Technical Guide
CCRTNGFC (WC-CP-FIO2)

PCIe Programmable Multi-Function I/O Card

Driver	ccrtngfc (WC-CP-FIO2)	
Platform	RedHawk Linux® *(CentOS/Rocky/RHEL & Ubuntu)*, Native Ubuntu® and Native Red Hat Enterprise Linux®[1]	
Vendor	Concurrent Real-Time	
Hardware	PCIe Programmable Multi-Function Card (CP-FPGA-4 & 5)	
Author	Darius Dubash	
Date	August 16th, 2024	Rev 2024.1

[1] All trademarks are the property of their respective owners

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 Next Generation FPGA card *(NGFC)*. 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:

- The NGFC Mother Board
  - TTY Digital Input/Output (DIO)
  - LVDS Input/Output (LIO)
  - Clocks
  - Modular Scatter-Gather DMA Engines
- Daughter Card: Analog to Digital (ADC) conversion
  - Calibration
  - Modular Scatter-Gather DMA support for ADC operation
- Daughter Card: Digital to Analog (DAC) conversion
  - Calibration
  - Modular Scatter-Gather DMA support for DAC operation

2. The NGFC Mother Board

The Next Generation FPGA card is basically a Gen-2 x4 high speed full size PCIe card that contains three separate I/O interfaces.

1) Digital Input/Output (DIO)
2) LVDS Input/Output (LIO)
3) Two positions for FMC style Daughter Cards.

Additionally, the mother board contains on-board Clocks and Modular Scatter-Gather DMA engines.

Currently the FMC slot supports the new Analog Daughter Card which supports both Analog to Digital and Digital to Analog signals.

In this document, the following terms are used for digital input/output functionality.

1) DIO – TTY Digital Input/Output
2) LIO – LVDS Input/Output
3) LDIO – Refers to either LIO *or* DIO context

2.1 TTY Digital Input/Output (DIO – Module 0)

The card supports 32-channels *(or lines)* of high speed TTL (3.3V/5V) (DIO) digital inputs and outputs. All access to this on-board DIO is via the API calls that specify the *CCRTNGFC_MAIN_DIO_MODULE_0 (or CCRTNGFC_LDIO_MODULE_0)* module as one of its argument.

Prior to performing any DIO operation, it needs to be activated with the *ccrtNGFC_LDIO_Activate()* API call. Without this activation, all other DIO calls will fail.

The direction call *ccrtNGFC_DIO_Set_Ports_Direction()* can be used to select the direction of a set of DIO ports. The channels or lines are grouped into ports where one channel *(or line)* is assigned to a single port.

The DIO can operate in either the *normal* DIO mode or the *custom* mode depending on whether the firmware loaded on the FPGA is a multi-function firmware or custom firmware. The *ccrtNGFC_LDIO_Set_Mode()* call is used to select the mode, which should match the type of firmware loaded, otherwise results will be unpredictable. For the rest of this discussion, we will be concentrating on the *normal* DIO mode.

The DIO also provides a capability to detect a change-of-state on any input line with the generation of an interrupt.

Note! This DIO interface is located on the mother board and is not supported on any currently available daughter cards.

2.1.1 TTY Digital Input

User can program 1 to 32 ports as *inputs* with the help of the *ccrtNGFC_DIO_Set_Ports_Direction()* call. A read issued to the lines associated with the input ports using the *ccrtNGFC_LDIO_Read_Input_Channel_Registers()* call will return the external digital signal connected to these lines. If this call is used to read ports programmed as *outputs*, then what is returned to the user is the output signals sent by the card to the external lines. In this way, a user can effectively perform an internal loopback of output lines.

The user has two modes of operation for reading the input channels:

- Continuous
- Snapshot *(Simultaneously)*

2.1.1.1 TTY Digital Continuous Input

This is the normal mode of operation where the user receives *asynchronously* the current state of each channel for every read. It is therefore possible that during the single read *ccrtNGFC_LDIO_Read_Input_Channel_Registers()* call, channels on different LDIO modules could change their current state asynchronously, thus not reflecting the *simultaneous* state of *all* the DIO or LIO channels between the various LDIO modules.

If this is the desired mode of operation, the user needs to first issue the *ccrtNGFC_LDIO_Set_Input_Snapshot()* call with the *CCRTNGFC_LDIO_INPUT_OPERATION_CONTINUOUS* option. This can be followed by multiple input channel reads with the *ccrtNGFC_LDIO_Read_Input_Channel_Registers()* call and the *ldio_snapshot* argument set to the *CCRTNGFC_LDIO_INPUT_OPERATION_DO_NOT_CHANGE* option.

If performance is not an issue, the user can skip the initial *ccrtNGFC_LDIO_Set_Input_Snapshot()* call and simply perform the input channel reads with the *CCRTNGFC_LDIO_INPUT_OPERATION_CONTINUOUS* option.

2.1.1.2 TTY Digital Snapshot Input

This mode of operation allows the user to receive all selected channels current state *simultaneously* for every read, i.e. takes a *snapshot* of the selected channels across all the LDIO modules.

If this is the desired mode of operation, the user can simply use the *ccrtNGFC_LDIO_Read_Input_Channel_Registers()* call with the *ldio_snapshot* argument set to the *CCRTNGFC_LDIO_INPUT_OPERATION_SNAPSHOT*. In this case, there is no need to issue the initial *ccrtNGFC_LDIO_Set_Input_Snapshot()* call.

Note: **As long the board is operating in the *snapshot* mode, the hardware will reflect the *simultaneous* state of all the input channels that were *last* snapshot for all the modules, i.e. the most recent hardware states will not be reflected until another *snapshot* was issued.**

2.1.1.3 TTY Digital Change-Of-State Input

The card provides capability to detect when a digital input line changes state. Detection can be for either the rising edge, falling edge or level detection. Level detection is when either rising or falling edge for a channel changes. In order to detect a change of state for a set of channels, the user will need to enable COS detection for the selected channels with the help of the *ccrtNGFC_LDIO_Set_COS_Channels_Enable()* API. Additionally the *ccrtNGFC_LDIO_Set_COS_Channels_Edge_Sense()* and the *ccrtNGFC_LDIO_Set_COS_Channels_Mode()* APIs are to be used to select what type of detection is to be performed on the channel.

The user will also need to create an interrupt handler with the help of the *ccrtNGFC_Create_UserLDioCosInterruptHandler()* API. This interrupt handler will be awoken every time a change of state interrupt has occurred for the selected channels. Useful information will be provided to assist the user in determining the cause of the interrupt. User needs to ensure that the duration of processing the interrupt in the user interrupt handler should be kept to a minimal; otherwise, there is a possibility of missing a change of state detection while it is in the routine.

Proper shielding and priority of both the application and driver needs to be conducted to ensure that no change of state is lost (overflow condition) or a user interrupt is missed. Redhawk provides the ability to shield and run applications at high priority. For example, to run the change-of-state test *ccrtngfc_ldio_intr* that is supplied with this driver, you can follow similar steps for your system:

```
# === as root ===
# Connect 17 KHz square wave to DIO (M0) channel 0 with a voltage of 0 to +4 volts and load of High-Z
# shield –a 2, 4-5                  (shield processors 2, 4 and 5)
# ./ccrtngfc_smp_affinity -c4       (force driver to CPU 2)
# run –b4-5 ./ccrtngfc_ldio_intr -M0   (run DIO test on CPU 4 & 5)
```

2.1.2 TTY Digital Output

User can program 1 to 32 ports as *outputs* with the help of the *ccrtNGFC_DIO_Set_Ports_Direction()* call. A write issued to the *output* registers with the *ccrtNGFC_LDIO_Write_Output_Channel_Registers()* call will cause the *output* registers to be written to. Those ports that have their direction as *outputs* will result in the digital signals being routed to the external lines. No routing of digital signals to external lines will occur for those lines whose ports have been configured as *inputs*. Those output channels that were written to ports that were configured as *inputs* will not output their digital signals to the external lines until the port's directions are switched to *outputs*. At any time, the users can read back the *output* registers that were last written to with the *ccrtNGFC_LDIO_Read_Output_Channel_Registers()* call.

The user has two modes of operation for writing the output channels:

- Continuous
- Synchronous *(Simultaneously)*

2.1.2.1 TTY Digital Continuous Output

This is the normal mode of operation where the asynchronous writes to the *output* registers will immediately appear on the external output lines. It is therefore possible that during the single write *ccrtNGFC_LDIO_Write_Output_Channel_Registers()* call, *simultaneous* output of channels on different LDIO modules would not occur.

