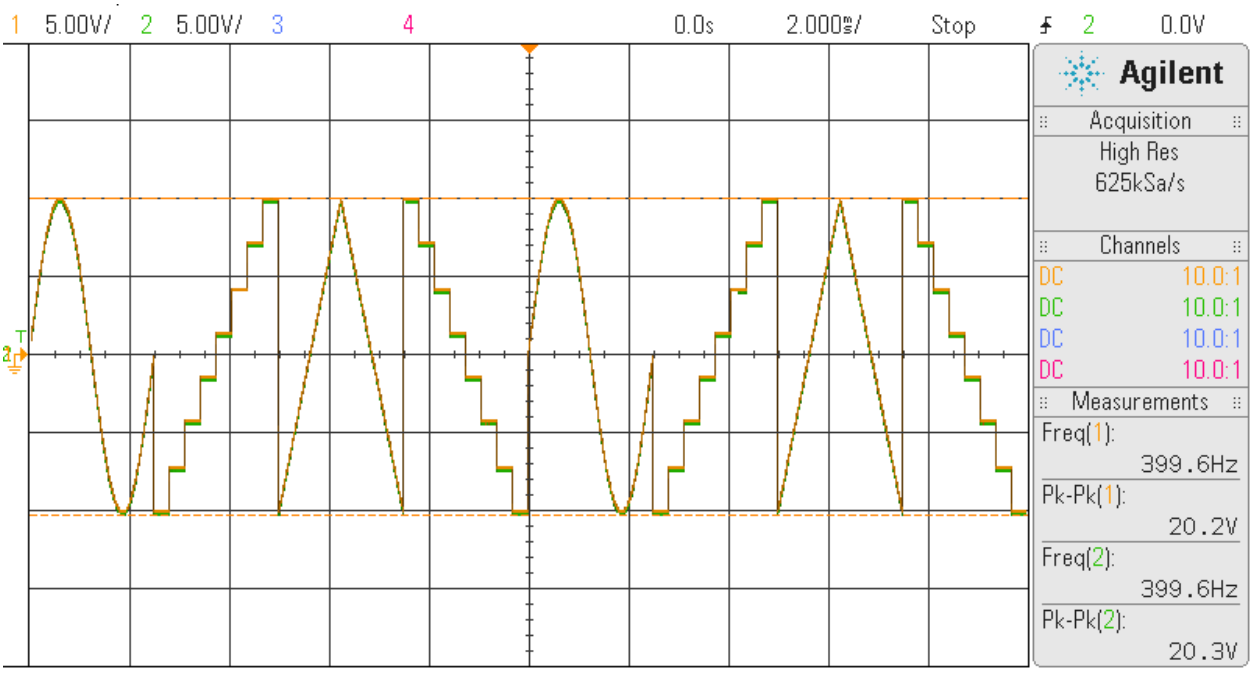
Please note that for immediate values greater than '0', we are no longer going to perform a seamless operation. The larger the immediate value, the longer it will take before the wave change takes place. The reason is that the immediate value is a count of DMA writes to fill the hardware FIFO. Normally, a count of '1' should suffice, however, depending on the number of active channels and other activity in the system, it is possible that the wave could under-run after only filling a partial FIFO.

```
/* file list */
typedef struct _AOCC_aos_file_list_t {
    uint    ChanMask;               /* channel mask */
    double  frequency;              /* channel frequency */
    char    WaveFile[PATH_MAX];     /* channel wave file */
    double  amplitude;              /* channel amplitude (really gain) */
    double  bias;                   /* channel bias (really offset) */
    double  phase;                  /* channel initial phase (-360.0 -
                                       +360.0)*/
} AOCC_aos_file_list_t;
```

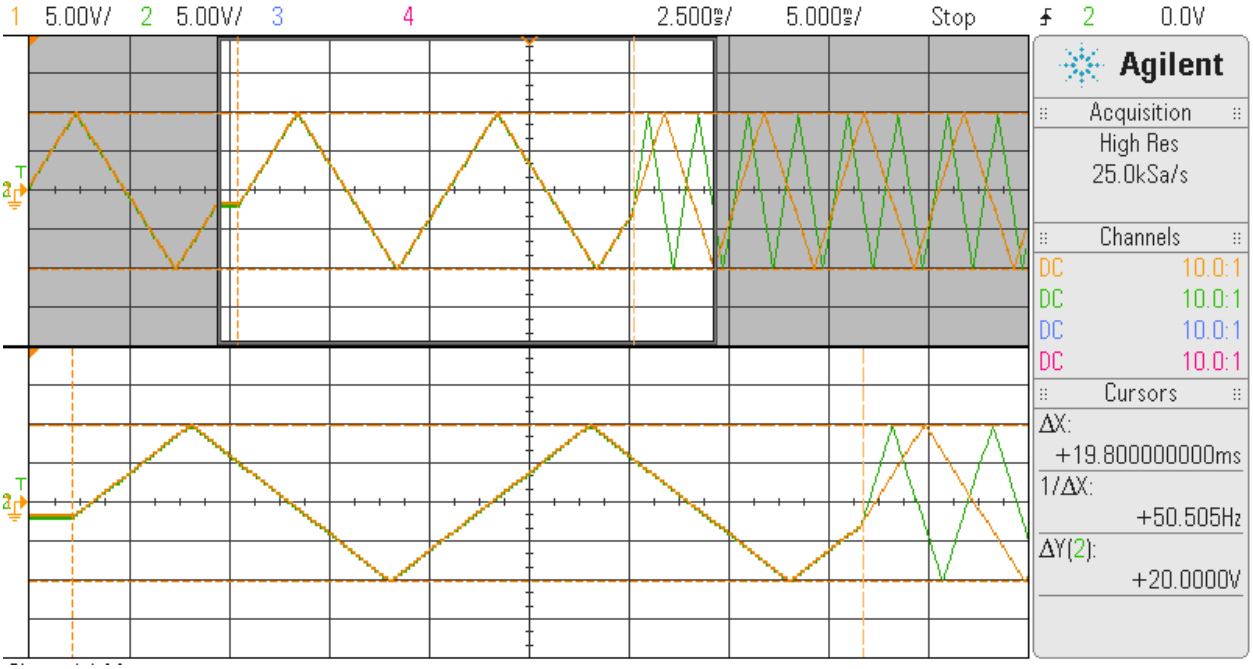
Example:

```
AOCC_aos_file_list_t flist[CCURAOCC_MAX_CHANNELS+1];
memset(flist, 0,sizeof(flist);
flist[0].ChanMask  = 0x0101;    /* select channels 0 and 8 */
flist[0].frequency = 100.45;    /* run earlier defined wave at 100.45Hz */
flist[1].ChanMask  = 0x00c2;    /* select channels 1 , 6 and 7 */
flist[1].frequency = 1295.0;    /* run earlier defined wave at 1295Hz */

flist[2].ChanMask  = 0;         /* end of list */

if(AOCC_ActionChangeChanFreq(handle, flist, 0)) /* seamless */
    exit(1);
```

```
if(AOCC_ActionChangeChanFreq(handle, flist, 1)) /* immediate=1 */
    exit(1);
```

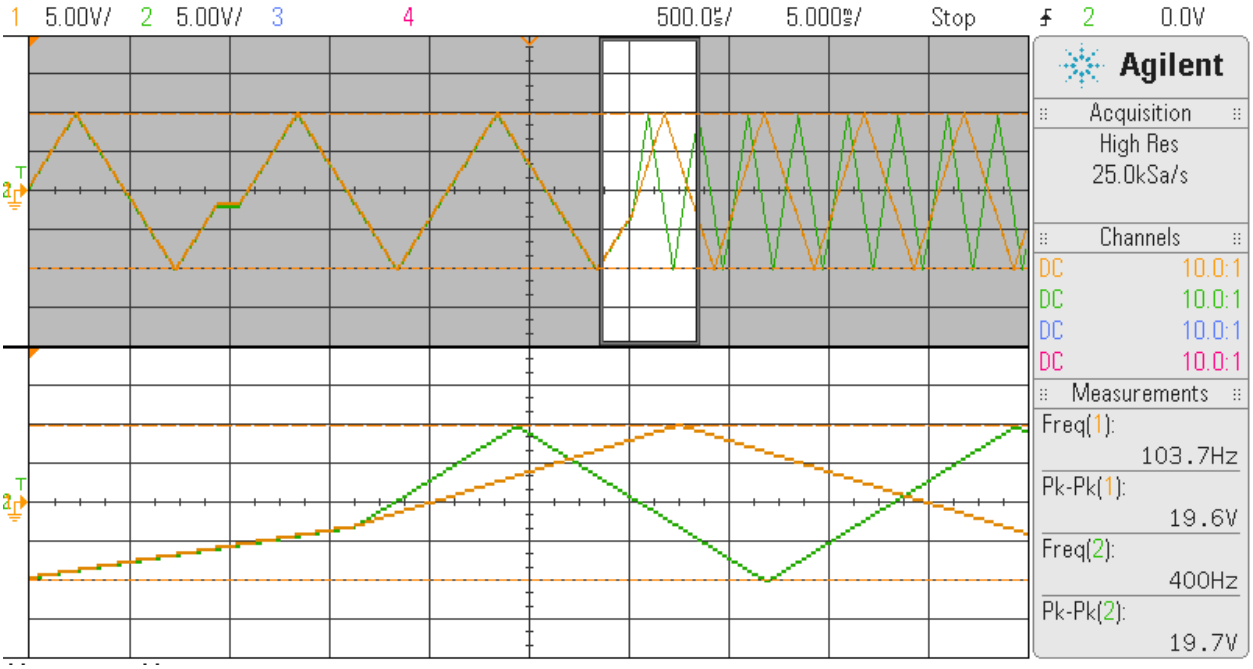


Two waves (Ch0 - yellow & Ch31 - green) displayed above using the wave generation. They were both started at the same time and running at the same frequency of 400 Hz. They are placed one on top of the other to show that they are in phase. During this run, all 32 channels were running the same wave at 400KSPS.

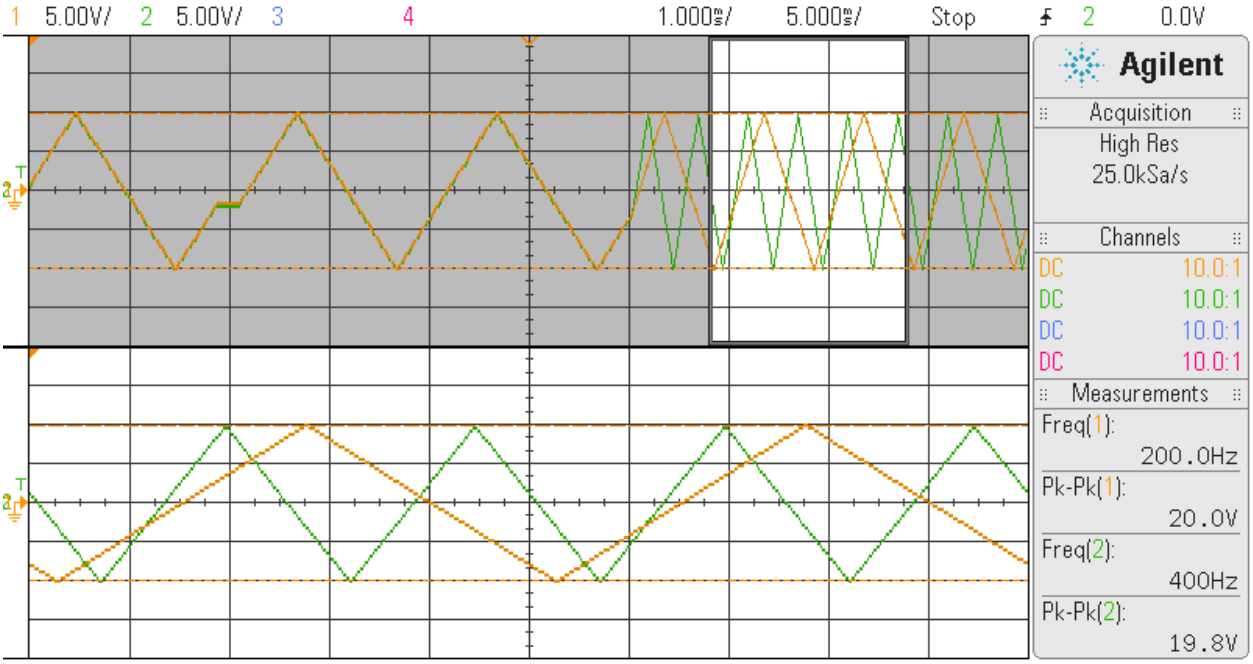


This time, we are demonstrating a seamless transition of two channels. Here, both Ch0 and Ch31 are generating a triangle wave at 100Hz. The frequency is then changed *seamlessly* where Ch0 is changed to 200Hz (yellow) and Ch31(green) is changed to 400Hz. Also, a temporary “hook” was placed in the test program to stop/start the system clock just prior to calling the change *frequency* API. The reason is to “spot” in the trace where the clock stops/starts (shown as a flat line). Measuring the distance from the end of the flat line (when the clock is restarted) to the point where the two waves change frequency shows approximately 20ms. This is because at the time the change frequency was issued, there were several

