Technical Guide
CCURPMFC (WC-CP-FIO)

PCIe Programmable Multi-Function I/O Card (MIOC)

Driver	ccurpmfc (WC-CP-FIO)	
OS	RedHawk (CentOS or Ubuntu based)	
Vendor	Concurrent Real-Time	
Hardware	PCIe Programmable Multi-Function Card (CP-FPGA-Ax)	
Author	Darius Dubash	
Date	February 25th, 2021	Rev 2021.1

This page intentionally left blank

Table of Contents

This page intentionally left blank

1. Introduction

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

- Analog to Digital (ADC) conversion
- Digital to Analog (DAC) conversion
- Digital Input/Output (DIO)
- SDRAM
- Clocks
- Calibration
- Serial Prom

2. Analog to Digital (ADC) Conversion

The ADC has 16 channels with 16-bit resolution, controlled by two ADC converters; each can be assigned one of seven 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 7
ADC 1 – Channel 8 to 15

Prior to performing any conversion, the ADC converter needs to be activated with the *ccurPMFC_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 16 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
- ADC FIFO

Prior to any data being collected, the user needs to configure each ADC in order to select one of 7 individual clocks (0 to 6) as the input signal. 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 two individual ADCs can also be programmed with data format of *offset binary* or *two's complement* and a *bipolar* voltage range of either 5 or 10 volts.

2.1.1 ADC Channel Registers

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

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

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

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

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

b) Alternatively, the user can use the *ccurPMFC_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 *ccurPMFC_Transfer_Data()* library routine. The advantage of this call is that it allows the user to transfer the data via DMA or Programmed I/O. If this call is going to use DMA, then the received user buffer must be a buffer that can allow the board to perform DMA writes. This buffer can be obtained with the help of the *ccurPMFC_MMap_Physical_Memory()* library call.

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

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

The user has the option to supply a *NULL* pointer instead of the *adc_csr* argument, in which case the *ccurPMFC_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 *ccurPMFC_ADC_Get_CSR()* first and then supplying the current configuration to the *adc_csr* argument in the following *ccurPMFC_ADC_Read_Channels()* calls, with the assumption that the ADC configuration is not going to change for the duration of the reads.

2.1.2 ADC FIFO

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

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

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

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

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

   If the user wishes to determine the *floating point* voltages for the *raw* data, they can do so with the help of the *ccurPMFC_DataToVolts()* library call. This call requires as an argument a pointer to the *ccurpmfc_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 *ccurPMFC_Transfer_Data()* library routine. The advantage of this call is that it allows the user to transfer the data via DMA or Programmed I/O. If this call is going to use DMA, then the received user buffer must be a buffer that can allow the board to perform DMA. This buffer can be obtained with the help of the *ccurPMFC_MMap_Physical_Memory()* library call.

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

2.1.3 ADC Input Options

Each of the two ADC's has the option of selecting its inputs either from the external lines *(normal mode)* or from the *calibration bus* with the *ccurPMFC_ADC_Set_CSR()* call. If external lines are selected for an ADC, all 8 ADC channels will return the *raw* digital values for the 8 inputs lines. If calibration bus is selected, then the ADC can receive one of the following with the *ccurPMFC_Set_Calibration_CSR()* call:

- Calibration Ground
- Calibration Postive Reference Voltage
- Calibration Negative Reference Voltage
- Calibration 2.5 Volts Reference
- Calibration 5 Volts Reference
- One of 16 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 8 ADC channels will display the same Calibration reference voltage or DAC channel, depending on the calibration bus selection.***

3. Digital to Analog (DAC) Conversion

The DAC has 16 channels with 16-bit resolution, controlled by four DAC converters. It supports both single-ended and differential outputs. The outputs can be software configured as 16-channel single-ended outputs, 8-channel differential outputs, or a combination of single-ended and differential on a per-DAC granularity. Different channels are identified as a pair of odd/even channels. Unlike the ADC converters where each converter can have

its own clock, all four DAC converters can either be assigned for *software update* or a selection of one of seven update clocks.