If this is the desired mode of operation, the user needs to first issue the *ccrtNGFC_LDIO_Set_Output_Sync()* call with the *CCRTNGFC_LDIO_OUTPUT_OPERATION_CONTINUOUS* option. This can be followed by multiple output channel writes with the *ccrtNGFC_LDIO_Write_Output_Channel_Registers()* call and the *ldio_sync* argument set to the *CCRTNGFC_LDIO_OUTPUT_OPERATION_NOT_CHANGE* option.

If performance is not an issue, the user can skip the initial *ccrtNGFC_LDIO_Set_Output_Sync()* call and simply perform the output channel writes with the *CCRTNGFC_LDIO_OUTPUT_OPERATION_CONTINUOUS* option.

2.1.2.2 TTY Digital Synchronous Output

This mode of operation allows the user to write to all the selected channels and output them simultaneously i.e. *synchronize* the output channels across all the LDIO modules.

If this is the desired mode of operation, the user can simply use the *ccrtNGFC_LDIO_Write_Output_Channel_Registers()* call and the *ldio_sync* argument set to the

CCRTNGFC_LDIO_OUTPUT_OPERATION_SYNC option. In this case, there is no need to issue the initial *ccrtNGFC_LDIO_Set_Output_Sync()* call.

Note: As long the board is operating in the *synchronous* mode, the hardware will reflect the state of the output registers after a synchronization of channels occur, i.e. change will occur on the output lines only after the writes to the output registers are followed by a synchronization of outputs.

2.2 LVDS Input/Output (LIO – Module 1)

The card supports 32-channels *(or lines)* of high speed Low Voltage Differential Signalling (LVDS) digital inputs and outputs. All access to this on-board LIO is via the API calls that specify the *CCRTNGFC_MAIN_LIO_MODULE_1 (or CCRTNGFC_LDIO_MODULE_1)* module as one of its argument.

Prior to performing any LIO operation, it needs to be activated with the *ccrtNGFC_LDIO_Activate()* API call. Without this activation, all other LIO calls will fail.

The direction call *ccrtNGFC_LIO_Set_Ports_Direction()* can be used to select the direction of a set of LIO ports. The channels or lines are grouped into ports where *four* channels *(or lines)* are assigned to a single port.

The LIO can operate in either the *normal* LIO mode or the *custom* mode depending on whether the firmware loaded on the FPGA is a multi-function firmware or custom firmware. The *ccrtNGFC_LDIO_Set_Mode()* call is used to select the mode, which should match the type of firmware loaded, otherwise results will be unpredictable. For the rest of this discussion, we will be concentrating on the *normal* LIO mode.

The LIO also provides a capability to detect a change-of-state on any input line with the generation of an interrupt.

Note! This LVDS interface is located on the mother board and is not supported on any currently available daughter cards.

2.2.1 LVDS Input

User can program 1 to 8 ports as *inputs* with the help of the *ccrtNGFC_LIO_Set_Ports_Direction()* call. The channels or lines are grouped into ports where *four* channels *(or lines)* are assigned to a single port.

A read issued to the lines associated with the input ports using the *ccrtNGFC_LDIO_Read_Input_Channel_Registers()* call will return the external digital signal connected to these lines.

Unlike the DIO modules where the *ccrtNGFC_LDIO_Read_Input_Channel_Registers()* API returns the output signals sent by the card to the external lines if the ports directions are *outputs*, the LIO module behaves differently.

To obtain the same functionality as the DIO, you will need to set the Test Mode *CCRTNGFC_LIO_TEST_MODE* along with a SINGLE *port* to an *output* direction and read the *four* channels associated with the board. In order to perform this function, a useful API *ccrtNGFC_LIO_Read_Output_Loopbacked_Channels()* has been supplied.

The user has two modes of operation for reading the input channels:

ccrtNGFC_LIO_Read_Output_Loopbacked_Channels()

- Continuous
- Snapshot *(Simultaneously)*

2.2.1.1 LVDS Continuous Input

This is the normal mode of operation where the user receives *asynchronously* the current state of each channel for every read. It is therefore possible that during the single read *ccrtNGFC_LDIO_Read_Input_Channel_Registers()* call, channels on different LDIO modules could change their current state asynchronously, thus not reflecting the *simultaneous* state of *all* the DIO or LIO channels between the various LDIO modules.

If this is the desired mode of operation, the user needs to first issue the *ccrtNGFC_LDIO_Set_Input_Snapshot()* call with the *CCRTNGFC_LDIO_INPUT_OPERATION_CONTINUOUS* option. This can be followed by multiple input channel reads with the *ccrtNGFC_LDIO_Read_Input_Channel_Registers()* call and the *ldio_snapshot* argument set to the *CCRTNGFC_LDIO_INPUT_OPERATION_DO_NOT_CHANGE* option.

If performance is not an issue, the user can skip the initial *ccrtNGFC_LDIO_Set_Input_Snapshot()* call and simply perform the input channel reads with the *CCRTNGFC_LDIO_INPUT_OPERATION_CONTINUOUS* option.

2.2.1.2 LVDS Snapshot Input

This mode of operation allows the user to receive all selected channels current state *simultaneously* for every read, i.e. takes a *snapshot* of the selected channels across all the LDIO modules.

If this is the desired mode of operation, the user can simply use the *ccrtNGFC_LDIO_Read_Input_Channel_Registers()* call with the *ldio_snapshot* argument set to the *CCRTNGFC_LDIO_INPUT_OPERATION_SNAPSHOT*. In this case, there is no need to issue the initial *ccrtNGFC_LDIO_Set_Input_Snapshot()* call.

Note: **As long the board is operating in the *snapshot* mode, the hardware will reflect the *simultaneous* state of all the input channels that were *last* snapshot for all the modules, i.e. the most recent hardware states will not be reflected until another *snapshot* was issued.**

2.2.1.3 LVDS Change-Of-State Input

The card provides capability to detect when a digital input line changes state. Detection can be for either the rising edge, falling edge or level detection. Level detection is when either rising or falling edge for a channel changes. In order to detect a change of state for a set of channels, the user will need to enable COS detection for the selected channels with the help of the *ccrtNGFC_LDIO_Set_COS_Channels_Enable()* API. Additionally the *ccrtNGFC_LDIO_Set_COS_Channels_Edge_Sense()* and the *ccrtNGFC_LDIO_Set_COS_Channels_Mode()* APIs are to be used to select what type of detection is to be performed on the channel.

The user will also need to create an interrupt handler with the help of the *ccrtNGFC_Create_UserLDioCosInterruptHandler()* API. This interrupt handler will be awoken every time a change of state interrupt has occurred for the selected channels. Useful information will be provided to assist the user in determining the cause of the interrupt. User needs to ensure that the duration of processing the interrupt in the user interrupt handler should be kept to a minimal; otherwise, there is a possibility of missing a change of state detection while it is in the routine.

Proper shielding and priority of both the application and driver needs to be conducted to ensure that no change of state is lost (overflow condition) or a user interrupt is missed. Redhawk provides the ability to shield and run applications at high priority. For example, to run the change-of-state test *ccrtngfc_ldio_intr* that is supplied with this driver, you can follow similar steps for your system:

```
# === as root ===
# Connect 15 KHz square wave to LIO (M1) channel 3 with a voltage of +/- 200 milli-volts and load of 50 Ohms
# shield –a 2, 4-5                  (shield processors 2, 4 and 5)
# ./ccrtngfc_smp_affinity -c4       (force driver to CPU 2)
# run –b4-5 ./ccrtngfc_ldio_intr -M1   (run LIO test on CPU 4 & 5)
```

2.2.2 LVDS Output

User can program 1 to 8 ports as *outputs* with the help of the *ccrtNGFC_LIO_Set_Ports_Direction()* call. The channels or lines are grouped into ports where *four* channels *(or lines)* are assigned to a single port. A write issued to the *output* registers with the *ccrtNGFC_LDIO_Write_Output_Channel_Registers()* call will cause the *output* registers to be written to. Those ports that have their direction as *outputs* will result in the digital signals being routed to the external lines. No routing of digital signals to external lines will occur for those lines whose ports have been configured as *inputs*. Those output channels that were written to ports that were configured as *inputs* will not output their digital signals to the external lines until the port's directions are switched to *outputs*. At any time, the users can read back the *output* registers that were last written to with the *ccrtNGFC_LIO_Read_Output_Loopbacked_Channels()* call.