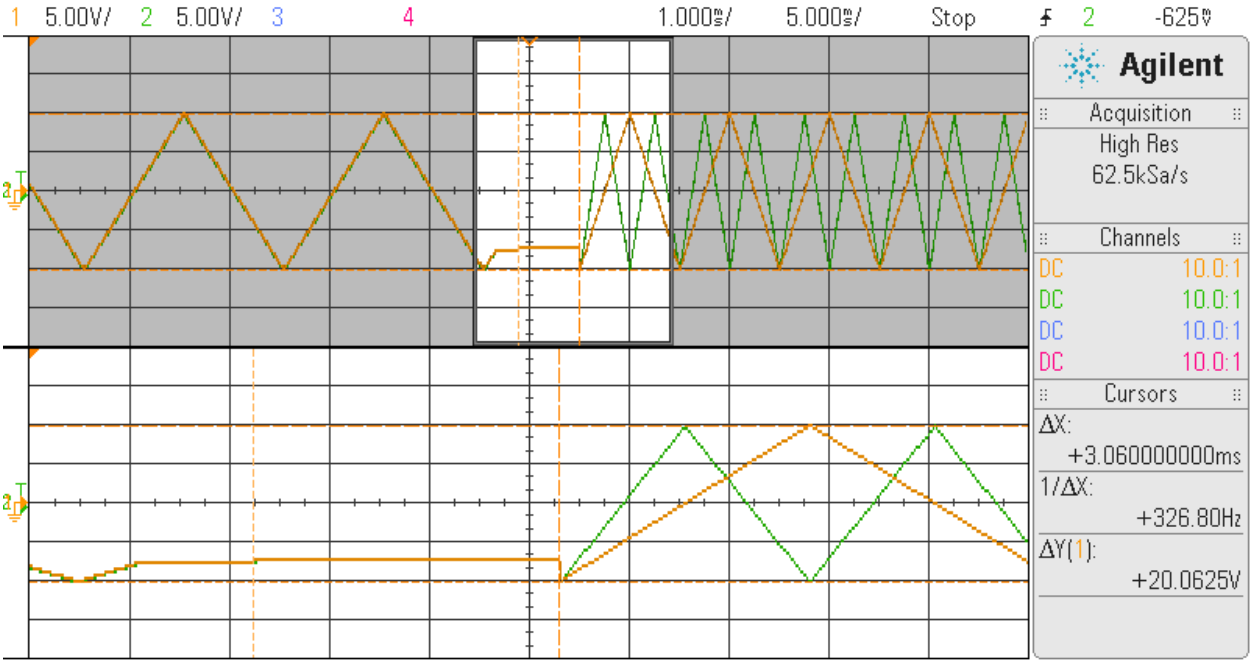
samples, both in the hardware FIFO and the internal wave generation buffers that were queued for output prior to the frequency change. There are tunable parameters for the DMA size, the number of DMA buffers and the number of Multiplexor buffers that are under the control of the user during the opening of the stream that can be changed to shorten the latency. Currently, the default settings of DMA size (16KSamples), number of DMA buffers (4) and number of Multiplexor buffers (3) are used during this run. The full 128K FIFO is used in this run.



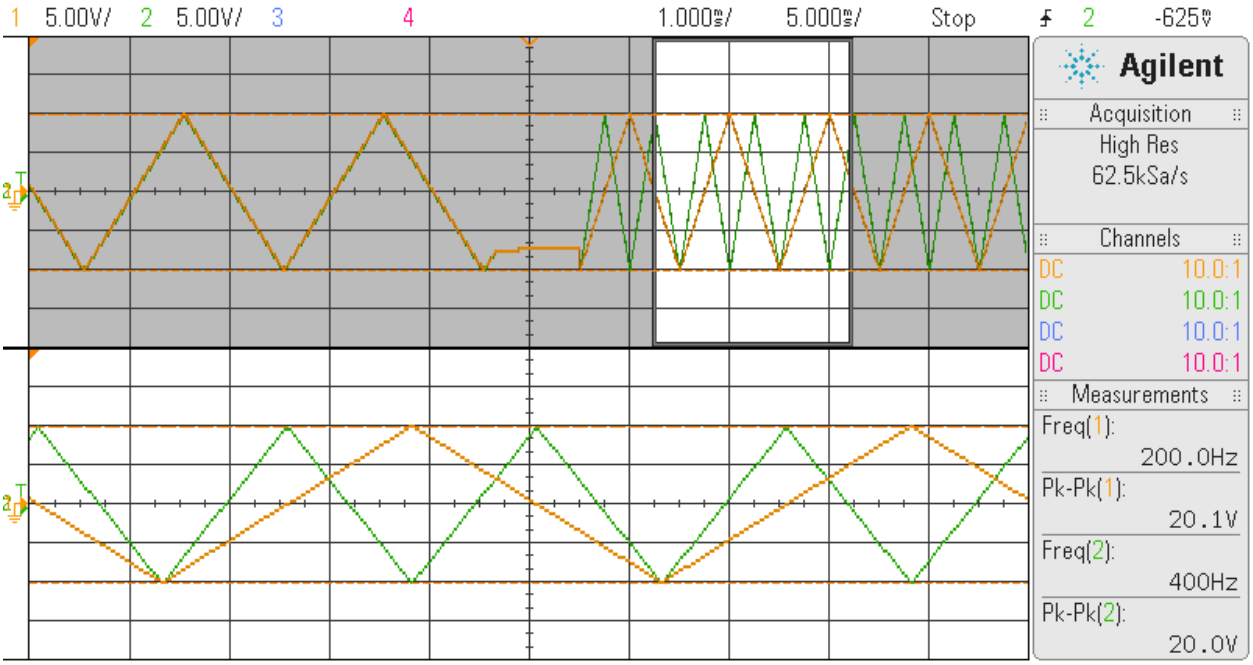
Same wave run as above, however, zooming into the region where the wave change is occurring from 100Hz to 200Hz (Ch0-yellow) and 400Hz (Ch31-green).



Same wave run as above, however, showing the changed waves. As seen on the right under Measurement, the changed frequency of the two waves is shown by the scope.



The above displays two channels (Ch0-yellow and Ch32-green) generating a triangle wave at 100Hz. Now, the user performs the frequency change request, however, this time setting the *immediate* value to '1'. Once again, the test was modified to stop/start the clock and hence the timing from the re-starting of the clock to the restarting of the waves is approximately 3ms. This time basically represents the partial filling of the hardware FIFO. Once again, in this run, all 32 channels are active and running at 400KSPS. During the time the FIFO is filling, ALL 32 channels will stop generating their corresponding waves. Once the wave generator starts outputting waves, all 32 channels will commence at the initial offset (offset 0) within their respective wave form files. The full 128K FIFO is used in this run.



Same wave run as above, however, showing the changed waves. As seen on the right under Measurement, the changed frequency of the two waves is shown by the scope.

8.4. AOCC_ActionChangeChanPhase()

Function: int AOCC_ActionChangeChanPhase(void *handle, AOCC_aos_file_list_t *flist, uint immediate)

Return: good return - AOCC_ERROR_NONE
Error - one of various errors

Description: Change the *phase* of a set of active channels that have been previously running. User specifies a list of *channel masks* and only *phase,* via the *flist* structure. All other fields in this structure are ignored. If the *immediate* value is set to '0', change of *phase* will occur seamlessly once the internal buffered samples and samples in the hardware FIFO have been exhausted. Users can specify any phase shift between -360.0 and +360.0. A phase of 0 or +/-360.0 will result in no phase shift..

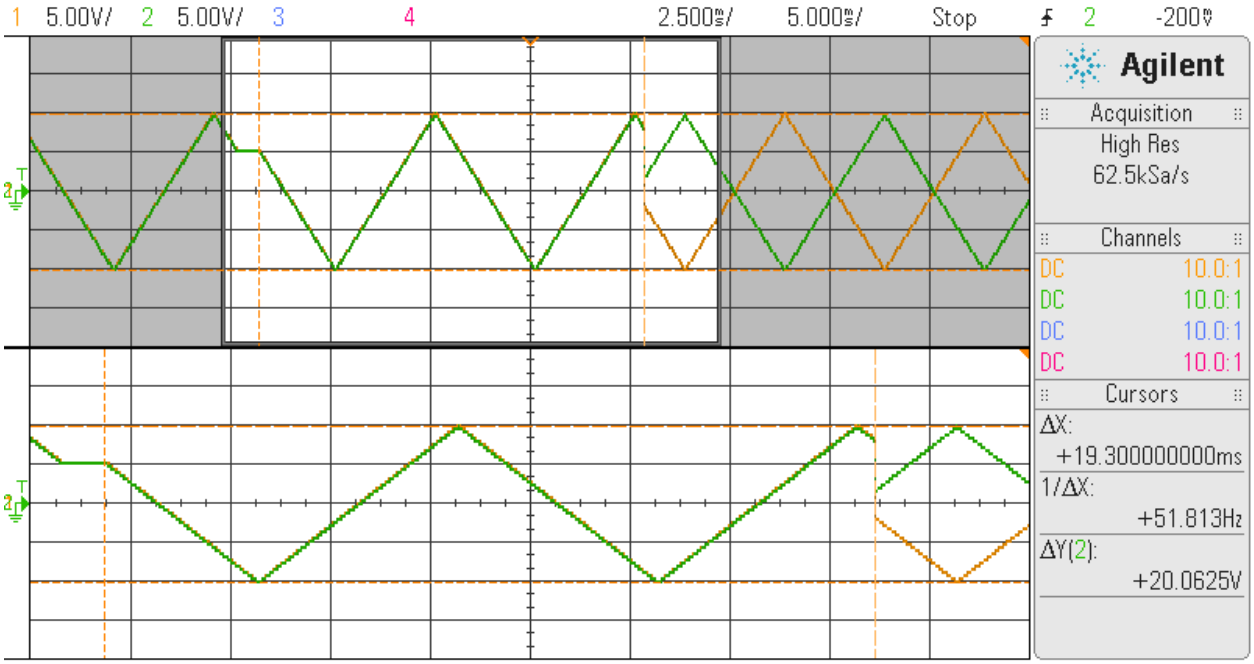
Since this seamless operation requires changes to be "appended" to the already processed samples for all the active channels, the actual *phase* change on the output will be delayed by the samples that have been buffered for the active channels. If for some reason, the user wishes to shorten the duration when the change is to occur, they may consider setting the *immediate* value to '1'. In this case, the call flushes the entire queued samples (for all active channels) in both the internal buffers and the hardware FIFO and re-primes the FIFO with the new data prior to commencing generation of the change. Seamless operation cannot be achieved in this case. The result is that waves on ALL the active channels will abruptly stop for a short duration before they all resume once again from their initial starting point (first sample in the individual waveform files). Additionally, those waves that requested a *phase* change will resume at the new phase. All waves on all the active channels will commence concurrently at the same instant.

Please note that for immediate values greater than '0', we are no longer going to perform a seamless operation. The larger the immediate value, the longer it will take before the wave change takes place. The reason is that the immediate value is a count of DMA writes to fill the hardware FIFO. Normally, a count of '1' should suffice, however, depending on the number of active channels and other activity in the system, it is possible that the wave could under-run after only filling a partial FIFO.