DAC to channel association is as follows:

DAC 0 – Channel 0 to 3
DAC 1 – Channel 4 to 7
DAC 2 – Channel 8 to 11
DAC 3 – Channel 12 to 15

Prior to performing any conversion, the DAC converter needs to be activated with the *ccurPMFC_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 16 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
- DAC FIFO

Prior to writing any samples, the user needs to configure the DACs in order to select one of 7 individual clocks (0 to 6) as the *input clock* or instead select *software update*. 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 *ccurPMFC_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 four individual DACs can be programmed with data format of *offset binary* or *two's complement* and a *bipolar* voltage range of either 5 or 10 volts or a *unipolar* voltage range of 10 or 20 volts with the *ccurPMFC_DAC_Set_CSR()* call.

Note: Though the API allows the user to set a maximum *unipolar* range of *20 volts*, the actual maximum voltage that can be output is approximately *12 volts* due to hardware limitations.

Users can also program each individual DAC to operate in *Immediate* or *Synchronized* mode with the *ccurPMFC_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.

3.1.1 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 four 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 four 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_ptr* can be obtained by using the *ccurPMFC_Get_Library_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 *ccurPMFC_VoltsToData()* library call. This call requires as an argument a pointer to the *ccurpmfc_volt_convert_t* structure that holds the current DAC configuration information.

b) Alternatively, the user can use the *ccurPMFC_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 *ccurPMFC_Transfer_Data()* library routine. The advantage of this call is that it allows the user to transfer the data via DMA or Programmed I/O. If this call is going to use DMA, then the transmitting user buffer must be a buffer that can allow the board to perform DMA reads. This buffer can be obtained with the help of the *ccurPMFC_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 *(Programmed I/O or DMA)* with the *ccurPMFC_DAC_Set_Driver_Write_Mode()* library call and then call the *ccurPMFC_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 *ccurPMFC_Write()* call.

e) Finally, the *ccurPMFC_DAC_Write_Channels()* library call 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 *ccurPMFC_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 *ccurPMFC_DAC_Get_CSR()* first and then supplying the current configuration to the *dac_csr* argument in the following *ccurPMFC_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.

3.1.2 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 *ccurPMFC_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 *ccurPMFC_DAC_Reset_Fifo()* call to empty the *DAC FIFO* and resume from a known state. At any time, the *ccurPMFC_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 channel number. Once sampling has resumed, 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 *ccurPMFC_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 be lost.

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 collection is to start the clocks, let them settle and then select a set of channels whose samples are going to be placed in the *DAC FIFO* and then reset the *DAC FIFO* and start writing samples into it for selected channels.

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_ptr* can be obtained by using the *ccurPMFC_Get_Library_Info()* call. Once the pointer is available, the channels can be accessed via the *DAC_FifoData* FIFO 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 *ccurPMFC_VoltsToData()* library call. This call requires as an argument a pointer to the *ccurpmfc_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 *ccurPMFC_Transfer_Data()* library routine. The advantage of this call is that it allows the user to transfer the data via DMA or Programmed I/O. If this call is going to use DMA, then the transmitting user buffer must be a buffer that can allow the board to perform DMA reads. This buffer can be obtained with the help of the *ccurPMFC_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 *(Programmed I/O or DMA)* with the *ccurPMFC_DAC_Set_Driver_Write_Mode()* library call and then call the *ccurPMFC_Write()* routine to write *raw* data to the DAC FIFO. ***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 *ccurPMFC_Write()* call.

4. Digital Input/Output (DIO)

This board supports 96 digital input or output lines. The direction call *ccurPMFC_DIO_Set_Ports_Direction()* can be used to select the direction of a set of DIO ports. The lines are grouped into ports where there are four consecutive lines assigned to a port.

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

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 *ccurPMFC_DIO_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.