The user has two modes of operation for writing the output channels:

- Continuous
- Synchronous *(Simultaneously)*

2.2.2.1 LVDS Continuous Output

This is the normal mode of operation where the asynchronous writes to the *output* registers will immediately appear on the external output lines. It is therefore possible that during the single write *ccrtNGFC_LDIO_Write_Output_Channel_Registers()* call, *simultaneous* output of channels on different LDIO modules would not occur.

If this is the desired mode of operation, the user needs to first issue the *ccrtNGFC_LDIO_Set_Output_Sync()* call with the *CCRTNGFC_LDIO_OUTPUT_OPERATION_CONTINUOUS* option. This can be followed by multiple output channel writes with the *ccrtNGFC_LDIO_Write_Output_Channel_Registers()* call and the *ldio_sync* argument set to the *CCRTNGFC_LDIO_OUTPUT_OPERATION_NOT_CHANGE* option.

If performance is not an issue, the user can skip the initial *ccrtNGFC_LDIO_Set_Output_Sync()* call and simply perform the output channel writes with the *CCRTNGFC_LDIO_OUTPUT_OPERATION_CONTINUOUS* option.

2.2.2.2 LVDS Synchronous Output

This mode of operation allows the user to write to all the selected channels and output them simultaneously i.e. *synchronize* the output channels across all the LDIO modules.

If this is the desired mode of operation, the user can simply use the *ccrtNGFC_LDIO_Write_Output_Channel_Registers()* call and the *ldio_sync* argument set to the *CCRTNGFC_LDIO_OUTPUT_OPERATION_SYNC* option. In this case, there is no need to issue the initial *ccrtNGFC_LDIO_Set_Output_Sync()* call.

Note: As long the board is operating in the *synchronous* mode, the hardware will reflect the state of the output registers after a synchronization of channels occur, i.e. change will occur on the output lines only after the writes to the output registers are followed by a synchronization of outputs.

3. Daughter Card: Analog to Digital (ADC) Conversion

The ADC has 12 channels with 16-bit resolution, controlled by three ADC converters; each can be assigned one of six 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 3
ADC 1 -> Channel 4 to 7
```

```
ADC 2 -> Channel 8 to 11
```

Prior to performing any conversion, the user needs to *open(2)* the ADC module *(installed in one of the two FMC slots for the daughter cards)* with the *ccrtNGFC_DC_ADC_Open()* API. Once the ADC daughter card is successfully opened, it needs to be activated with the *ccrtNGFC_DC_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 12 channel registers 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.

- ADC Channel Registers *(asynchronous operation)*
- ADC FIFO *(synchronous operation)*

Prior to any data being collected, the user needs to configure each ADC in order to select one of 6 individual clocks *(0 to 5)* as the *input clock* with the *ccrtNGFC_DC_ADC_Set_CSR()* API. 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 three individual ADCs can also be programmed with data format of *offset binary* or *two's complement*, however, its inputs are always *bipolar* with a voltage range of 10 volts *(i.e. +/- 10V).*

3.1.1 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* 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_adc_ptr* can be obtained by using the *ccrtNGFC_DC_ADC_Get_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 *ccrtNGFC_DataToVolts()* library call. This call requires as an argument a pointer to the *ccrtngfc_volt_convert_t* structure that holds the current ADC configuration information.

b) Alternatively, the user can use the *ccrtNGFC_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 *ccrtNGFC_Transfer_Data()* library routine. The advantage of this call is that it allows the user to transfer the data via MsgDma or Programmed I/O. If this call is going to use MsgDma, then the received user buffer must be a buffer that can allow the board to perform MsgDma reads. This buffer can be obtained with the help of the *ccrtNGFC_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 *(CCRTNGFC_ADC_PIO_CHANNEL)* with the *ccrtNGFC_DC_ADC_Set_Driver_Read_Mode()* library call and then call the *ccrtNGFC_DC_ADC_Read()* routine to read the *raw* channel registers. ***At present, the driver***

does NOT support MsgDma transfers. In this case *(i.e. PIO mode)*, any buffer *(not necessarily a DMA capable one)* can be supplied to the *ccrtNGFC_DC_ADC_Read()* call.

e) Another approach is for the user to make use of the *ccrtNGFC_DC_ADC_Read_Channels()* library call. It 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 *ccrtNGFC_DC_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 *ccrtNGFC_DC_ADC_Get_CSR()* first and then supplying the current configuration to the *adc_csr* argument in the following *ccrtNGFC_DC_ADC_Read_Channels()* calls, with the assumption that the ADC configuration is not going to change for the duration of the reads.

f) Finally, to read the channels in the fastest manner possible, the user can make use of the MsgDma engines supplied with the card. In this case, the user will first need to acquire one of 6 available MsgDma engines via the *ccrtNGFC_MsgDma_Seize()* call.

Next, they will need to configure the MsgDma channel with the *ccrtNGFC_DC_ADC_MsgDma_Configure_Channel()* call, passing the MsgDma engine as one of the arguments to the call. Users also need to specify the start channel number *StartChannelNumber* and end channel number *EndChannelNumber* to this call. If the user plans to use one of the MsgDma engines 0 through 3, they need to ensure that the start channel number is a multiple of 4 and that the number of channels being transferred are also a multiple of 4. If that is not the case, then the user will need to use the slightly slower MsgDma engines 4 or 5 where there are no alignment restrictions.

Once the MsgDma is successfully configured in the above step, the user will then use the *ccrtNGFC_DC_ADC_MsgDma_Fire_Channel()* call supplying the same seized MsgDma engine. This call is then repeated for each new transfer of channel registers to the user supplied *PciDmaMemory* buffer, which is also one of the arguments in the above *ccrtNGFC_DC_ADC_MsgDma_Configure_Channel()* call.

Once the user is done with the transfer, they can release the MsgDma engine with the *ccrtNGFC_MsgDma_Release()* call.

3.1.2 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 *ccrtNGFC_DC_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 *ccrtNGFC_DC_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* 16-bit sample data, the *ADC FIFO* samples contain the *raw* 16-bit channel data along with the channel number in the most significant nibble 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.

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 *ccrtNGFC_DC_ADC_Reset_Fifo()* call.

Recommended method of data collection is to program the clocks and let them settle. Then disable them with the *ccrtNGFC_Clock_Set_Generator_CSR()* call. Next reset the *ADC FIFO* with the *ccrtNGFC_DC_ADC_Reset_Fifo()* call. Finally, when ready to start data collection, enable the clocks with the *ccrtNGFC_Clock_Set_Generator_CSR()* call.

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_adc_fifo_ptr* can be obtained by using the *ccrtNGFC_DC_ADC_Get_Info()* call. Once the pointer is available, the channels can be accessed via this register.

   If the user wishes to determine the *floating point* voltages for the *raw* data, they can do so with the help of the *ccrtNGFC_DataToVolts()* library call. This call requires as an argument a pointer to the *ccrtngfc_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 *ccrtNGFC_Transfer_Data()* library routine. The advantage of this call is that it allows the user to transfer the data via MsgDma or Programmed I/O. If this call is going to use MsgDma, 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 *ccrtNGFC_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 *(CCRTNGFC_ADC_PIO_FIFO)* with the *ccrtNGFC_DC_ADC_Set_Driver_Read_Mode()* library call and then call the *ccrtNGFC_DC_ADC_Read()* routine to read the *raw* channel samples. ***At present, the driver does NOT support MsgDma transfers.*** In this case *(i.e. PIO mode)*, any buffer *(not necessarily a DMA capable one)* can be supplied to the *ccrtNGFC_DC_ADC_Read()* call.

d) Finally, to read the channels in the fastest manner possible, the user can make use of the MsgDma engines supplied with the card. In this case, the user will first need to acquire one of 6 available MsgDma engines via the *ccrtNGFC_MsgDma_Seize()* call.

   Next, they will need to configure the MsgDma Fifo with the *ccrtNGFC_DC_ADC_MsgDma_Configure_Fifo()* call, passing the MsgDma engine as one of the arguments to the call. Users also need to specify the number of samples *NumberOfSamples* as one of its arguments. If the user plans to use one of the MsgDma engines 0 through 3, they need to ensure that the number of samples specified is a multiple of 4. If that is not the case, then the user will need to use the slightly slower MsgDma engines 4 or 5 where there are no alignment restrictions.

   Once the MsgDma is successfully configured in the above step, the user will then use the *ccrtNGFC_DC_ADC_MsgDma_Fire_Fifo()* call supplying the same seized MsgDma engine. This call is then repeated for each new transfer of ADC FIFO to the user supplied *PciDmaMemory* buffer, which is also one of the arguments in the above *ccrtNGFC_DC_ADC_MsgDma_Configure_Fifo()* call.

   Once the user is done with the transfer, they can release the MsgDma engine with the *ccrtNGFC_MsgDma_Release()* call.

Note: The first four set of samples that are processed after a FIFO reset are discarded as the high speed ADC requires time to settle before valid data is presented to the user.

3.1.3 ADC Input Options

Each of the three ADC's has the option of selecting its inputs either from the external lines *(normal mode)* or from the *calibration bus* with the *ccrtNGFC_DC_ADC_Set_CSR()* call. If external lines are selected for an ADC, all 4 ADC channels will return the *raw* digital values for the 4 inputs lines. If calibration bus is selected, then the ADC can receive one of the following with the *ccrtNGFC_DC_ADC_Set_Calibration_CSR()* call:

- Calibration Ground
- Calibration Postive Reference Voltage *(+9.91 volts)*
- Calibration Negative Reference Voltage *(-9.91 volts)*
- Calibration 4 Volts Reference *(+4.030 volts)*
- Calibration Positive 10 Volts Reference *(+10.0 volts)*
- Calibration Negative 10 Volts Reference *(-10.0 volts)*
- One of 12 DAC channels as input

The calibration connections are used for calibrating the ADCs, while the DAC inputs can be used to loopback the DAC outputs for diagnostics. ***Note that all 4 ADC channels will display the same Calibration reference voltage or DAC channel, depending on the calibration bus selection.***

4. Daughter Card: Digital to Analog (DAC) Conversion

The DAC has 12 channels with 16-bit resolution, controlled by six DAC converters. It supports both single-ended and differential outputs. The outputs can be software configured as 12-channel single-ended outputs, 6-channel differential outputs, or a combination of single-ended and differential on a per-DAC granularity. Differential channels are identified as a pair of odd/even channels. Unlike the ADC converters where each converter can have its own clock, all six DAC converters can either be assigned for *software update* or a selection of one of six update clocks.

DAC to channel association is as follows:

```
DAC 0 -> Channel   0 to  1
DAC 1 -> Channel   2 to  3
DAC 2 -> Channel   4 to  5
DAC 3 -> Channel   6 to  7
DAC 4 -> Channel   8 to  9
DAC 5 -> Channel  10 to 11
```

Prior to performing any conversion, the user needs to *open(2)* the DAC module *(installed in one of the two FMC slots for the daughter cards)* with the *ccrtNGFC_DC_DAC_Open()* API. Once the DAC daughter card is successfully opened, it needs to be activated with the *ccrtNGFC_DC_DAC_Activate()* API call. Without this activation, all other DAC calls will fail.

There are two mechanisms implemented by the hardware to enable the user to generate analog signals. The DAC channels can be written to either 12 channel registers or a DAC FIFO that is 128K samples deep. Either of these approaches can be used to write digital samples to the channels.

- DAC Channel Registers *(asynchronous operation)*
- DAC FIFO *(synchronous operation)*

Prior to writing any samples, the user needs to configure the DACs in order to select *software update* or one of 6 individual clocks *(0 to 5)* as the *input clock*. Unlike the ADCs where the users can select a different clock for each of the two ADCs, all four DACs are controlled by a *single* source. This can be accomplished by using the *ccrtNGFC_DC_DAC_Set_Update_Source_Select()* routine. If the *update source* is a *clock*, then the onboard clock generator needs to be programmed to the user desired sample generation rate.

In addition to the above setup which affects all DACs, each of the six individual DACs can be programmed with data format of *offset binary* or *two's complement*, however, its outputs are always *bipolar* with a voltage range of 10 volts *(i.e. +/- 10V).*

Users can also program each individual DAC to operate in *Immediate* or *Synchronized* mode with the *ccrtNGFC_DC_DAC_Set_CSR()* call. Depending on the mode of operation, the hardware will determine when to output analog signals on the individual channels. Conceptually, immediate mode would cause analog signals to be output to the channel the moment the digital sample was written to the registers. In the case of *Synchronized* mode, all registers belonging to a particular DAC would be synchronized and output simultaneously by the hardware.

4.1.1 DAC Channel Registers

This mechanism allows the user to write *raw* digital values to any of the DAC channel registers. The hardware, in turn, outputs the converted analog signals according to the update *source selection* and *operational mode*.

Note: Make sure that you do not have any samples in the DAC FIFO with a clock running during writing to the DAC channel registers as the board will overwrite the channel registers with the FIFO samples. It is best to ensure that the FIFO is empty before commencing DAC channel register writes.

If the *operational mode* for any of the six DACs is *Immediate*, the analog outputs for the channels associated with the DAC will continuously output the last converted analog signal. The moment a write occurs on any channel register for the associated DAC, the hardware will convert the *raw* digital value to the new analog signal and output the new value on the corresponding channel. In this *operational mode*, the *update source* selection is ignored.

If the *operational mode* for any of the six DACs is *Synchronized*, all channels associated with the DAC will output in accordance with the *update source* selection. If *software update* mode is selected, then a write to any channel with bit 31 *(sync update flag)* set will cause all the DACs that have an *operational mode* set to *Synchronized* to convert and simultaneously output its corresponding channels. If instead, the *update source* is set to an active *clock*, then the hardware will convert to analog signals the *raw* digital values for *all* the DAC channels *(that have an operational mode of Synchronized)* and *simultaneously* update these channels on *every clock cycle.* The rate of update will be dependent on the clock rate of the clock assigned to the DAC *update source.* In this case, setting bit 31 *(sync update flag)* in the *raw* digital value for any channel will be ignored.

Note: If the user has operational mode for any DACs as *Synchronized* and the source selection set to *software update*, then no analog signal change will occur on the outputs until a channel is written with bit 31 *(sync update flag)* set. If, instead, the source selection for the DACs is a clock, analog signal change will only occur on the outputs if the clock associated with the DACs is programmed and running.

There are various methods available at the disposal of the user to output the contents of the 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_dac_ptr* can be obtained by using the *ccrtNGFC_DC_DAC_Get_Info()* call. Once the pointer is available, the channels can be accessed via the *DAC_Data[ ]* array.

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

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

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

d) Another approach is for the user to make use of the driver to write to the DAC channel registers. In this case, the user needs to first select the appropriate channel write mode operation *(CCRTNGFC_DAC_PIO_CHANNEL)* with the *ccrtNGFC_DC_DAC_Set_Driver_Write_Mode()* library call and then call the *ccrtNGFC_DC_DAC_Write()* routine to write to 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 *ccrtNGFC_DC_DAC_Write()* call.

e) Another approach is for the user to make use of the *ccrtNGFC_DC_DAC_Write_Channels()* library call. It not only allows the user to select individual channels via a channel mask, but also allows the user to supply *floating point* voltages and lets the call perform the necessary conversion to *raw* data prior to writing the channel registers.

The user has the option to supply a *NULL* pointer instead of the *dac_csr* argument. In this case the *ccrtNGFC_DC_DAC_Write_Channels()* call will internally extract the current hardware DAC configuration prior to computing the *raw* data. This would add considerable overhead to the call if it is being called multiple times. Alternatively, the user could first determine the current DAC configuration using the *ccrtNGFC_DC_DAC_Get_CSR()* first and then supplying the current configuration to the *dac_csr* argument in the following *ccrtNGFC_DC_DAC_Write_Channels()* calls, with the assumption that the DAC configuration is not going to change for the duration of the writes.

Additionally, this call always sets bit 31*(sync update flag)* in the *raw* data for the *last* channel. In this way, if the user had set the operational mode to *Synchronized* for any DACs, all channels for the DACs will be sent out simultaneously by the hardware. There is therefore no need for the user to set the last channel with bit 31 when using this call, in case they wanted outputs of channels to be synchronized.

f) Finally, to write the channels in the fastest manner possible, the user can make use of the MsgDma engines supplied with the card. In this case, the user will first need to acquire one of 6 available MsgDma engines via the *ccrtNGFC_MsgDma_Seize()* call

Next, they will need to configure the MsgDma channel with the *ccrtNGFC_DC_DAC_MsgDma_Configure_Channel()* call, passing the MsgDma engine as one of the arguments to the call. Users also need to specify the start channel number *StartChannelNumber* and end channel number *EndChannelNumber* to this call. If the user plans to use one of the MsgDma engines 0 through 3, they need to ensure that the start channel number is a multiple of 4 and that the number of channels being transferred are also a multiple of 4. If that is not the case, then the user will need to use the slightly slower MsgDma engines 4 or 5 where there are no alignment restrictions.

Once the MsgDma is successfully configured in the above step, the user will then use the *ccrtNGFC_DC_DAC_MsgDma_Fire_Channel()* call supplying the same seized MsgDma engine. This call is then repeated for each new transfer of channel registers to the user supplied *PciDmaMemory* buffer, which is also one of the arguments in the above *ccrtNGFC_DC_DAC_MsgDma_Configure_Channel()* call.

Once the user is done with the transfer, they can release the MsgDma engine with the *ccrtNGFC_MsgDma_Release()* call.

4.1.2 DAC FIFO

This mechanism allows the hardware to convert *raw* sample voltages placed in the DAC FIFO by the user and *synchronously* output analog signals on the selected DAC channels on every clock cycle. Once the clocks have started *(after programming the DACs and clocks)*, the board will continuously convert the *raw* sample voltages in the DAC FIFO for the selected DAC channels and output them at the programmed clock rate.

The user can select which channels are to be converted by the hardware and placed in the *DAC FIFO* with the channel selection mask supplied to the *ccrtNGFC_DC_DAC_Set_Fifo_Channel_Select()* call. Note that for differential channels, the odd channels will be masked out and outputs will only appear on even numbered channels. Synchronous output will occur for the set of selected DAC channels either sequentially or simultaneously based on the *update mode* selection of *Immediate* or *Synchronized*.

Care must be taken to ensure that the *DAC FIFO* does not get empty *(underflow)* or go beyond full *(overflow)*, otherwise synchronous signal generation will be compromised. If this occurs, the *DAC FIFO* should be reset with the *ccrtNGFC_DC_DAC_Reset_Fifo()* call to empty the *DAC FIFO* and resume from a known state. At any time, the *ccrtNGFC_DC_DAC_Get_Fifo_Info()* call can be invoked to determine the status of the *DAC FIFO*.

Unlike the samples in the *ADC FIFO* which contain the *raw* sample data and the associated channel number, the *DAC FIFO* samples do not contain the channel number. Once sampling has commenced, the hardware will map each sample in the *DAC FIFO* with the channel selection mask. In order to guarantee synchronization between the samples in the *DAC FIFO* and the channel selection mask, it is necessary to perform a *DAC FIFO* reset with the *ccrtNGFC_DC_DAC_Reset_Fifo()* call prior to commencing sample conversion. Additionally, the channel selection mask must not be changed while sampling, otherwise, unpredictable results will occur as the sample to channel association will no longer be valid.

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

Recommended method of data conversion is to program the clocks and let them settle. Then disable them with the *ccrtNGFC_Clock_Set_Generator_CSR()* call. Next select a set of channels whose samples are going to be placed in the *DAC FIFO* and then reset the *DAC FIFO* with the *ccrtNGFC_DC_DAC_Reset_Fifo()* call and start a few writing samples into it for selected channels *(priming the FIFO).* Finally, when ready to start conversion, enable the clocks with the *ccrtNGFC_Clock_Set_Generator_CSR()* call.

There are various methods available at the disposal of the user to write to the *DAC FIFO* so that the hardware can convert the samples to analog signals. 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_dac_fifo_ptr* can be obtained by using the *ccrtNGFC_DC_DAC_Get_Info()* call. Once the pointer is available, the channels can be accessed via this register.