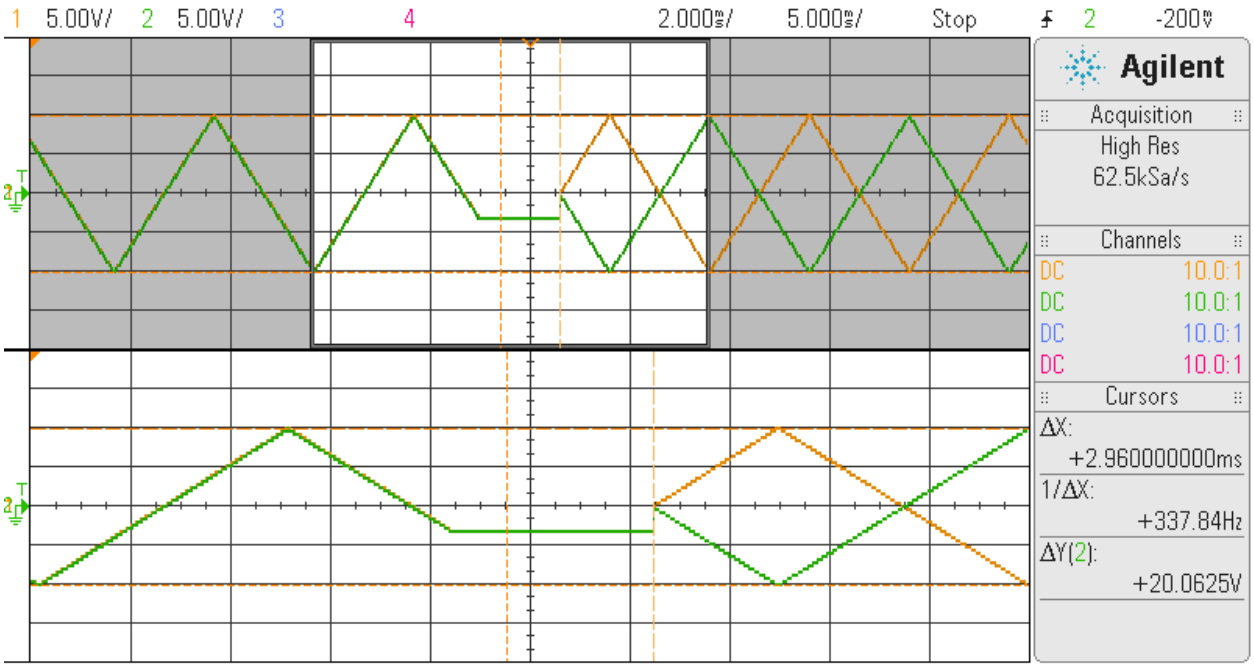
```
/* file list */
typedef struct _AOCC_aos_file_list_t {
    uint    ChanMask;               /* channel mask */
    double  frequency;              /* channel frequency */
    char    WaveFile[PATH_MAX];     /* channel wave file */
    double  amplitude;              /* channel amplitude (really gain) */
    double  bias;                   /* channel bias (really offset) */
    double  phase;                  /* channel initial phase (-360.0 -
                                       +360.0)*/
} AOCC_aos_file_list_t;
```

Example:

```
AOCC_aos_file_list_t flist[CCURAOCC_MAX_CHANNELS+1];
memset(flist, 0,sizeof(flist);
flist[0].ChanMask  = 0x0101;    /* select channels 0 and 8 */
flist[0].phase     = -90.0;     /* change phase of running wave */
flist[1].ChanMask  = 0x00c2;    /* select channels 1 , 6 and 7 */
flist[1].phase     = +32.5;     /* change phase of running wave */
flist[2].ChanMask  = 0;         /* end of list */

if(AOCC_ActionChangeChanPhase(handle, flist, 0)) /* seamless */
     exit(1);
if(AOCC_ActionChangeChanPhase(handle, flist, 1)) /* immediate=1 */
     exit(1);
```



This time, we are demonstrating a seamless transition of two channels. Here, both Ch0 and Ch31 are generating a triangle wave at 100Hz. The phase is then changed *seamlessly* where Ch0 is changed to +90 degrees (yellow) and Ch31 is changed to -90 degrees. Also, a temporary "hook" was placed in the test program to stop/start the system clock just prior to calling the *change phase* call. The reason is to see "spot" in the trace where the clock stops/starts (shown as a flat line). Measuring the distance from the end of the flat line (when the clock is restarted) to the point where the two waves change phase shows approximately 19ms. This is because at the time the change phase was issued, there were several samples, both in the hardware FIFO and the internal wave generation utility that were to be sent prior to the phase change. There are tunable parameters for the DMA size, the number of DMA buffers and the number of Multiplexor buffers that are under the control of the user during the opening of the stream that can be changed to shorten the switching time. Currently, the default settings of DMA size (16KSamples), number of DMA buffers (4) and number of Multiplexor buffers (3) are used during this run. The full 128K FIFO is used in this run.



The above displays two channels (Ch0-yellow and Ch32-green) generating a triangle wave at 100Hz. Now, the user performs the phase change request, however, this time setting the *immediate* value to '1'. Once again, the test was modified to stop/start the clock and hence the timing from the re-starting of the clock to the restarting of the waves is approximately 3ms. This time basically represents the partial filling of the hardware FIFO. Once again, in this run, all 32 channels are active and running at 400KSPS. During the time the FIFO is filling, ALL 32 channels will stop generating their corresponding waves. Once the wave generator starts outputting waves, all 32 channels will commence at the initial offset (offset 0) within their respective wave form files.

8.5. AOCC_ActionChangeChanWave()

Function: int AOCC_ActionChangeChanPhase(void *handle, AOCC_aos_file_list_t *flist, uint immediate)

Return: good return - AOCC_ERROR_NONE
Error - one of various errors

Description: This API changes the wave, frequency, phase, amplitude and bias of a set of active channels that have been previously running. User specifies a list of channel masks, wave files, frequency, phase, amplitude and bias via the *flist* structure. If the user enters a negative frequency, no frequency adjustment is applied to the basic wave. The wave runs at the boards sampling rate, i.e. board sampling frequency divided by the number of samples in the wave. A phase of 0.0, -360.0 or +360.0 will not result in any phase shift. An amplitude of 1 or a bias of 0 will not affect the original wave. Change of wave, frequency, phase, amplitude and bias will occur seamlessly once the internal buffered samples have been exhausted. If user has specified calibration for the channel, the amplitude will be multiplied by any calibration gain and the bias will be added to any calibration offset. The user will need to make sure that the resulting waves range does not exceed the maximum voltage range for the board, otherwise clipping will occur and warning messages will be generated.

Since *seamless* operation requires changes to be "appended" to the already processed samples for all the active channels, the actual *wave change* on the output will be delayed by the samples that have been buffered for the active channels. If *seamless* operation is not of paramount importance, the user can shorten the duration when the change is to occur by setting the *immediate* value to '1'. In this case, the call flushes the entire queued samples (for all active channels) in both the internal buffers and the hardware FIFO and re-primes the FIFO with the new data prior to commencing generation of the change. The result is that waves on ALL the active channels will abruptly stop for a short duration before they all resume once again from their initial starting point (first sample in the individual waveform files). Additionally, those waves that requested a *wave* change will resume with the new wave. All waves on all the active channels will commence concurrently at the same instant.

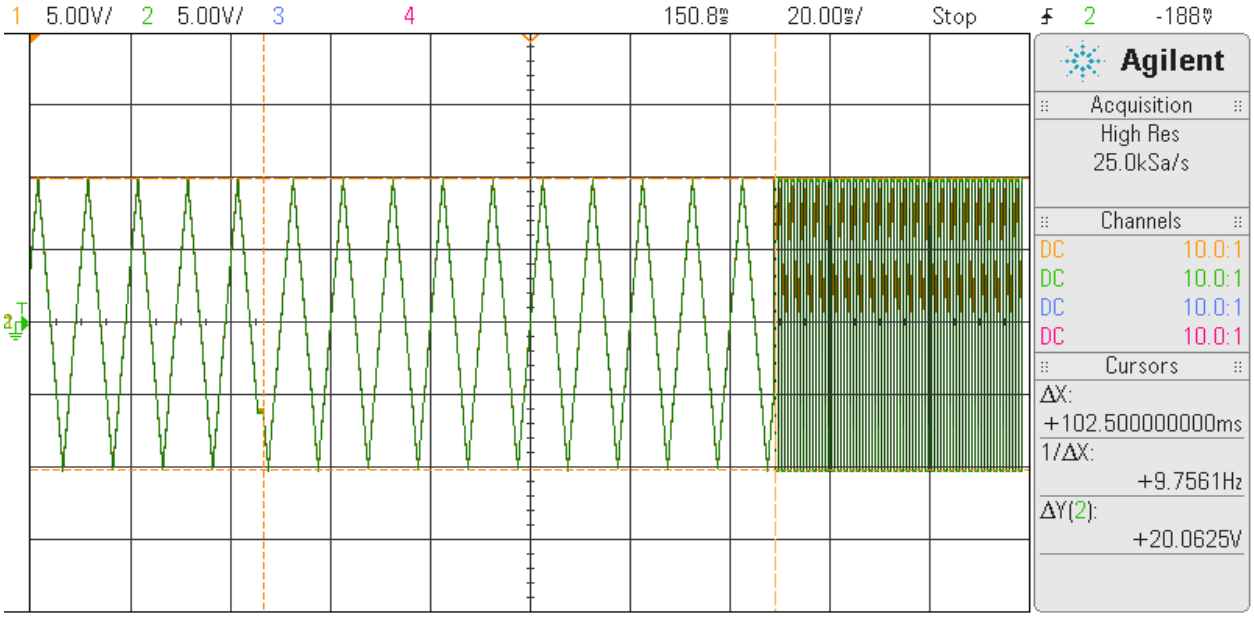
Please note that for immediate values greater than '0', we are no longer going to perform a seamless operation. The larger the immediate value, the longer it will take before the wave change takes place. The reason is that the immediate value is a count of DMA writes to fill the hardware FIFO. Normally, a count of '1' should suffice, however, depending on the number of active channels and other activity in the system, it is possible that the wave could under-run after only filling a partial FIFO.

```
/* file list */
typedef struct _AOCC_aos_file_list_t {
    uint    ChanMask;               /* channel mask */
    double  frequency;              /* channel frequency */
    char    WaveFile[PATH_MAX];     /* channel wave file */
    double  amplitude;              /* channel amplitude (really gain) */
    double  bias;                   /* channel bias (really offset) */
    double  phase;                  /* channel initial phase (-360.0 -
                                       +360.0)*/
} AOCC_aos_file_list_t;
```

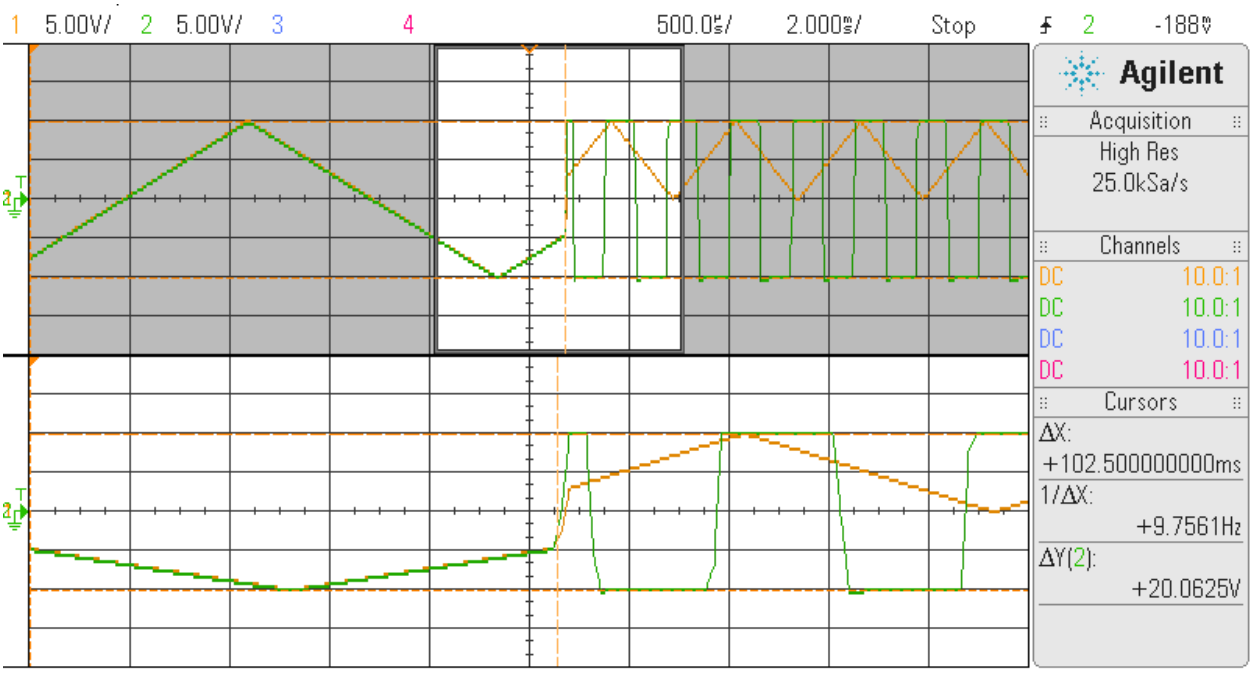
Example:

```
AOCC_aos_file_list_t flist[CCURAOCC_MAX_CHANNELS+1];
memset(flist, 0,sizeof(flist);
flist[0].ChanMask  = 0x000F;    /* select channels 0, 1,2 and 3 */
strcpy(flist[0].WaveFile, "Sine_Wave"); /* select user wave */
flist[0].frequency = 207.0;     /* run earlier defined wave at 207Hz */
```