4.1.1 Digital Input

User can program 1 to 24 ports as *inputs* with the help of the *ccurPMFC_DIO_Set_Ports_Direction()* call. A read issued to the lines associated with the input ports using the *ccurPMFC_DIO_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)*

4.1.1.1 Continuous Mode

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 *ccurPMFC_DIO_Read_Input_Channel_Registers()* call, channels could change their current state, thus not reflecting the *instantaneous* state of *all* the channels.

If this is the desired mode of operation, the user needs to first issue the *ccurPMFC_DIO_Set_Input_Snapshot()* call with the *CCURPMFC_DIO_INPUT_OPERATION_CONTINUOUS* option. This can be followed by multiple input channel reads with the *ccurPMFC_DIO_Read_Input_Channel_Registers()* call and the *dio_snapshot* argument set to the *CCURPMFC_DO_NOT_CHANGE* option.

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

4.1.1.2 Snapshot Mode

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

If this is the desired mode of operation, the user can simply use the *ccurPMFC_DIO_Read_Input_Channel_Registers()* call with the *dio_snapshot* argument set to the *CCURPMFC_DIO_INPUT_OPERATION_SNAPSHOT*. There is no need to issue the initial *ccurPMFC_DIO_Set_Input_Snapshot()* call.

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

4.1.1.3 Change-Of-State

The card provides capability to detect when a digital input line changes state. Detection can be for either for 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 the selected channels with the help of the *ccurPMFC_DIO_Set_COS_Channels_Enable()* API. Additionally the *ccurPMFC_DIO_Set_COS_Channels_Edge_Sense()* and the *ccurPMFC_DIO_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 *ccurPMFC_Create_UserDioCosInterruptHandler()* 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 enable the user to determine the cause of the interrupt. User needs to ensure that the duration of processing 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 *ccurpmfc_dio_intr* that is supplied with this driver, you can follow similar steps for your system:

```
# === as root ===
# shield –a 2, 4-5                    (shield processors 2, 4 and 5)
# cat /proc/ccurpmfc                  (get board irq – in this case it is 'irq=56')
# echo 4 > /proc/irq/56/smp_affinity  (direct board irq to be handled by processor 2)
# (if irq '56' is not present in the proc/irq directory, then you will need to start the test at least once to get it
  assigned by the kernel)
# run –b4-5 ./ccurpmfc_dio_intr
```

4.1.2 Digital Output

User can program 1 to 24 ports as *outputs* with the help of the *ccurPMFC_DIO_Set_Ports_Direction()* call. A write issued to the *output* registers with the *ccurPMFC_DIO_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 were switched to *outputs*.
At any time, the users can read back the *output* registers that were last written to with the *ccurPMFC_DIO_Read_Output_Channel_Registers()* call.

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

- Continuous
- Synchronous *(Simultaneously)*

4.1.2.1 Continuous Mode

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

If this is the desired mode of operation, the user needs to first issue the *ccurPMFC_DIO_Set_Output_Sync()* call with the *CCURPMFC_DIO_OUTPUT_OPERATION_CONTINUOUS* option. This can be followed by multiple output channel writes with the *ccurPMFC_DIO_Write_Output_Channel_Registers()* call and the *dio_sync* argument set to the *CCURPMFC_DO_NOT_CHANGE* option.

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

4.1.2.2 Synchronous Mode

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

If this is the desired mode of operation, the user needs to first issue the *ccurPMFC_DIO_Set_Output_Sync()* call with the *CCURPMFC_DIO_OUTPUT_OPERATION_SYNC* option. This can be followed by multiple output channel writes with the *ccurPMFC_DIO_Write_Output_Channel_Registers()* call and the *dio_sync* argument set to the *CCURPMFC_DIO_OUTPUT_OPERATION_SYNC* option.

Note: As long the board is operating in *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.