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

b) Another method to transfer the contents of a consecutive set of *raw* values in a local buffer to the *DAC FIFO* is to use the *ccrtNGFC_Transfer_Data()* library routine. The advantage of this call is that it allows the user to transfer the data via MsgDma or Programmed I/O. If this call is going to use MsgDma, then the transmitting

user buffer must be a buffer that can allow the board to perform MsgDma writes. This buffer can be obtained with the help of the *ccrtNGFC_MMap_Physical_Memory()* library call.

c) Another approach is for the user to make use of the driver to write to the *DAC FIFO*. In this case, the user needs to first select the appropriate channel write mode operation *(CCRTNGFC_DAC_PIO_FIFO)* with the *ccrtNGFC_DC_DAC_Set_Driver_Write_Mode()* library call and then call the *ccrtNGFC_DC_DAC_Write()* routine to write *raw* data to the DAC FIFO. ***At present, the driver does NOT support MsgDma transfers.*** In this case *(i.e. PIO mode)*, any buffer *(not necessarily a DMA capable one)* can be supplied to the *ccrtNGFC_DC_DAC_Write()* call.

e) Finally, to write to the channels in the fastest manner possible, the user can make use of the MsgDma engines supplied with the card. In this case, the user will first need to acquire one of 6 available MsgDma engines via the *ccrtNGFC_MsgDma_Seize()* call.

Next, they will need to configure the MsgDma Fifo with the *ccrtNGFC_DC_DAC_MsgDma_Configure_Fifo()* call, passing the MsgDma engine as one of the arguments to the call. Users also need to specify the number of samples *NumberOfSamples* as one of its arguments. If the user plans to use one of the MsgDma engines 0 through 3, they need to ensure that the number of samples specified is a multiple of 4. If that is not the case, then the user will need to use the slightly slower MsgDma engines 4 or 5 where there are no alignment restrictions.

Once the MsgDma is successfully configured in the above step, the user will then use the *ccrtNGFC_DC_DAC_MsgDma_Fire_Fifo()* call supplying the same seized MsgDma engine. This call is then repeated for each new transfer from the user supplied *PciDmaMemory* buffer *(which is also one of the arguments in the above ccrtNGFC_DC_DAC_MsgDma_Configure_Fifo() call)* to the DAC FIFO.

Once the user is done with the transfer, they can release the MsgDma engine with the *ccrtNGFC_MsgDma_Release()* call.

5. 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. Modular Scatter-Gather DMA
3. Modular Scatter-Gather DMA Cloning

Of the three 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.

Modular Scatter-Gather DMA is a lot 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 Modular Scatter-Gather DMA operation commences, entire transfer occurs between the card and memory without CPU intervention. Since there is a finite setup time to initialize the Modular Scatter-Gather DMA, it is only useful for large transfers as the overhead of setup would offset any gains for smaller transfers. Additionally, the user has the ability to setup multiple DMA accesses with a single call. 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 Cloning is identical to the Modular Scatter-Gather DMA operation with the exception that once the Cloning operation has commenced, it keeps repeating the Modular Scatter-Gather DMA under hardware control until it is stopped.

There are *six* Modular Scatter-Gather DMA engines present on the mother board. They can be shared between various resources and multiple Modular Scatter-Gather DMA engines can be running concurrently. Before any

access can be granted to a resource for MsgDma operation, the engine needs to be seized via the *ccrtNGFC_MsgDma_Seize()* call. At this point, no other resource can access this engine until it is released by the *ccrtNGFC_MsgDma_Release()* call. When using MsgDma engines 0 through 3, the data supplied to the call must be quad-word aligned *(multiple of 16 bytes)* and the transfer size must also be a multiple of 4 words or 16 bytes. If the above restriction on alignment and size cannot be satisfied, then the user will have to use the MsgDma engines 4 or 5 which require only word *(i.e. 4 bytes)* alignment.

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

1. Programmed I/O

- ccrtNGFC_Fast_Memcpy()
- ccrtNGFC_Fast_Memcpy_Unlocked()
- ccrtNGFC_Fast_Memcpy_Unlocked_FIFO()
- ccrtNGFC_Transfer_Data()
- ccrtNGFC_Get_Mapped_Local_Ptr() // pointer to the card local memory - advanced users only
- ccrtNGFC_DC_ADC_Read() // for reading ADC FIFO or channels via driver
- ccrtNGFC_DC_ADC_Read_Channels() // for reading ADC channels
- ccrtNGFC_DC_ADC_Read_Channels_Calibration() // for reading ADC channel calibration values
- ccrtNGFC_DC_ADC_Get_Value() // to read specific values on the board registers
- ccrtNGFC_DC_ADC_Set_Value() // to write specific values to the board registers

- ccrtNGFC_DC_DAC_Read_Channels() // for reading DAC channels
- ccrtNGFC_DC_DAC_Read_Channels_Calibration() // for reading DAC channel calibration values
- ccrtNGFC_DC_DAC_ReadBack_Channels // for reading DAC readback channels
- ccrtNGFC_DC_DAC_Write() // for writing DAC FIFO and channels via driver
- ccrtNGFC_DC_DAC_Write_Channels() // for writing to DAC channels
- ccrtNGFC_DC_DAC_Write_Channels_Calibration() // for writing to DAC channel calibration
- ccrtNGFC_DC_DAC_Get_Value() // to read specific values on the board registers
- ccrtNGFC_DC_DAC_Set_Value() // to write specific values to the board registers

2. Modular Scatter-Gather DMA

- ccrtNGFC_Transfer_Data() // single MsgDma transfer

- ccrtNGFC_MsgDma_Seize() // single MsgDma Channel register transfer
  ccrtNGFC_DC_ADC_MsgDma_Configure_Channel()
  ccrtNGFC_DC_ADC_MsgDma_Fire_Channel() - repeat multiple times
  ccrtNGFC_MsgDma_Release()

- ccrtNGFC_MsgDma_Seize() // single MsgDma FIFO transfer
  ccrtNGFC_DC_ADC_MsgDma_Configure_Fifo()
  ccrtNGFC_DC_ADC_MsgDma_Fire_Fifo() - repeat multiple times
  ccrtNGFC_MsgDma_Release()

- ccrtNGFC_MsgDma_Seize() // single MsgDma Channel register transfer
  ccrtNGFC_DC_DAC_MsgDma_Configure_Channel()
  ccrtNGFC_DC_DAC_MsgDma_Fire_Channel() - repeat multiple times
  ccrtNGFC_MsgDma_Release()

- ccrtNGFC_MsgDma_Seize() // single MsgDma FIFO transfer
  ccrtNGFC_DC_DAC_MsgDma_Configure_Fifo()
  ccrtNGFC_DC_DAC_MsgDma_Fire_Fifo() - repeat multiple times
  ccrtNGFC_MsgDma_Release()

- ccrtNGFC_MsgDma_Seize() // multiple MsgDma transfer
  ccrtNGFC_MsgDma_Configure_Descriptor()

ccrtNGFC_MsgDma_Setup()
ccrtNGFC_MsgDma_Fire() - repeat multiple times
ccrtNGFC_MsgDma_Release()

- ccrtNGFC_MsgDma_Seize() // single MsgDma transfer
  ccrtNGFC_MsgDma_Configure_Single()
  ccrtNGFC_MsgDma_Fire_Single() - repeat multiple times
  ccrtNGFC_MsgDma_Release()

3. Modular Scatter-Gather DMA Cloning

- ccrtNGFC_MsgDma_Seize()
  ccrtNGFC_MsgDma_Configure_Descriptor()
  ccrtNGFC_MsgDma_Setup()
  ccrtNGFC_MsgDma_Clone() - start Cloning
  ccrtNGFC_MsgDma_Release()

6. Cloning *(CCRT US Patent US 11.281.584 B1, Inventor Darius Dubash)*

6.1 Scope

The CCRTNGFC allows Cloning of its entire local memory.

This card has six MsgDma engines and therefore up to a maximum of six Cloning or MsgDma operation can be active at a given time. Additionally, it is meaningless to perform Cloning on a FIFO region for two reasons. Firstly, each data in a FIFO is synchronous, however, the Cloned region is accessed asynchronously. Secondly, when the FIFO runs empty *(underflow)* or cannot accept more data *(overflow)* the results are unpredictable.

6.2 What is Cloning

It is a mechanism under hardware control, setup by the user to continuously reflect *(copy)* a section of physical or local memory on a card *(the source)* to another physical or local memory located on the same or another card *(the destination)*. Once Cloning has initiated, an image of *source* region appears on the *destination* region continuously at MsgDma transfer speed and the process cannot be throttled. The transfers are repeated continuously until the Cloning operation is stopped by the user. For example, the source can be the analog input registers on the card and its changing values can be reflected in the Cloned destination. If the Cloned destination is the analog output registers of the card, any change in values to the Cloned source will be reflected in the analog output registers.

Types of Cloning: There are two types of Cloning available:

- Basic
- Region Addressing

6.3 Basic Cloning

Basic Cloning is an option that can be purchased and involves a user having the ability to Clone a section of the board's local memory or a physical memory. All Cloning ***must*** reside within the user domain. In this case, board addresses are relative offsets within the local board memory area and not ABSOLUTE addresses *(as seen by the kernel)*. Additionally, any physical memory created must be one that the user previously created by the driver *(i.e. not an acquired physical memory from another user).*

With Basic Cloning the user can Clone:

- any MsgDma *(not FIFO)* local memory on the board as its source and a physical memory it created as its destination

- a physical memory it created as its source and any MsgDma *(not FIFO)* local memory on the board as its destination
- any MsgDma *(not FIFO)* local memory on the board as its source and another MsgDma *(not FIFO)* local memory on the *same* board as its destination
- a physical memory is created as its source and another physical memory is created as its destination *(as long as the user has created the physical memory and has full access to it).*

6.4 Region Addressing Cloning

This option expands the above Basic Cloning functionality. The Region Addressing option can be purchased with either one of the following:

1. The first option is to allow only the ***root*** user to perform region addressing
2. The second option is to give permission to any user to perform region addressing. The ***root*** always has permission even in this second option, however, only a selected set of users up to a maximum of 512 users can be given permission to perform region addressing.

With the purchase of either option, a user can perform Cloning outside their domain by Cloning ***ANY*** physical region that is visible to the kernel even if the region is currently in use by another user. There are therefore several security and stability ramifications with the use of this option. In addition to other restrictions, the main caveat is that the region being Cloned must be capable of handling MsgDma *(not FIFO)* and be made available to the user by the kernel. Since physical addresses are supplied to Region Addressing, care must be taken in making sure that the address and size is valid otherwise results could be unpredictable, resulting in possible DMA and/or kernel crashing or hanging. Recovery from a hanging DMA would require a reloading of the driver.

If the second option is purchased, ***by default whenever the driver is reloaded, no users are given region addressing permission unless specifically granted by the root user.*** User permission is granted on a per card basis and is limited to maximum of 512 users. The way user permission is granted or denied by the ***root*** user is as follows:

```
=== root ===
echo “ccrtngfc_manage_clone_user=<+|->,<Board_Serial_Number|*>,UID1,UID2,…,UIDn”
                                                        > /proc/ccrtngfc
```

The first token must be “*ccrtngfc_manage_clone_user*” and be followed by the ‘=’ sign. Next can be either ‘+’ to add a user or ‘-‘ to remove a user from the list, followed by the ‘,’ sign. After that the ***root*** user can specify either a specific ‘*Board_Serial_Number*’ or ‘*’. If ‘*’ is specified, then the command is applicable to all the boards in the system. The board specification must be followed by the ‘,’ sign, followed by a comma separated list of User IDs. This information is then passed to the driver via the directive ‘*> /proc/ccrtngfc*’. The driver will parse this information and maintain a list of users internally since the driver was loaded, on a per board basis, that are granted *region addressing* permission. Though the driver allows imbedded spaces in the above command, it is recommended to encase them in <”>, especially if you are selecting ‘*’ to specify all the cards.

e.g. to add specific User IDs 1234, 5678, 9 and 10 to a board with a serial number 665413:

```
sudo echo “ccrtngfc_manage_clone_user=+,665413,1234,5678,9,10” > /proc/ccrtngfc
```

If successful, the driver will output on the terminal:

[0:665413] Count of number of users ADDED: 4
[0:665413] CloneRA Users: 1234, 5678, 9, 10
[0:665413] Total number of users allowed Region Addressing permission: 4

To add another user 11223344 to all the cards, you can issue the following:

```
sudo echo "ccrtngfc_manage_clone_user=+,*,11223344" > /proc/ccrtngfc
```

If successful, the driver will output on the terminal:

[0:665413] Count of number of users ADDED: 1
[0:665413] CloneRA Users: 1234, 5678, 9, 10, 11223344
[0:665413] Total number of users allowed Region Addressing permission: 5

If a second board with serial number 665527 exists in the system:

[0:665527] Count of number of users ADDED: 1
[0:665527] CloneRA Users: 11223344
[0:665527] Total number of users allowed Region Addressing permission: 1

To remove the 5678 User ID from the specific board 665413 you can issue the following:

```
sudo echo "ccrtngfc_manage_clone_user=-,665413,5678" > /proc/ccrtngfc
```

If successful, the driver will output on the terminal:

[0:665413] Count of number of users REMOVED: 1
[0:665413] CloneRA Users: 1234, 9, 10, 11223344
[0:665413] Total number of users allowed Region Addressing permission: 4

To remove all User ID entries quickly for all cards, reload the driver.

If you get an invalid argument error, issue the 'dmesg' command and it will supply more information on the error.

No error is generated if the user supplies a specific board serial number that does not exist in the system or attempts to remove User IDs for a board that does not have the User ID in its list or attempts to duplicate a User ID for a board.

At any time, you can issue the *'cat /proc/ccrtngfc'* directive to get information on the list of User IDs that have region addressing permission.

If several users are added in a *single* command line making the string very long *(1000+ characters)*, it is possible that the kernel may break up the string into partial multiple strings before giving to the driver. If that happens, the driver will only see a partial command and could error out. It is therefore suggested that if several User IDs are to be specified multiple commands should be used.

6.5 Reason for Cloning

The ability to Clone a region opens up a whole new way of thinking about accessing hardware and provides an infinite number of scenarios. For example, the simplest case would be to Clone the analog input channels of a card to a physical memory. The user can read the physical memory instead

of the real hardware to get the latest channel information, thus incurring little to no overhead as the read is being performed on a physical memory instead of the board's hardware registers. *(see Example 1)*

A more complex scenario could be if the Region Addressing option is selected. For example, it is possible for the Analog Input card to Clone its local input registers to a physical memory and another physical memory Cloned to the analog output registers of another Analog Output card. In this way, the user can instantaneously acquire changing analog input data from the Analog Input card by reading its Cloned physical memory, process the signals and then write the values to the Analog Output card's Cloned physical memory without incurring any overhead that would have resulted if they were reading from the Analog Inputs local registers and writing to the Analog Outputs local registers. *(see Example 3)*

6.6 Technical

Cloning basically causes the MsgDma engine to run continuously under hardware control. Once initiated, there is no software intervention required to sustain it. The Cloning is asynchronous and a finite interval is required to completely Clone a given region. The time it takes to complete a single Cloning pass is a direct function of the number of words being Cloned, the number of descriptors in use and the region being Cloned.

CCRTNGFC card:

The CCRTNGFC card uses a 100MHz clock for its entire region. It will therefore take 10 nono-seconds to perform a MsgDma transfer for a single 32-bit word. Hence, transfering 12 channels of ADC or DAC will take approximately 120+ nano-seconds for each MsgDma burst. There also appears to be some delay between bursts for hardware synchronization.

6.7 Licensing

This Cloning option is disabled by default. License can be obtained for

- Basic Cloning
- Basic Cloning plus region addressing by only the ***root*** user
- Basic Cloning plus region addressing by *any* user in addition to the ***root*** user

6.8 Features and Limitations

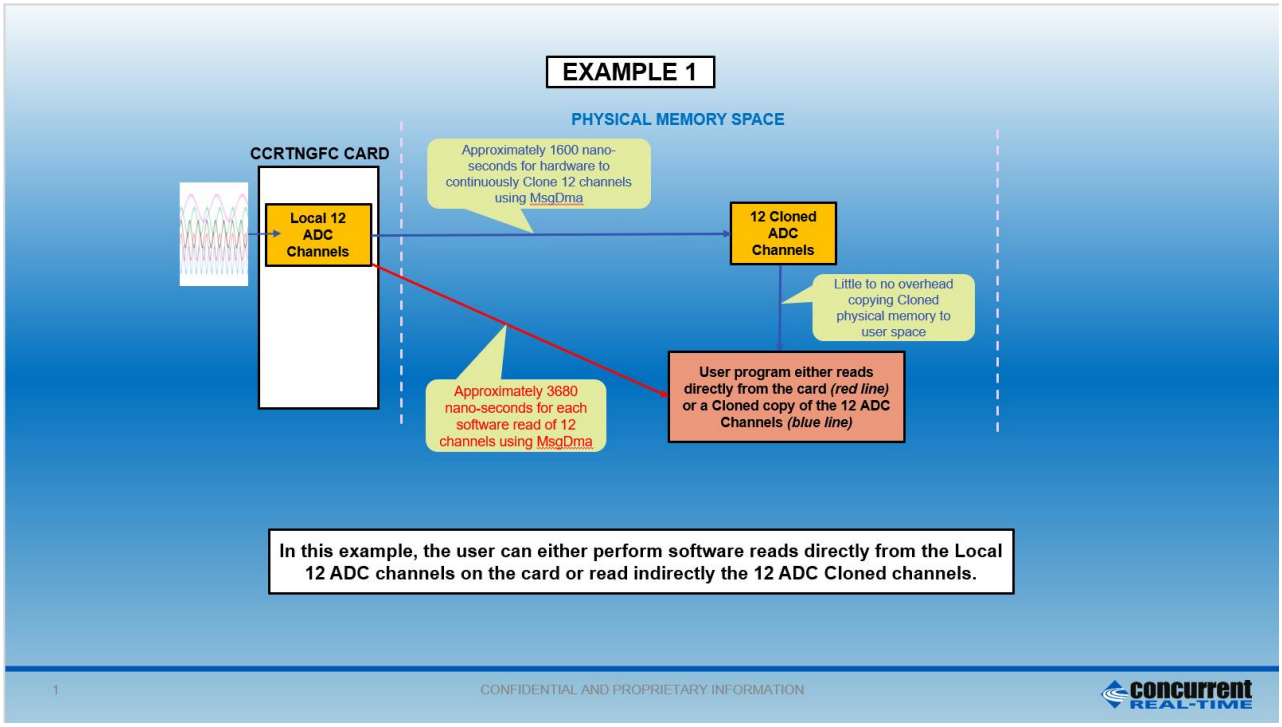
Features:

1. Cloning allows a user the ability to access its hardware with minimal to no overhead
2. Region Addressing can allow Cloning outside the user domain within a system
3. With proper licensing, Region Addressing can be granted access permission to a specific set of users *(maximum 512 users)* in addition to the root user on a per board basis
4. This card can perform Cloning of its entire local memory
5. Cloning operation is easily controlled by various APIs included with the library
6. No CPU intervention occurs during Cloning once transfer has begun
7. Cloning uses MsgDMA as its backbone

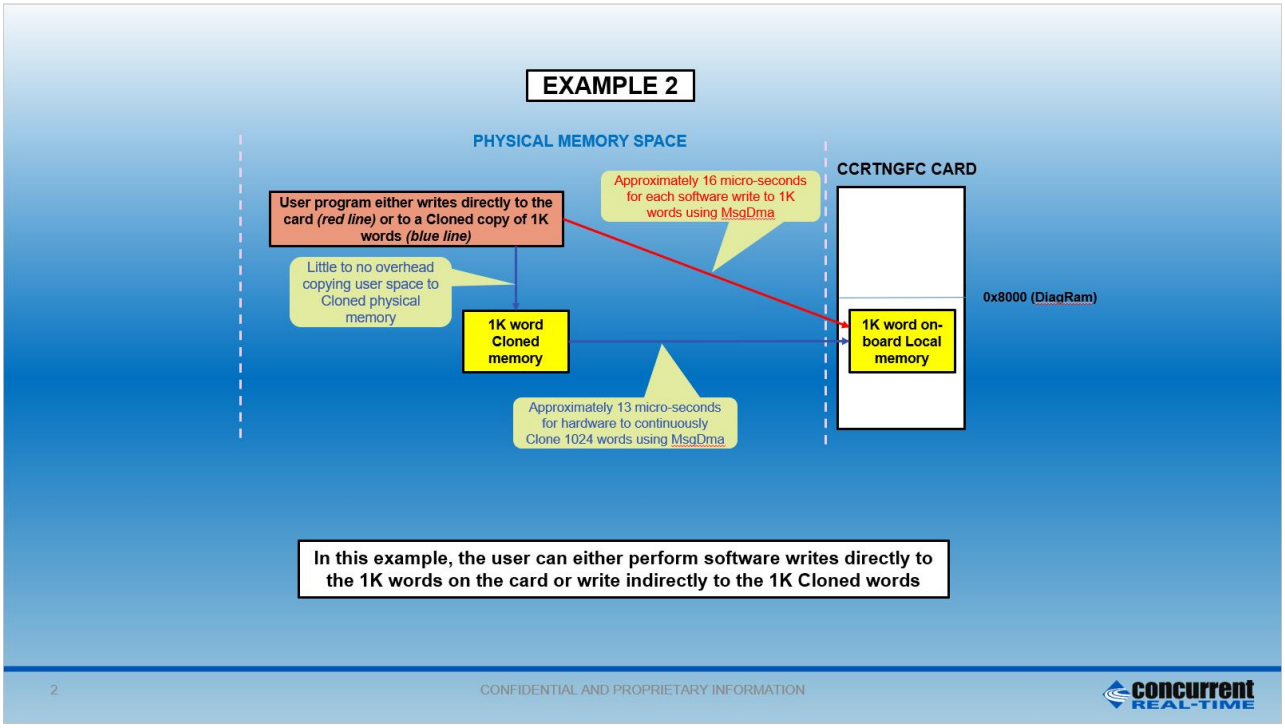
Limitations:

1. Cloning of FIFO region is not supported
2. Cloning it limited to within a system
3. Larger Cloning region will use more PCIe bandwidth as it is Cloning the entire selected region and not just the changing elements within the region
4. Successful Cloning outside the user domain is directly dependent on the region being Cloned

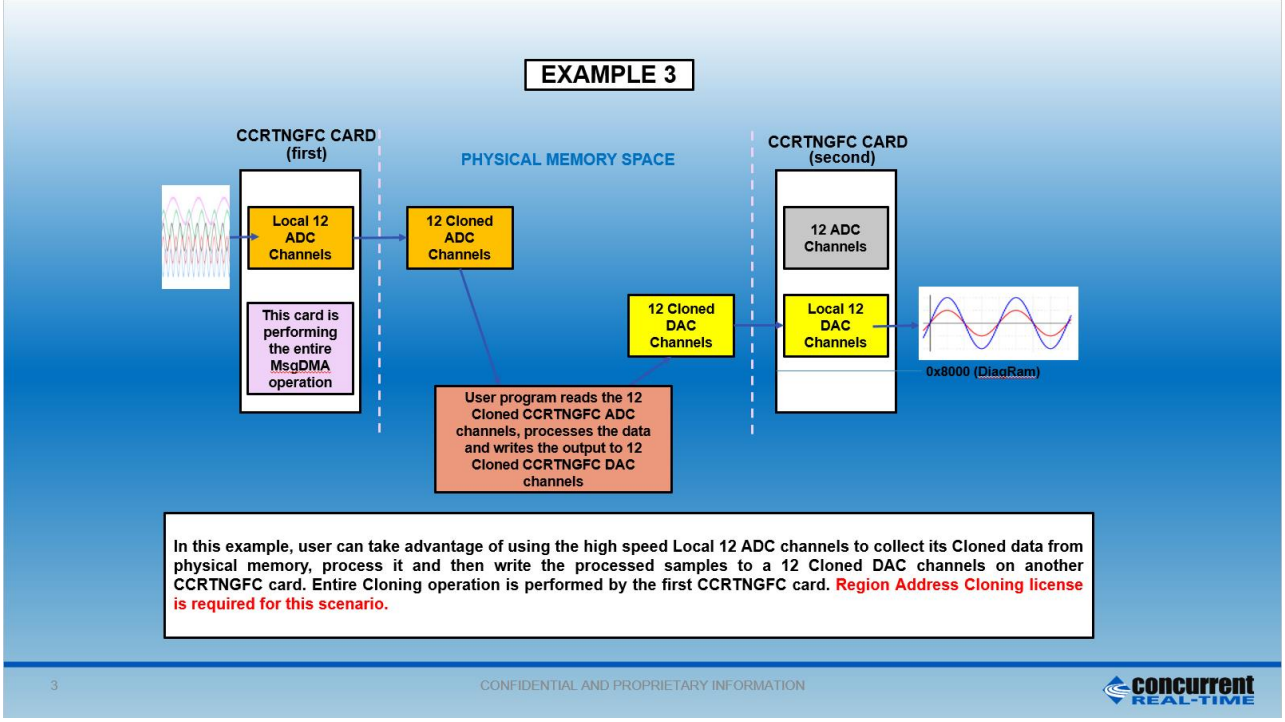
6.9 Example 1



6.10 Example 2