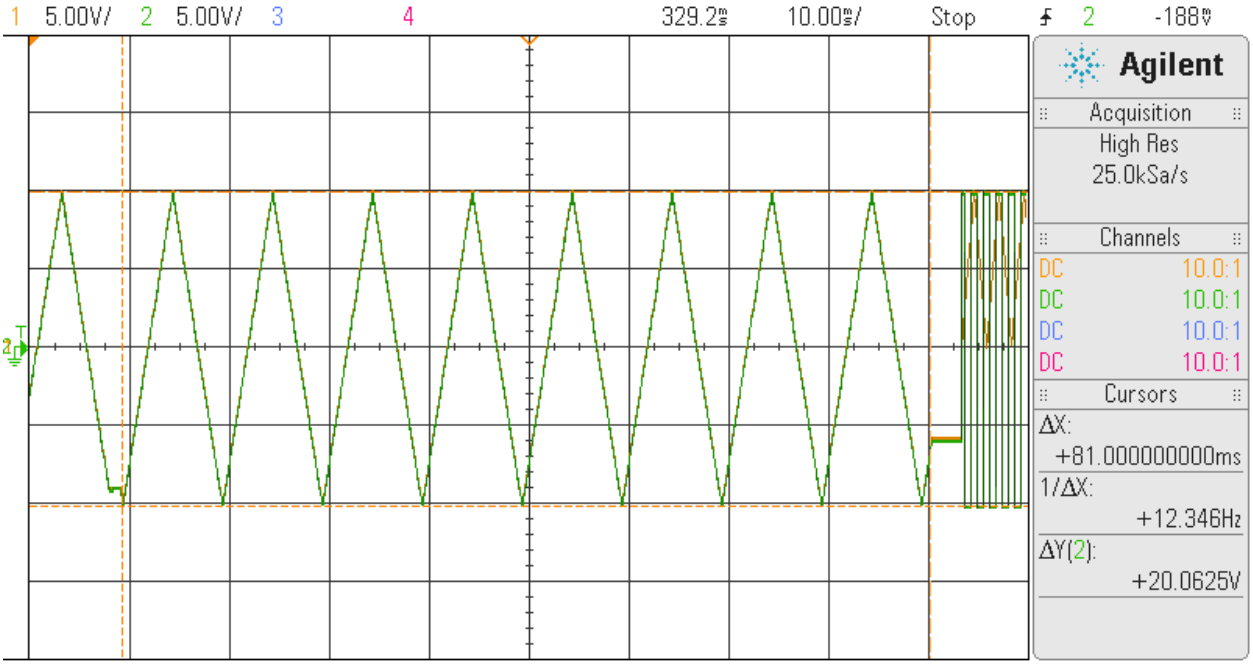
```
flist[0].amplitude = 1.0;        /* do not change amplitude of original wave */
flist[0].bias      = 0.0;        /* do not add any offset to the original wave */
flist[0].phase     = +180.0;     /* change phase of wave */

flist[1].ChanMask  = 0x00F0;     /* select channels 4, 5, 6 and 7 */
strcpy(flist[1].WaveFile, "Custom_Wave");  /* select user wave */
flist[1].frequency = 47.0;       /* run earlier defined wave at 47Hz */
flist[1].amplitude = 0.5;        /* reduce wave to half of original wave */
flist[1].bias      = -1.0;       /* add -1 volt offset to original wave */
flist[1].phase     = 0.0;        /* do not change phase of wave */

flist[2].ChanMask  = 0;          /* end of list */

if(AOCC_ActionChangeChanWave(handle, flist, 0))  /* seamless */
    exit(1);
if(AOCC_ActionChangeChanWave(handle, flist, 1))  /* immediate=1 */
    exit(1);
```



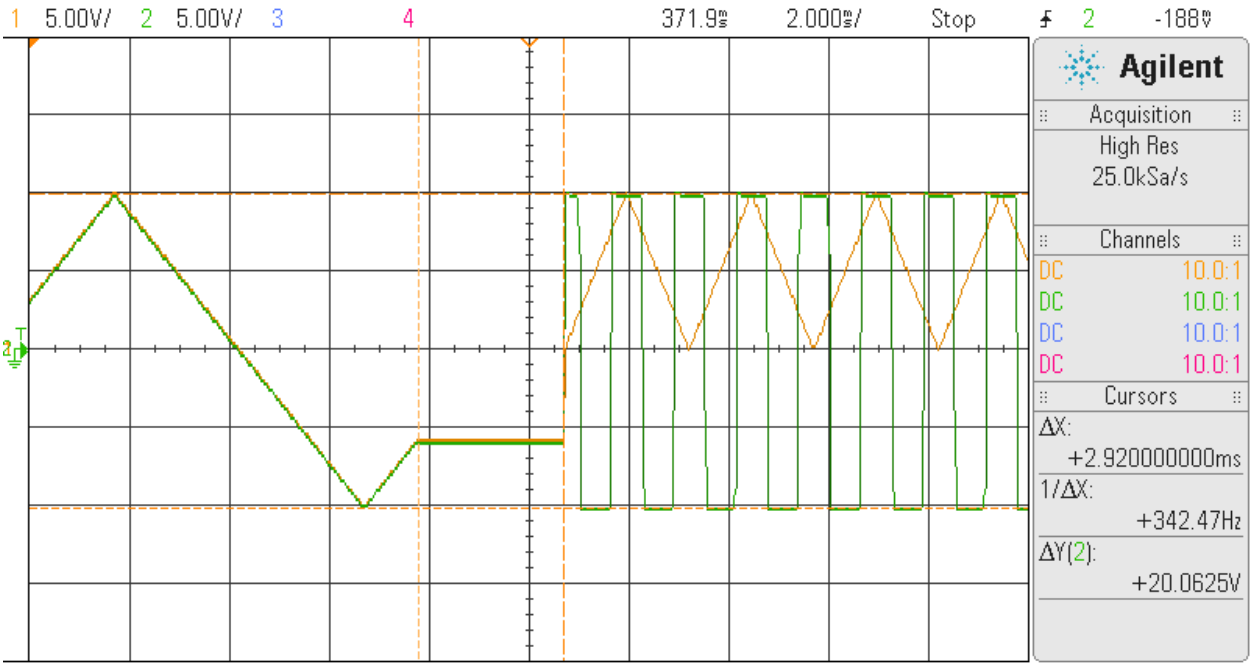
This time we are demonstrating a seamless transition of two channels. Here, both Ch0 and Ch31 are generating a triangle wave at 100Hz. Ch0 is changed to a 400Hz Triangle Wave of size 32K samples in the wave file. Its phase is 0, amplitude is 0.5 (half the new wave) and bias is 5.0 volts. Ch31 is changed to a 800Hz Square Wave of size 256 samples in the wave file. Its phase is 90 degrees, amplitude is 1 and bias is 0. The above change occurs *seamlessly*. Also, a "hook" was placed in the test program to stop/start the system clock just prior to calling the change wave call. The reason is to see "spot" in the trace where the clock stops/starts (shown as a flat line). Measuring the distance from the end of the flat line (when the clock is restarted) to the point where the two waves change wave shows approximately 103ms. This is because at the time the change phase was issued, there were several samples, both in the hardware FIFO and the internal wave generation utility that were to be sent prior to the wave change. Additional time is required to read the entire sample file from the disk and apply the calibration, amplitude and gain to each sample point. There are tunable parameters for the DMA size, the number of DMA buffers and the number of Multiplexor buffers that are under the control of the user during the opening of the stream that can be changed to shorten the switching time. Currently, the default settings of DMA size (16KSamples), number of DMA buffers (4) and number of Multiplexor buffers (3) are used during this run. The full 128K FIFO is used in this run.



This is another snapshot of the same wave showing the *seamless* transitions to the new waves.



The above display is similar to the earlier display, however, this time we set the immediate value to a '1'. In this case, we see that it takes approximately 81ms from the time the command is issued to the time we flush the FIFO and internal buffers. This is the time required to read the large sample files from the disk, apply the calibration, amplitude and bias. It then takes approximately 3ms to generate the multiplexed buffer and prime the partial FIFO. Once again, the first "flat" line on the left where the vertical red dotted line appears is the starting of the clock after it was stopped. The second vertical red dotted line on the right is when the API stops the clocks to flushes the FIFO and buffers. The "flat" line following that is the time it takes to fill the FIFO before resuming the clocks.



This is the same wave where we zoom into where the transition of the waves occurs. In this case, all waves start at zero offset.

8.6. AOCC_ActionLinkWave()

Function: int AOCC_ActionLinkWave(void *handle, AOCC_aos_file_list_t *flist, int remove_waves)

Return: good return - AOCC_ERROR_NONE
Error - one of various errors

Description: This API selects a new set of waves, frequency, phase, amplitude and bias for a set of channels. User specifies a list of channel masks, wave files, frequency, phase, amplitude and bias via the *flist* structure. If the user enters a negative frequency, no frequency adjustment is applied to the basic wave. The wave runs at the boards sampling rate, i.e. board sampling frequency divided by the number of samples in the wave. A phase of 0.0, -360.0 or +360.0 will not result in any phase shift. An amplitude of 1 or a bias of 0 will not affect the original wave. If user has specified calibration for the channel, the amplitude will be multiplied by any calibration gain and the bias will be added to any calibration offset. The user will need to make sure that the resulting waves range does not exceed the maximum voltage range for the board; otherwise, clipping will occur and warning messages will be generated.

The *remove_waves* option when set to '1' causes the API to stop all previously active waves if present and replaces them immediately with this new set of waves. If the *remove_waves* option is set to '0', the currently active waves are not affected, however, these new linked waves are simply linked to the previously set of waves (or linked waves). The only way to now select these new linked waves is with the help of the *AOCC_ActionSelectNextwave()* API. In this way, the user can chain several sequences of waves without interrupting activity on current waves and when ready, simply use the *AOCC_ActionSelectNextwave()* API each time to sequence through each linked waves in turn until the last linked wave is reached. Note that when the next *wave* is selected, the previous wave set is stopped and removed.