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. Basic DMA *(Direct Memory Access)*
3. Modular Scatter-Gather DMA

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

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

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

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

1. Programmed I/O
   - ccurPMFC_Fast_Memcpy()

- ccurPMFC_Fast_Memcpy_Unlocked()
- ccurPMFC_Fast_Memcpy_Unlocked_FIFO()
- ccurPMFC_Transfer_Data()
- ccurPMFC_Get_Mapped_Local_Ptr() // pointer to the card local memory - advanced users only
- ccurPMFC_Read() // for reading ADC channels
- ccurPMFC_ADC_Read_Channels() // for reading ADC channels
- ccurPMFC_ADC_Read_Channels_Calibration() // for reading ADC channel calibration values
- ccurPMFC_DAC_Read_Channels() // for reading DAC channels
- ccurPMFC_DAC_Read_Channels_Calibration() // for reading DAC channel calibration values
- ccurPMFC_DAC_ReadBack_Channels // for reading DAC readback channels
- ccurPMFC_DAC_Write_Channels() // for writing to DAC channels
- ccurPMFC_Write() // for writing DAC channels
- ccurPMFC_DAC_Write_Channels_Calibration() // for writing to DAC channel calibration
- ccurPMFC_Get_Value() // to read specific values on the board registers
- ccurPMFC_Set_Value() // to write specific values to the board registers

2. Basic DMA
   - ccurPMFC_Transfer_Data()
   - ccurPMFC_DMA_Configure()
     ccurPMFC_DMA_Fire()

3. Modular Scatter-Gather DMA
   - ccurPMFC_Transfer_Data() // single MsgDma transfer
   - ccurPMFC_MsgDma_Seize() // single MsgDma transfer
     ccurPMFC_MsgDma_Configure_Single()
     ccurPMFC_MsgDma_Fire_Single()
     ccurPMFC_MsgDma_Release()
   - ccurPMFC_MsgDma_Seize() // multiple MsgDma transfer
     ccurPMFC_MsgDma_Configure_Descriptor()
     ccurPMFC_MsgDma_Setup()
     ccurPMFC_MsgDma_Fire()
     ccurPMFC_MsgDma_Release()

6. SDRAM

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

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

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

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

6.1.1 SDRAM Read

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

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

6.1.2 SDRAM Write

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

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

7. Clocks

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

- Clock 0 to 6 – for ADC or DAC
- Clock 7 – for SDRAM
- Clock 8 and 9 – Reserved

Currently, users can select any of the seven clocks (Clock 0 to 6) for ADC or DAC. They can also use the same clock if both ADC and DAC are to run at the same clock speed.

Clock 7 is only used by the SDRAM and must be programmed and running at 10MHz prior to performing any SDRAM operations.

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:

- *ccurPMFC_Reset_Clock()*
- *ccurPMFC_Compute_All_Output_Clocks()*
- *ccurPMFC_Program_All_Output_Clocks()*
- *ccurPMFC_Clock_Get_Generator_Info()*

Due to the complexity of programming the clock generator and due to hardware limitations *(different clocks sharing same resources)*, a user cannot *append* to or *change* already running clocks. If multiple clocks are to be used, then the user needs to program all the clocks with the single call prior to commencing.

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 DIO)* with the card until the clocks are programmed and running.

7.1.2 Compute All Output Clocks

Any of the ten clocks can be selected to be programmed with any frequency ranging from 1 Hz to 250 MHz. Since the clocks are sharing hardware resources, there may be certain frequency and clock combinations that will make programming the board impossible. 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 *ccurPMFC_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 *ccurPMFC_Program_All_Output_Clocks()* call.

7.1.3 Program All Output Clocks

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

7.1.4 Get Clock Generator Information

This call provides detailed information for any of the selected clock generators in the *CgInfo* argument of the *ccurPMFC_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 *ccurPMFC_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 *ccurPMFC_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 *ccurPMFC_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 *_ccurPMFC_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 *ccurPMFC_ADC_Set_Offset_Cal()* call with 0 volts offset and a gain of 1 for the *ccurPMFC_ADC_Set_Positive_Cal()* and *ccurPMFC_ADC_Set_Negative_Cal()* calls. Users can skip calibration data for channels being update by setting the corresponding channel with the *CCURPMFC_DO_NOT_CHANGE* flag instead.

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 *ccurPMFC_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.

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.

9. Serial PROM

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

This page intentionally left blank