6.11 Example 3



7. Clocks

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

- Clock 0 to 5 – for ADC or DAC
- Clock 6 – for External clock (currently not used)
- Clock 7 – for HighSpeed Daughter Card 1 (future use)
- Clock 8 – for HighSpeed Daughter Card 1 & 2 (future use)
- Clock 9 – Feed-back (Reserved)

Currently, users can select any of the six clocks (Clock 0 to 5) for ADC or DAC. They can also use the same clock if both ADC and DAC are to run at the same clock speed.

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:

- *ccrtNGFC_Reset_Clock()*
- *ccrtNGFC_Clock_Set_Generator_CSR()*
- *ccrtNGFC_Compute_All_Output_Clocks()*
- *ccrtNGFC_Program_All_Output_Clocks()*
- *ccrtNGFC_Clock_Get_Generator_Info()*

Due to the complexity of programming the clock generator and due to hardware limitations *(i.e. 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 which currently defaults to *0.020* parts/trillion. In the latter case, the call will attempt to program the frequencies closest to what the hardware will allow.

7.1.1 Reset All Clocks

This call simply resets and disables *all* the clocks on the board. Not much can be done *(other than Digital I/O)* with the card until the clocks are programmed and running.

7.1.2 Clock Set Generator CSR

This is a useful call to enable and disable clocks once they are programmed. Users can use this to control the contents of the FIFO for both ADC and DAC.

7.1.3 Compute All Output Clocks

Any of the ten clocks can be selected to be programmed with any frequency ranging from 1 Hz to 750 MHz. Since the clocks are sharing hardware resources, there may be certain frequency and clock combinations that will make programming the board impossible. 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 *ccrtNGFC_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 *ccrtNGFC_Program_All_Output_Clocks()* call.

7.1.4 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, DAC etc)* 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.

7.1.5 Get Clock Generator Information

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

8. Calibration

For accurate representation of samples, users can perform calibration of ADC or DAC channels prior to sampling. ADC calibration makes use of either the on-board reference voltage or an external input. DAC calibration uses the ADC input channels to read-back analog output signals. Hence, it is recommended that the ADC be calibrated first prior to calibrating the DAC channels.

8.1.1 ADC Calibration

The simplest way to calibrate all the channels using the internal reference voltage is to use the single call *ccrtNGFC_DC_ADC_Perform_Auto_Calibration()*. This call requires the ADC and the clocks to be in an active state, otherwise it will fail. In normal circumstances, both are active so there is no need to activate them. This call first programs clock generator 0 to the maximum ADC clock frequency and associates all the ADCs with this clock. It also programs the ADCs for two's complement, bi-polar 10 volts operation and then calibrates ADC channels for offset, positive reference and finally negative reference.

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 *ccrtNGFC_DC_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 *ccrtNGFC_DC_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 *_ccrtNGFC_DC_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 corresponding channel selection mask *ChanMask* for any of the above calls.

Note: Since the ADC calibration programs the clock generator for *clock 0* at the *maximum ADC frequency*, it is recommended to first complete auto calibration before programming the clocks for later use.

8.1.2 DAC Calibration

Caution: Anytime the DAC channels are being calibrated, full scale signals are driven on the output channels. It is recommended to *disconnect the outputs* from any external devices if there is any possibility of damaging them during calibration.

For accurate DAC calibration of channels, the user must first enable the ADC and complete *its* calibration. Users can use the *ccrtNGFC_DC_DAC_Perform_Auto_Calibration()* call to perform DAC calibration. Since the DAC is fairly accurate before calibration, you may not see any change to the calibrated DAC offset voltages.

If users prefer that the hardware not perform any calibration for specific channels, one can do that with the use of the corresponding channel selection mask *ChanMask* in the above call.

Note: Since the *ADC* channels are used for DAC calibration, the clocks have to be programmed for sample collection. It is therefore necessary to complete DAC calibration prior to any programming of the clocks for later use.

This page intentionally left blank