The *wave* files are user defined and have two formats. *FORMAT_FLOAT* or *FORMAT_HEX*. This token needs to be supplied in the wave file somewhere at the beginning of the *wave* file *(for further information, refer to Section 10)*. *FORMAT_FLOAT* is a wave file that is filled with floating point voltage values. The voltages specified (including any offset/gain calibration applied) must lie within the maximum voltage range of the board; otherwise, errors will be generated. The *FORMAT_HEX* wave file is a wave file that contains raw hex representation of voltages. The user needs to make sure that the hex representation corresponds to the correct board format, i.e. OFFSET_BINARY or TWOS_COMPLEMENT otherwise incorrect waves will be generated. Offset/gain calibration can be applied to either wave format. Sample wave files are located in the *…./test/WaveFiles* directory.

```
/* file list */
typedef struct _AOCC_aos_file_list_t {
    uint    ChanMask;               /* channel mask */
    double  frequency;              /* channel frequency */
    char    WaveFile[PATH_MAX];     /* channel wave file */
    double  amplitude;              /* channel amplitude (really gain) */
    double  bias;                   /* channel bias (really offset) */
    double  phase;                  /* channel initial phase (-360.0 -
                                       +360.0)*/
} AOCC_aos_file_list_t;
```

Example:

```
AOCC_aos_file_list_t flist[CCURAOCC_MAX_CHANNELS+1];
memset(flist, 0,sizeof(flist);
flist[0].ChanMask  = 0x8001;    /* select channels 0 and 15 */
strcpy(flist[0].WaveFile, "Wave_A"); /* select user wave */
flist[0].frequency = 256.0;     /* run earlier defined wave at 256Hz */
```

```
flist[0].amplitude = 1.5;       /* increase wave to 1.5 times original wave */
flist[0].bias      = 0.2;       /* add 200 millivolt offset to original wave */
flist[0].phase     = -45.0;     /* change phase of wave */

flist[1].ChanMask  = 0x0020;    /* select channel 5 */
strcpy(flist[1].WaveFile, "Custom_Wave"); /* select user wave */
flist[1].frequency = 100.7;     /* run earlier defined wave at 100.7Hz */
flist[1].amplitude = 2.0;       /* double the original wave */
flist[1].bias      = 0.0;       /* do not add offset to original wave */
flist[1].phase     = +12.5;     /* change phase of wave */

flist[2].ChanMask  = 0;         /* end of list */

if(AOCC_ActionLinkWave(handle, flist, 1))/* replace old waves and
     exit(1);                             start immediately */

if(AOCC_ActionLinkWave(handle, flist, 0)) /* simply link to previous
     exit(1);                             set of linked waves */

/* There is no limit to the number of wave sets that can be linked */

/* NOTE!!! If the remove_waves flag is set at any time, all previously linked wave sets
 * are removed and the new one put at the head of the chain and started. */
```

8.7. AOCC_ActionRemoveWaves()

Function: AOCC_ActionRemoveWaves(void *handle)

Return: good return - AOCC_ERROR_NONE
Error - one of various errors

Description: This API removes all existing wave sets and linked wave sets. The board stops generating any waves in progress.

Example:
```
AOCC_ActionRemoveWaves(handle);
```

8.8. AOCC_ActionSelectNextWave()

Function: AOCC_ActionSelectNextWave(void *handle)

Return: none

Description: This API starts the next linked wave set if it has been previously created by the AOCC_ActionLinkWave() API. Prior to starting the next linked wave set, the currently active wave set will be stopped and removed. If there is no new linked wave set, then no action takes place.

Example:
```
AOCC_ActionSelectNextWave(handle);
```

8.9. AOCC_CloseAOStream()

Function: int AOCC_CloseAOStream(void *handle)

Return: good return - AOCC_ERROR_NONE
Error - one of various errors

Description: This API stops all wave activities and closes a previously opened AOCC Stream that was opened with the AOCC_OpenAOStream() API. No further APIs will operate on this handle. A new AOCC_OpenAOStream() call will need to be issued to start a new wave generation.

Example:
```
if(AOCC_CloseAOStream(handle))
     exit(1);
```

8.10. AOCC_CommandAOStream()

Function: int AOCC_CommandAOStream(void *handle, int cmd)

Return: good return - AOCC_ERROR_NONE
Error - one of various errors

Description: This API issues several basic AO Stream commands.

Command: CMD_SETUP - this is the first command to be used the first time the board is opened with the AOCC_OpenAOStream() command. When the wave generator is running, you will need to stop the wave generation prior to issuing the CMD_SETUP command.
CMD_RUN - start running the loaded wave set. A wave must be loaded prior to

issuing this command with the AOCC_ActionLinkWave() command.
CMD_STOP - stop an actively running wave set.
CMD_PAUSE - suspend an actively running wave set. Clocks are stopped and all channels will generate a constant output until the CMD_RESUME is issued and waves will resume where left off.
CMD_RESUME - resume a previously paused wave set. (start the clocks)
CMD_TERMINATE - terminate the AOCC Stream. Once this is called, no further wave action can be performed. Users will need to close and open a new AOCC_Stream session.

Example:
```
If(AOCC_CommandAOStream(handle, CMD_SETUP)     /* setup the board */
     Exit(1);
If(AOCC_CommandAOStream(handle, CMD_RUN)       /* start wave session */
     Exit(1);
If(AOCC_CommandAOStream(handle, CMD_STOP)      /* stop wave session */
     Exit(1);
If(AOCC_CommandAOStream(handle, CMD_PAUSE)     /* pause wave session */
     Exit(1);
If(AOCC_CommandAOStream(handle, CMD_RESUME)    /* resume wave session */
     Exit(1);
If(AOCC_CommandAOStream(handle, CMD_TERMINATE)/* terminate AOStream */
     Exit(1);
```

8.11. AOCC_DebugAOStream()

Function: int AOCC_DebugAOStream(int bno, int on_off, int *state)

Return: good return - AOCC_ERROR_NONE
Error - one of various errors

Description: This API enables generation of debug messages for AOCC_Stream. It can be issued even prior to opening the device as a *handle* is not required. This can severely impact performance so care must be used to enable this feature. Users can turn on or off debugging as necessary and the current debug state is returned if state in not NULL. Debugging needs to be compiled in the AOCC_Stream Library with the *_AOCC_AOSTREAM_DEBUG* flag before it is enabled.

Example:
```
int current_state;
if(AOCC_DebugAOStream(board_no, AO_ON, &current_state))
     exit(1);                                   /* enable debug */

if(AOCC_DebugAOStream(board_no, AO_OFF, &current_state))
     exit(1);                                   /* disable debug */
```

8.12. AOCC_ErrorAOStream()

Function: int AOCC_ErrorAOStream(void *handle, AOCC_aos_error_t *error)

Return: good return - AOCC_ERROR_NONE
Error - one of various errors

Description: During running of the AOCC Stream wave generation, it is possible to encounter errors. This is reported in the form of error messages displayed on *stderr.* Additionally, some use information for the errors is stored internally in the library and can be made available with this call. The call also contains information of any errors that have been generated by the driver and the driver library.

```
typedef struct {
    ccuraocc_lib_error_t  lib_error;        /* last library error */
    ccuraocc_user_error_t driver_error;     /* last driver error */
    ccuraocc_lib_error_t  aocc_error;       /* last AOCC Stream error */
} AOCC_aos_error_t;


typedef struct _ccuraocc_lib_error_t
{
    uint error;                                         /* lib error number */
    char name[CCURAOCC_LIB_ERROR_NAME_SIZE];            /* error name used in lib */
    char desc[CCURAOCC_LIB_ERROR_DESC_SIZE];            /* error description */
    int  line_number;                                   /* line number in library */
    char function[CCURAOCC_LIB_ERROR_FUNC_SIZE];        /* library function */
} ccuraocc_lib_error_t;

typedef struct _ccuraocc_user_error_t
{
    uint error;                                 /* error number */
    char name[CCURAOCC_ERROR_NAME_SIZE];        /* error name used in driver */
    char desc[CCURAOCC_ERROR_DESC_SIZE];        /* error description */
} ccuraocc_user_error_t;
```

Example:

```
AOCC_aos_error_t    Error;

AOCC_ErrorAOStream(handle, &Error);    /* check for errors */

if(Error.lib_error.error) {
    fprintf(stderr,"=============================================\n");
    fprintf(stderr,"last driver library error information:\n");
    fprintf(stderr,"   error:   %d\n",Error.lib_error.error);
    fprintf(stderr,"    name:   %s\n",Error.lib_error.name);
    fprintf(stderr,"    desc:   %s\n",Error.lib_error.desc);
    fprintf(stderr,"=============================================\n");
}

if(Error.driver_error.error) {
    fprintf(stderr,"=============================================\n");
    fprintf(stderr,"last driver error information:\n");
    fprintf(stderr,"   error:   %d\n",Error.driver_error.error);
    fprintf(stderr,"    name:   %s\n",Error.driver_error.name);
    fprintf(stderr,"    desc:   %s\n",Error.driver_error.desc);
    fprintf(stderr,"=============================================\n");
}

if(Error.aocc_error.error) {
    fprintf(stderr,"=============================================\n");
    fprintf(stderr,"last AOCC Stream error information:\n");
    fprintf(stderr,"   error:   %d\n",Error.aocc_error.error);
    fprintf(stderr,"    line:   %d\n",Error.aocc_error.line_number);
    fprintf(stderr,"    func:   %s\n",Error.aocc_error.function);
    fprintf(stderr,"    name:   %s\n",Error.aocc_error.name);
    fprintf(stderr,"    desc:   %s\n",Error.aocc_error.desc);
    fprintf(stderr,"=============================================\n");
}
```

8.13. AOCC_InfoAOStream()

Function: int AOCC_InfoAOStream(void *handle, AOCC_aos_info_t *info)

Return: good return - AOCC_ERROR_NONE
Error - one of various errors

Description: This API returns useful information of AOCC_Stream.

Example:

```
AOCC_aos_info_t info;
if(AOCC_InfoAOStream(handle, &info))
     exit(1);
```

```
/* lib_info struct */
typedef struct {
    ushort              BoardNo;
    ccuraocc_board_info_t BoardInfo;       /* board information */
    AOCC_aos_init_options_t  opts;         /* user options */
    ccuraocc_local_ctrl_data_t   *LocalPtr;
    ccuraocc_config_local_data_t *ConfigPtr;
    char                CalFile[PATH_MAX]; /* calibration file in use */
    uint                ChanMask;          /* channel mask */
    uint                nchans;            /* number of channels */
    uint                hw_nchans;         /* available h/w channels */
    uint                hw_ChanMask;       /*available h/w channels mask*/
    uint                differential;      /* differential/single-ended */
    int                 command;           /* cmd sent to thread */
    uint                write_count;       /* write count */
    uint                mlw_index;         /* mux list write index */
    uint                mlr_index;         /* mux list read index */
    int                 InputClockSelect;  /* input update clock select */
    int                 OutputClockSelect; /* output clock select */
    void                *MyHandle;         /* pointer to library handle */
    uint                DmaAbortCount;     /* DMA abort count */
    _AOCC_aos_statistics_t  stats;         /* statistics */
} AOCC_aos_info_t;
```

```
/* board information */
typedef struct
{
    int     board_id;           /* board id */
    int     board_type;         /* board type */
    int     firmware_rev;       /* firmware revision */
    int     board_wiring;       /* single-ended or differential
           */
    int     number_of_channels; /* number of hardware channels
           */
    int     all_channels_mask;  /* all channels mask */
    int     all_converters_mask;/* all converters mask */
    double  cal_ref_voltage;    /* calibration reference voltage
           */
    double  voltage_range;      /* maximum voltage range */
    double  MinSampleFreq;      /* minimum sample frequency */
    double  MaxSampleFreq;      /* maximum sample frequency */
    double  MasterClock;        /* master clock */
} ccuraocc_board_info_t;
```

```
/* user input supplied to AOCC_OpenAOStream() */
typedef struct {
    double  BoardSampleFreq;       /* sample frequency to run board at */
```

```
    double  ActualBoardSampleFreq;     /* actual sample frequency of board -
           returned */
    int      InputClockSelect;       /* sample clock selection */
    int      OutputClockSelect;      /* output clock selection */
    int      Format[CCURAOCC_MAX_CHANNELS];     /* offset binary or two's
           complement */
    int      VoltageRange[CCURAOCC_MAX_CHANNELS];/* output voltage
           range */
    int      SchedulePolicy;         /* thread schedule policy */
    int      ThreadPriority;         /* thread priority */
    int      cpuAffinity;            /* CPU on which MuxThread will run */
    int      cpuCount;               /* no. of cpus to run on starting at base */
    void     *MyHandle;              /* returned CCURAOCC library handle */
    int      Calibrate;              /* 1=use offset/gain, 0=raw data mode */
    char     *CalDir;                /* name of calibration directory */
    int      KiloSamplesPerWrite;    /* write size in KiloSamples */
    int      FifoSizeKiloSamples;    /* fifo size in KiloSamples */
    int      NumberOfWriteBuffers;   /* number of write buffers */
    int      NumberOfMultiplexBuffers;  /* number of Multiplex buffers */
} AOCC_aos_init_options_t;

/* statistics */
typedef struct {
    uint     fifo_overflow;          /* fifo overflow count */
    uint     fifo_empty;             /* fifo empty count */
    uint     fifo_over_threshold;    /* fifo over threshold count */
    uint     issue_sw_trigger;       /* s/w trigger issue count */
    uint     write_failure;          /* write failure count */
    uint     write_size_mismatch;    /* write size mismatch count */
} _AOCC_aos_statistics_t;
```

8.14. AOCC_OpenAOStream()

Function: int AOCC_OpenAOStream(int bno, AOCC_aos_init_options_t *opts, void **handle)

Return: good return - AOCC_ERROR_NONE
Error - one of various errors

Description: This API opens an AOCC_Stream for a specific board. If a stream is already opened, the call will fail. Once successfully opened, the user can then start controlling waves. This API returns a handle that will be used for various APIs. The AOCC_aos_init_options_t struct needs to be filled prior to calling AOCC_OpenAOStream().

Parameters: *bno* – Board number – 0 to 15. Device /dev/ccuraocc_wave<bno> must exist.
The following parameter are located in the AOCC_aoc_init_options_t struct.
BoardSampleFreq – Sample frequency of 0 or negative tells API to use maximum board sample frequency. In this case, we have a maximum frequency of 400000 SPS. The minimum sample frequency is 0.2SPS.
ActualBoardSampleFreq – This is the actual board sampling frequency that is returned to the user. This may be different from the user supplied *BoardSampleFreq* if this frequency is not exactly divisible by the clock reference frequency of the board.
InputClockSelect–
- *CCURAOCC_CONVERTER_UPDATE_SELECT_EXTERNAL_CLOCK*
- *CCURAOCC_CONVERTER_UPDATE_SELECT_PLL_CLOCK*

OutputClockSelect –
- *CCURAOCC_BCSR_EXTCLK_OUTPUT_PLL_CLOCK*
- *CCURAOCC_BCSR_EXTCLK_OUTPUT_EXTERNAL_CLOCK*

Format []– *CCURAOCC_CONVERTER_TWOS_COMPLEMENT*
- *CCURAOCC_CONVERTER_OFFSET_BINARY*

VoltageRange[] – *CCURAOCC_CONVERTER_UNIPOLAR_5V*
- *CCURAOCC_CONVERTER_UNIPOLAR_10V*
- *CCURAOCC_CONVERTER_BIPOLAR_5V*
- *CCURAOCC_CONVERTER_BIPOLAR_10V*
- *CCURAOCC_CONVERTER_BIPOLAR_2_5V*

SchedulePolicy – *SCHED_FIFO, SCHED_RR* or *SCHED_OTHER.*
ThreadPriority – *0* to *99* if *SCHED_FIFO or SCHED_RR is selected.*
cpuAffinity – This is not a mask but a CPU number on which to run the MuxThread() and the WorkerThread(). A value of 0, less than 0 or greater than the number of CPUs available, indicates it is allowed to run on any available CPU.
cpuCount – Number of CPUs on which to run the MuxThread() starting with cpuAffinity. A count of less than or equal to 0 indicates a count of 1. A count that causes CPU assignments to go beyond last CPU will cause it to run on all CPUs starting with cpuAffinity. If cpuAffinity is less than or equal to 0, this option is ignored.
Calibrate – *AOCC_USE_OFFSET_GAIN* or *AOCC_RAW_DATA_MODE.* The latter option basically disables calibration. *(refer to Section 11 for further information).*
CalDir – This is the first directory to search for calibration files named *16AO16.<bno + 1>* if not *'NULL'.* If a calibration file is not found in this directory, then the current directory where the API is executed is searched. If not found in the current directory, then the */usr/lib/config/CAL* directory is searched for the calibration file. If no calibration file is found, offset/gain calibration is not performed for the board. Refer to the *…test/CCURAOCC.1.ex* calibration file for an example of its usage.

KiloSamplesPerWrite – This represents the size of each DMA buffer. Minimum size is 1 (Kilosample) and Maximum size is 96 (KiloSamples). Default is 16.

FifoSizeKiloSamples – This count represents the hardware FIFO size in KiloSamples. The maximum hardware FIFO is 128K samples and so is the maximum value of this parameter. It also defaults to this maximum value of 128K, i.e. uses the full FIFO that is available. The minimum size the FIFO can be set is 2 (KiloSamples). One must ensure that this FIFO size must always be greater that the *KiloSamplesPerWrite* parameter.

NumberOfWriteBuffers – This count represents the number of DMA buffers. Minimum is 1, Maximum is 16 and the default is 4.

NumberOfMultiplexBuffers – This count represents the number of Multiplexor buffers. Minimum is 1, Maximum is 16. Default is 3.

Example:

```
AOCC_aos_init_option_t opts;
int board_no=0;
int Chan;
void *handle=NULL;
opts.BoardSampleFreq = 0; /* select max board sample freq */
opts.InputClockSelect=
                  CCURAOCC_CONVERTER_UPDATE_SELECT_PLL_CLOCK;
opts.OutputClockSelect= CCURAOCC_BCSR_EXTCLK_OUTPUT_PLL_CLOCK;
for(Chan=0; Chan < CCURAOCC_MAX_CHANNELS; Chan++) {
     opts.Format[Chan] = CCURAOCC_CONVERTER_OFFSET_BINARY;
     opts.VoltageRange[Chan]= CCURAOCC_CONVERTER_BIPOLAR_10V;
}
opts.SchedulePolicy  = SCHED_FIFO;
opts.ThreadPriority  = 50;
opts.cpuAffinity     = 3;
opts.cpuCount        = 2;
op.Calibrate         = AOCC_USE_OFFSET_GAIN;
opts.CalDir          = NULL;
opts.KiloSamplesPerWrite      = AOCC_KSAMPLE_DEFAULT;
opts.FifoSizeKiloSamples      = AOCC_KSAMPLE_FIFOSIZE_DEFAULT;
opts.NumberOfWriteBuffers     = AOCC_NUMWRITEBUFS_DEFAULT;
opts.NumberOfMultiplexBuffers = AOCC_NUMUXBUFS_DEFAULT;

if(AOCC_OpenAOStream(board_no, &opts, &handle)) /* open stream */
     exit(1);
```

8.15. AOCC_PrintFifoStats()

Function: int AOCC_PrintFifoStats(void *handle, FILE *fd)

Return: good return - AOCC_ERROR_NONE
Error - one of various errors

Description: This API prints useful information of FIFO statistics to a user specified file descriptor.

Example:
```
if(AOCC_PrintFifoStats(handle, stderr))
    exit(1);
```

Sample Printout:
FifoData: cur=18854 min=18520 max=20000 av=1adb0

8.16. AOCC_PrintInfoAOStream()

Function: int AOCC_PrintInfoAOStream(void *handle, FILE *fd)

Return: good return - AOCC_ERROR_NONE
Error - one of various errors

Description: This API prints useful information of AOCC_Stream to a user specified file descriptor.

Example:
```
if(AOCC_PrintInfoAOStream(handle, stderr))
    exit(1);
```

Sample Printout:
```
============================================================
BoardNo                       = 0
  Max Channels                = 32
  Master Clock                = 65.536
  Minimum Sample Freq         = 0.200
  Maximum Sample Freq         = 400000.000
  Firmware Revision           = 0x001
  Board Type                  = 2 (32-Channel, 10-Volt, Single-Ended Card)
  Board ID                    = 0x9287
  Board Wiring                = Single-Ended
 opts: Board Sample Freq      = 400000.000000 (Requested=400000.000000)
 opts:  0. Format             = 0 (CCURAOCC_CONVERTER_OFFSET_BINARY)
 opts:  0. voltage range      = 3 Bipolar 10 Volts (+/- 10V)
 opts:  1. Format             = 0 (CCURAOCC_CONVERTER_OFFSET_BINARY)
 opts:  1. voltage range      = 3 Bipolar 10 Volts (+/- 10V)
 opts:  2. Format             = 0 (CCURAOCC_CONVERTER_OFFSET_BINARY)
 opts:  2. voltage range      = 3 Bipolar 10 Volts (+/- 10V)
 opts:  3. Format             = 0 (CCURAOCC_CONVERTER_OFFSET_BINARY)
 opts:  3. voltage range      = 3 Bipolar 10 Volts (+/- 10V)
 opts:  4. Format             = 0 (CCURAOCC_CONVERTER_OFFSET_BINARY)
 opts:  4. voltage range      = 3 Bipolar 10 Volts (+/- 10V)
 opts:  5. Format             = 0 (CCURAOCC_CONVERTER_OFFSET_BINARY)
 opts:  5. voltage range      = 3 Bipolar 10 Volts (+/- 10V)
 opts:  6. Format             = 0 (CCURAOCC_CONVERTER_OFFSET_BINARY)
 opts:  6. voltage range      = 3 Bipolar 10 Volts (+/- 10V)
 opts:  7. Format             = 0 (CCURAOCC_CONVERTER_OFFSET_BINARY)
 opts:  7. voltage range      = 3 Bipolar 10 Volts (+/- 10V)
 opts:  8. Format             = 0 (CCURAOCC_CONVERTER_OFFSET_BINARY)
 opts:  8. voltage range      = 3 Bipolar 10 Volts (+/- 10V)
 opts:  9. Format             = 0 (CCURAOCC_CONVERTER_OFFSET_BINARY)
 opts:  9. voltage range      = 3 Bipolar 10 Volts (+/- 10V)
 opts: 10. Format             = 0 (CCURAOCC_CONVERTER_OFFSET_BINARY)
 opts: 10. voltage range      = 3 Bipolar 10 Volts (+/- 10V)
 opts: 11. Format             = 0 (CCURAOCC_CONVERTER_OFFSET_BINARY)
 opts: 11. voltage range      = 3 Bipolar 10 Volts (+/- 10V)
```

```
opts: 12. Format            = 0 (CCURAOCC_CONVERTER_OFFSET_BINARY)
opts: 12. voltage range     = 3 Bipolar 10 Volts (+/- 10V)
opts: 13. Format            = 0 (CCURAOCC_CONVERTER_OFFSET_BINARY)
opts: 13. voltage range     = 3 Bipolar 10 Volts (+/- 10V)
opts: 14. Format            = 0 (CCURAOCC_CONVERTER_OFFSET_BINARY)
opts: 14. voltage range     = 3 Bipolar 10 Volts (+/- 10V)
opts: 15. Format            = 0 (CCURAOCC_CONVERTER_OFFSET_BINARY)
opts: 15. voltage range     = 3 Bipolar 10 Volts (+/- 10V)
opts: 16. Format            = 0 (CCURAOCC_CONVERTER_OFFSET_BINARY)
opts: 16. voltage range     = 3 Bipolar 10 Volts (+/- 10V)
opts: 17. Format            = 0 (CCURAOCC_CONVERTER_OFFSET_BINARY)
opts: 17. voltage range     = 3 Bipolar 10 Volts (+/- 10V)
opts: 18. Format            = 0 (CCURAOCC_CONVERTER_OFFSET_BINARY)
opts: 18. voltage range     = 3 Bipolar 10 Volts (+/- 10V)
opts: 19. Format            = 0 (CCURAOCC_CONVERTER_OFFSET_BINARY)
opts: 19. voltage range     = 3 Bipolar 10 Volts (+/- 10V)
opts: 20. Format            = 0 (CCURAOCC_CONVERTER_OFFSET_BINARY)
opts: 20. voltage range     = 3 Bipolar 10 Volts (+/- 10V)
opts: 21. Format            = 0 (CCURAOCC_CONVERTER_OFFSET_BINARY)
opts: 21. voltage range     = 3 Bipolar 10 Volts (+/- 10V)
opts: 22. Format            = 0 (CCURAOCC_CONVERTER_OFFSET_BINARY)
opts: 22. voltage range     = 3 Bipolar 10 Volts (+/- 10V)
opts: 23. Format            = 0 (CCURAOCC_CONVERTER_OFFSET_BINARY)
opts: 23. voltage range     = 3 Bipolar 10 Volts (+/- 10V)
opts: 24. Format            = 0 (CCURAOCC_CONVERTER_OFFSET_BINARY)
opts: 24. voltage range     = 3 Bipolar 10 Volts (+/- 10V)
opts: 25. Format            = 0 (CCURAOCC_CONVERTER_OFFSET_BINARY)
opts: 25. voltage range     = 3 Bipolar 10 Volts (+/- 10V)
opts: 26. Format            = 0 (CCURAOCC_CONVERTER_OFFSET_BINARY)
opts: 26. voltage range     = 3 Bipolar 10 Volts (+/- 10V)
opts: 27. Format            = 0 (CCURAOCC_CONVERTER_OFFSET_BINARY)
opts: 27. voltage range     = 3 Bipolar 10 Volts (+/- 10V)
opts: 28. Format            = 0 (CCURAOCC_CONVERTER_OFFSET_BINARY)
opts: 28. voltage range     = 3 Bipolar 10 Volts (+/- 10V)
opts: 29. Format            = 0 (CCURAOCC_CONVERTER_OFFSET_BINARY)
opts: 29. voltage range     = 3 Bipolar 10 Volts (+/- 10V)
opts: 30. Format            = 0 (CCURAOCC_CONVERTER_OFFSET_BINARY)
opts: 30. voltage range     = 3 Bipolar 10 Volts (+/- 10V)
opts: 31. Format            = 0 (CCURAOCC_CONVERTER_OFFSET_BINARY)
opts: 31. voltage range     = 3 Bipolar 10 Volts (+/- 10V)
opts: Schedule Policy       = 1 (SCHED_FIFO)
opts: Thread Priority       = 50
opts: CPU Affinity          = 0 (Any CPU)
opts: CPU Count             = 1
opts: Calibrate             = 1 (YES - OFFSET/GAIN)
opts: Calibration Dir.      = <None>
opts: MyHandle Pointer      = 0x643f00
opts: Kilo Samples/Write    = 16 Ksamples (16384 samples)
opts: FIFO Size             = 128 Ksamples (131072 samples)
opts: Num of Write Bufs     = 4
opts: Num of Mux Bufs       = 3
Local Register Pointer      = 0x7ffff7ffc000
COnfig Register Pointer     = 0x7ffff7ffb800
DMA Write Size              = 65536 (0x00010000) bytes
Calibration File in use     = <None>
ChanMask                    = 0xffffffff
nchans                      = 32
hw_nchans                   = 32
hw_ChanMask                 = 0xffffffff
command                     = 2
AOCC_aostream_debug         = 0x00000000
write_count                 = 2
mlw_index                   = 5
mlr_index                   = 2
Input Clock Selection       = Converter Update Select: PLL Clock
Output Clock Selection      = External Clock Output: PLL Clock
============================================================
```

8.17. AOCC_PrintTimingAOStream()

Function: int AOCC_PrintTimingAOStream(void *handle, FILE *fd, int what)

Return: good return - AOCC_ERROR_NONE
Error - one of various errors

Description: This API prints timing information for AOCC_Stream. This can severely impact performance so care must be used to enable this feature. Timing needs to be compiled in the AOCC_Stream Library with the *_AOCC_AOSTREAM_TIMING* flag before it is enabled.

Parameters: *TIME_WRITE* – time taken for DMA writes to FIFO.
TIME_MUXW – time taken to multiplex wave buffers.
TIME_DRAIN – time to drain FIFO once it has reached the threshold.
TIME_TBW – time between consecutive writes.

Example:
```
int what = TIME_WRITE | TIME MUXW | TIME_DRAIN | TIME_TBW;
if(AOCC_PrintTimingAOStream(handle, stderr, what))
     exit(1);                          /* print timing info */
```

9. Understanding the wave generation mechanism

The basic concept of the wave generation program is to first read the user supplied wave forms which are in the form of ASCII files on a per channel basis, and apply the necessary calibration, amplitude, bias and phase against each sample point in the wave for the given channel. For each active channel, this information is then stored in internal buffers which are then multiplexed into single buffers that are then written out to the hardware FIFO for wave generation.

When the wave generation is running, the process of continuous multiplexing of all active channels is taking place. Frequency and phase changes are accomplished without altering the samples of individual channels. Any other changes to running waves involve reprocessing the individual channel file prior to multiplexing with the remaining active channels.

For seamless operation of wave changes, the wave generation ensures that the waves on the all the active channels continue undisturbed while the appropriate changes are applied to the requested channels. There is a finite time, usually in milliseconds that take place from the instant the change API is issued to the actual change being reflected on the channel output. A mechanism has been provided during opening of the wave generation to affect this latency time. Understanding these tunable parameters can provide some insight into this latency.

The following four tunable parameters are available that could affect the latency.

1. ***KiloSamplesPerWrite***: Size of DMA writes in kilo-samples. Default is *AOCC_KSAMPLE_DEFAULT* (16). Minimum is *AOCC_KSAMPLE_WRITE_MIN* (1), maximum is *AOCC_KSAMPLE_WRITE_MAX* (96).
2. ***FifoSizeKiloSamples***: Limit the FIFO size during wave generation. Default is AOCC_KSAMPLE_FIFOSIZE_DEFAULT (128). Minimum is AOCC_KSAMPLE_FIFOSIZE_MIN (2), maximum is AOCC_KSAMPLE_FIFOSIZE_MAX (128).This size must always be larger than *KiloSamplesPerWrite*
3. ***NumberOfWriteBuffers***: Number of write buffers. Default is *AOCC_NUMWRITEBUFS_DEFAULT* (4). Minimum is *AOCC_NUMWRITEBUFS_MIN* (1), maximum is *AOCC_NUMWRITEBUFS_MAX* (16).
4. ***NumberOfMultiplexBuffers***: Number of multiplexor buffers. Default is *AOCC_NUMUXBUFS_DEFAULT* (3). Minimum is *AOCC_NUMUXBUFS_MIN* (1), maximum is *AOCC_NUMUXBUFS_MIN* (16).

The hardware FIFO is 128 Kilo-samples in size. If all 32 channels are active and running at the maximum sampling rate of 400,000 samples/second, it will take 10.24 milliseconds to drain the entire FIFO from full. Normally, the wave generation attempts to keep the FIFO full until no more room is available. Once full, it blocks for 1.5 milliseconds while the next multiplex operation takes place. A software parameter *FifoSizeKiloSamples* has been provided to the user to reduce the size of the FIFO. Normally, this should not be done when using all 32 channels at the fastest sampling rate. When less number of channels are active, then it will take longer to drain the FIFO and hence the latency for frequency, phase and wave changes would take longer. The smaller the FIFO size, the more chances there are of an under-run occurring.

Based on the above default parameters, the size of each DMA buffer is 4*16 = 64 Kilo-samples. There are 3 such buffers, called the multiplexor buffers. While the wave generation is running, one thread is busy filling empty multiplexor buffers, while the other thread is writing out the multiplexor buffer to the hardware FIFO as the samples drain.

The latency time is therefore determined by the number of samples in the hardware FIFO and the number of samples held in the available multiplexor buffers. Therefore, in the above example of running all 32 channels at the maximum sampling rate of 400,000 SPS, if two multiplexor buffers are already processed and the hardware FIFO is almost full, then the latency will be due to draining approximately 64+64+128 kilo-samples or approximately 20 milliseconds. This number could also be larger depending on how many samples have been processed in the third multiplexor buffer.

At the risk of not getting under-runs, if you were to reduce the number of DMA buffers to 2 and the number of multiplexor buffers also two 2, you could theoretically reduce the latency drainage to approximately 32+128 kilo-samples or approximately 13 milliseconds. Once again, this number could also be larger depending on how many samples have been processed in the second multiplexor buffer.

If the users can reduce the number of active channels, then the chances of under-runs are reduced; however, it will also take longer to drain the FIFO as fewer channels will be draining simultaneously. In this case, the user could reduce the four tunable values accordingly. The user may experiment with the *aocc_stream* test located in the *…/test* directory using different parameter settings and observe the stability of the application *(no under-runs)* and the outputs of the wave to decide on suitable settings.

If the user is not concerned with seamless operation during change of frequency, phase, or wave, they can supply the *immediate* parameter in the change APIs with a non-zero value. The smaller the number, the shorter is the latency. In this case, all active waves are abruptly terminated and FIFO and buffers cleared and restarted with the new settings. All active waves will resume concurrently from their first sample in their original sample file unless a new phase change has been applied to any of them.

10. Creating Wave Files

Wave files are ASCII files used to import waveform data. The waveform data may be in one of two formats:

- Floating point format
- Hexadecimal format

The number of wave samples specified in the wave file may range from 2 to 524,288 (512k) and must be a power of two. This is a requirement for the software to provide a seamless frequency change. The total number of samples in each wave file represents a single period of a periodic waveform which will be continuously replicated until the user selects a new wave. The frequency at which this period is repeated is determined by the CCURAOCC card's sample frequency divided by the number of samples in the wave file. For example, if you were to create a single Sine wave containing 256 samples in a wave file and run at its maximum sample frequency of 400,000 samples per second, the frequency of the period (or the Sine wave) will be 400,000 ÷ 256, or 1562.5 Hz.

As the number of samples in a wave file increases, more memory and processing is required. When using 128K or larger wave files the additional processing overhead could result in sample under-runs (the on-board FIFO becomes empty) if multiple boards with several channels each are all active running at high sample rates. It is recommended to have samples per file limited to 64K samples or reduce the sampling rate when several boards with several channels are active at the same time.

If under-runs occur, the user has several options to rectify the problem:

- Reduce the sampling rate (0.2 to 400,000 SPS).
- Reduce the size of samples per buffer (i.e. the period of the wave).
- Reduce the number of channels being sampled.
- Reduce the number of boards being sampled.
- Use a faster multi-processor system with higher PCI bus bandwidth.
- Run the application on fully shielded processors so that they do not have interference from any other processes.

Users can include comment lines in the wave file by inserting a '**#**' or '*****' character in the first column of the wave file. The file format is specified by entering one of the two key words, *FORMAT_FLOAT* or *FORMAT_HEX*, somewhere at the beginning of the wave file, followed by the actual wave values in the selected format. There are no restrictions to the number of sample points that can be specified in a line. Floating point values can use epsilon 'e' format. Sample points can be separated by spaces or tabs. Hex values must be in the range 00000 to 3FFFF (18-bits).

The voltages specified (including any offset/gain calibration applied) must lie within the maximum voltage range of the channel, otherwise errors will be generated. The *FORMAT_HEX* wave file is a wave file that contains raw hex representation of voltages. The user needs to make sure that the hex representation corresponds to the correct channel format, i.e. CCURAOCC_CONVERTER_OFFSET_BINARY or CCURAOCC_CONVERTER_TWOS_COMPLEMENT; otherwise incorrect waves will be generated. Offset/gain calibration can be applied to either wave format.

Sample wave files are located in the *…./test/WaveFiles* directory.

Examples of wave files:

```
################## SAMPLES=256 [TWOs Complement] ##################
#  Sine Wave: 10.00 volts (7.07 RMS) (p-p=20.00 volts)
#  Wave Frequency: 1562.50000 Hz - 256 samples

FORMAT_HEX

00000 00c90 0191f 025aa 0322f 03eac 04b20 05788 063e2 0702e 07c67 0888e 094a0 0a09a 0ac7c 0b844
0c3ef 0cf7b 0dae8 0e633 0f15a 0fc5d 10738 111eb 11c73 126d0 130ff 13aff 144cf 14e6c 157d6 1610b
16a09 172d0 17b5d 183b0 18bc8 193a2 19b3e 1a29a 1a9b6 1b090 1b728 1bd7c 1c38b 1c954 1ced7 1d413
1d906 1ddb1 1e212 1e628 1e9f4 1ed74 1f0a7 1f38f 1f629 1f876 1fa75 1fc26 1fd88 1fe9c 1ff62 1ffd8
```

```
1ffff 1ffd8 1ff62 1fe9c 1fd88 1fc26 1fa75 1f876 1f629 1f38f 1f0a7 1ed74 1e9f4 1e628 1e212 1ddb1
1d906 1d413 1ced7 1c954 1c38b 1bd7c 1b728 1b090 1a9b6 1a29a 19b3e 193a2 18bc8 183b0 17b5d 172d0
16a09 1610b 157d6 14e6c 144cf 13aff 130ff 126d0 11c73 111eb 10738 0fc5d 0f15a 0e633 0dae8 0cf7b
0c3ef 0b844 0ac7c 0a09a 094a0 0888e 07c67 0702e 063e2 05788 04b20 03eac 0322f 025aa 0191f 00c90
00000 3f36f 3e6e0 3da55 3cdd0 3c153 3b4df 3a877 39c1d 38fd1 38398 37771 36b5f 35f65 35383 347bb
33c10 33084 32517 319cc 30ea5 303a2 2f8c7 2ee14 2e38c 2d92f 2cf00 2c500 2bb30 2b193 2a829 29ef4
295f6 28d2f 284a2 27c4f 27437 26c5d 264c1 25d65 25649 24f6f 248d7 24283 23c74 236ab 23128 22bec
226f9 2224e 21ded 219d7 2160b 2128b 20f58 20c70 209d6 20789 2058a 203d9 20277 20163 2009d 20027
20000 20027 2009d 20163 20277 203d9 2058a 20789 209d6 20c70 20f58 2128b 2160b 219d7 21ded 2224e
226f9 22bec 23128 236ab 23c74 24283 248d7 24f6f 25649 25d65 264c1 26c5d 27437 27c4f 284a2 28d2f
295f6 29ef4 2a829 2b193 2bb30 2c500 2cf00 2d92f 2e38c 2ee14 2f8c7 303a2 30ea5 319cc 32517 33084
33c10 347bb 35383 35f65 36b5f 37771 38398 38fd1 39c1d 3a877 3b4df 3c153 3cdd0 3da55 3e6e0 3f36f


################## SAMPLES=256 [Offset Binary] ##################
#  Sine Wave: 10.00 volts (7.07 RMS) (p-p=20.00 volts)
#  Wave Frequency: 1562.50000 Hz - 256 samples

FORMAT_HEX

20000 20c90 2191f 225aa 2322f 23eac 24b20 25788 263e2 2702e 27c67 2888e 294a0 2a09a 2ac7c 2b844
2c3ef 2cf7b 2dae8 2e633 2f15a 2fc5d 30738 311eb 31c73 326d0 330ff 33aff 344cf 34e6c 357d6 3610b
36a09 372d0 37b5d 383b0 38bc8 393a2 39b3e 3a29a 3a9b6 3b090 3b728 3bd7c 3c38b 3c954 3ced7 3d413
3d906 3ddb1 3e212 3e628 3e9f4 3ed74 3f0a7 3f38f 3f629 3f876 3fa75 3fc26 3fd88 3fe9c 3ff62 3ffd8
3ffff 3ffd8 3ff62 3fe9c 3fd88 3fc26 3fa75 3f876 3f629 3f38f 3f0a7 3ed74 3e9f4 3e628 3e212 3ddb1
3d906 3d413 3ced7 3c954 3c38b 3bd7c 3b728 3b090 3a9b6 3a29a 39b3e 393a2 38bc8 383b0 37b5d 372d0
36a09 3610b 357d6 34e6c 344cf 33aff 330ff 326d0 31c73 311eb 30738 2fc5d 2f15a 2e633 2dae8 2cf7b
2c3ef 2b844 2ac7c 2a09a 294a0 2888e 27c67 2702e 263e2 25788 24b20 23eac 2322f 225aa 2191f 20c90
20000 1f36f 1e6e0 1da55 1cdd0 1c153 1b4df 1a877 19c1d 18fd1 18398 17771 16b5f 15f65 15383 147bb
13c10 13084 12517 119cc 10ea5 103a2 0f8c7 0ee14 0e38c 0d92f 0cf00 0c500 0bb30 0b193 0a829 09ef4
095f6 08d2f 084a2 07c4f 07437 06c5d 064c1 05d65 05649 04f6f 048d7 04283 03c74 036ab 03128 02bec
026f9 0224e 01ded 019d7 0160b 0128b 00f58 00c70 009d6 00789 0058a 003d9 00277 00163 0009d 00027
00000 00027 0009d 00163 00277 003d9 0058a 00789 009d6 00c70 00f58 0128b 0160b 019d7 01ded 0224e
026f9 02bec 03128 036ab 03c74 04283 048d7 04f6f 05649 05d65 064c1 06c5d 07437 07c4f 084a2 08d2f
095f6 09ef4 0a829 0b193 0bb30 0c500 0cf00 0d92f 0e38c 0ee14 0f8c7 103a2 10ea5 119cc 12517 13084
13c10 147bb 15383 15f65 16b5f 17771 18398 18fd1 19c1d 1a877 1b4df 1c153 1cdd0 1da55 1e6e0 1f36f


################## SAMPLES=256 ##################
#  Sine Wave: 10.00 volts (7.07 RMS) (p-p=20.00 volts)
#  Wave Frequency: 1562.50000 Hz - 256 samples

FORMAT_FLOAT

 0.000000 0.245412 0.490677 0.735646  0.980171 1.224107 1.467305 1.709619 1.950903 2.191012
 2.429802 2.667128 2.902847 3.136817  3.368899 3.598950 3.826834 4.052413 4.275551 4.496113
 4.713967 4.928982 5.141027 5.349976  5.555702 5.758082 5.956993 6.152316 6.343933 6.531728
 6.715590 6.895405 7.071068 7.242471  7.409511 7.572088 7.730105 7.883464 8.032075 8.175848
 8.314696 8.448536 8.577286 8.700870  8.819213 8.932243 9.039893 9.142098 9.238795 9.329928
 9.415441 9.495282 9.569403 9.637761  9.700313 9.757021 9.807853 9.852776 9.891765 9.924795
 9.951847 9.972905 9.987955 9.996988 10.000000 9.996988 9.987955 9.972905 9.951847 9.924795
 9.891765 9.852776 9.807853 9.757021  9.700313 9.637761 9.569403 9.495282 9.415441 9.329928
 9.238795 9.142098 9.039893 8.932243  8.819213 8.700870 8.577286 8.448536 8.314696 8.175848
 8.032075 7.883464 7.730105 7.572088  7.409511 7.242471 7.071068 6.895405 6.715590 6.531728
 6.343933 6.152316 5.956993 5.758082  5.555702 5.349976 5.141027 4.928982 4.713967 4.496113
 4.275551 4.052413 3.826834 3.598950  3.368899 3.136817 2.902847 2.667128 2.429802 2.191012
 1.950903 1.709619 1.467305 1.224107  0.980171 0.735646 0.490677 0.245412 0.000000

 -0.245412 -0.490677 -0.735646 -0.980171 -1.224107 -1.467305 -1.709619 -1.950903 -2.191012
 -2.429802 -2.667128 -2.902847 -3.136817 -3.368899 -3.598950 -3.826834 -4.052413 -4.275551
 -4.496113 -4.713967 -4.928982 -5.141027 -5.349976 -5.555702 -5.758082 -5.956993 -6.152316
 -6.343933 -6.531728 -6.715590 -6.895405 -7.071068 -7.242471 -7.409511 -7.572088 -7.730105
 -7.883464 -8.032075 -8.175848 -8.314696 -8.448536 -8.577286 -8.700870 -8.819213 -8.932243
 -9.039893 -9.142098 -9.238795 -9.329928 -9.415441 -9.495282 -9.569403 -9.637761 -9.700313
 -9.757021 -9.807853 -9.852776 -9.891765 -9.924795 -9.951847 -9.972905 -9.987955 -9.996988
-10.000000 -9.996988 -9.987955 -9.972905 -9.951847 -9.924795 -9.891765 -9.852776 -9.807853
 -9.757021 -9.700313 -9.637761 -9.569403 -9.495282 -9.415441 -9.329928 -9.238795 -9.142098
 -9.039893 -8.932243 -8.819213 -8.700870 -8.577286 -8.448536 -8.314696 -8.175848 -8.032075
 -7.883464 -7.730105 -7.572088 -7.409511 -7.242471 -7.071068 -6.895405 -6.715590 -6.531728
 -6.343933 -6.152316 -5.956993 -5.758082 -5.555702 -5.349976 -5.141027 -4.928982 -4.713967
 -4.496113 -4.275551 -4.052413 -3.826834 -3.598950 -3.368899 -3.136817 -2.902847 -2.667128
 -2.429802 -2.191012 -1.950903 -1.709619 -1.467305 -1.224107 -0.980171 -0.735646 -0.490677
 -0.245412
```

11. Creating a Calibration File

The calibration file can be created on a per board basis. Its name is *CCURAOCC.<bno + 1>* where 'bno' represents the board number that was used to open the device. Comments can be inserted in the file by entering a '#' or '*' in the first column of the file. This file contains the following three tokens: channel, offset and gain. Users can specify calibration values for each of the 16 channels as follows:

channel=<channel number>, offset=<channel offset>, gain=<channel gain>

e.g. channel=0, offset=0.000123, gain=1.00002
channel=25, offset=0.003123, gain=1.00035

Refer to the example file *CCURAOCC.1.ex* in the *…./test* directory for further information.

No calibration will occur if the device is opened with the opts.Calibrate option set to AO_RAW_DATA_MODE. For the software to look at the calibration files, this option must be set to AO_USE_OFFSET_GAIN. Additionally, if user has supplied an *amplitude* and/or a *bias* to a wave via the *AOCC_ActionChangeChanWave()* or *AOCC_ActionLinkWave()* calls, the *amplitude* will be multiplied by the calibration *gain* while the *bias* will be added to the calibration *offset.* The user will need to make sure that the resulting waves range does not exceed the maximum voltage range for the board, otherwise clipping will occur and warning messages will be generated.

The following is the search for CAL files:

1- if opts.CalDir is specified by user in the `AOCC_OpenAOStream()` call, find the calibration files in that directory, and if not found, ignore and go to next search
2- search for calibration files in current directory and if not found ignore and go to next search.
3- search for the calibration files in the /usr/lib/config directory and if not found, ignore it.

The *CCURAOCC.1*.ex example wave file in the …/test directory.

```
################################################################################
#                                                                              #
# CCURAOCC Calibration                                                         #
# ====================                                                         #
# This file can specify the gain and offset for each individual channel.       #
# If a channel is not specified, a gain=1.0 and offset=0.0 is the default.     #
# Currently, this file is only looked at by the aocc_stream API. It can        #
# be located in the current directory, the /usr/lib/config/CAL directory       #
# or a user specified directory.                                               #
#                                                                              #
# The search order during opening the device is as follows:                    #
#    1) user specified directory                                               #
#    2) current directory where the application is run                          #
#    3) /usr/lib/config/CAL directory                                          #
#                                                                              #
# The API will not fail if the calibration file is not found. It will simply   #
# use the defaults.                                                            #
#                                                                              #
# The format of the configuration file must be CCURAOCC.<device_no + 1>        #
# where <device_no> is the device number 0..10 is the device being opened.     #
#                                                                              #
# Note that the number following the CCURAOCC string is device_number          #
# PLUS 1 and NOT simply the device_number.                                     #
#                                                                              #
################################################################################

channel=0,   offset=0.000000,  gain=1.00000
channel=1,   offset=0.000000,  gain=1.00000
channel=2,   offset=0.000000,  gain=1.00000
channel=3,   offset=0.000000,  gain=1.00000
channel=4,   offset=0.000000,  gain=1.00000
channel=5,   offset=0.000000,  gain=1.00000
channel=6,   offset=0.000000,  gain=1.00000
channel=7,   offset=0.000000,  gain=1.00000
```

```
channel=8,   offset=0.000000,  gain=1.00000
channel=9,   offset=0.000000,  gain=1.00000
channel=10,  offset=0.000000,  gain=1.00000
channel=11,  offset=0.000000,  gain=1.00000
channel=12,  offset=0.000000,  gain=1.00000
channel=13,  offset=0.000000,  gain=1.00000
channel=14,  offset=0.000000,  gain=1.00000
channel=15,  offset=0.000000,  gain=1.00000
channel=16,  offset=0.000000,  gain=1.00000
channel=17,  offset=0.000000,  gain=1.00000
channel=18,  offset=0.000000,  gain=1.00000
channel=19,  offset=0.000000,  gain=1.00000
channel=20,  offset=0.000000,  gain=1.00000
channel=21,  offset=0.000000,  gain=1.00000
channel=22,  offset=0.000000,  gain=1.00000
channel=23,  offset=0.000000,  gain=1.00000
channel=24,  offset=0.000000,  gain=1.00000
channel=25,  offset=0.000000,  gain=1.00000
channel=26,  offset=0.000000,  gain=1.00000
channel=27,  offset=0.000000,  gain=1.00000
channel=28,  offset=0.000000,  gain=1.00000
channel=29,  offset=0.000000,  gain=1.00000
channel=30,  offset=0.000000,  gain=1.00000
channel=31,  offset=0.000000,  gain=1.00000
```

This page intentionally left blank