Using Adc12x5

Introduction

The Adc12x5 library provides a comprehensive set of task-level I/O drivers for the Instrumentation Department's twelve channel 5 megahertz VME digitizer module. The library is written in C++ for VxWorks and contains 2 user classes, the first supporting programmed (software controlled) data transfers and another using the host CPU's DMA engine. A C language callable set of wrapper functions is provided to give C and VxWorks startup script programmers full access to the hardware module. The full flexibility of the Adc12x5 module's configuration and acquisition mode parameters is preserved and made available to the programmer through the various class methods. The C++ classes will be described in detail and the C language wrapper functions, which map one-to-one to the class methods, will be outlined.

Dehong Zhang has written interface software for EPICS to provide access to the Adc12x5 library via process variables. Please see "Using Adc12x5 in EPICS" for a description of that package.

Overview

To use the Adc12x5 module the programmer must first establish an A16 VME bus address for the base of the module's 512 byte configuration register space, and an A32 VME bus address for the base address of the ADC sample data. The 16 bit ADC sample data are packed 2 samples per thirty-two bit word in that A32 space. The number of active channels and number of samples per channel per trigger are programmable so the A32 data size is determined by the module configuration. Instantiation of the Adc12x5 class establishes the A16 address while ADC channel configuration with the ChannelSelect method establishes the A32 address and size:

```
#include "vmemap.h"        // Map VME Addresses to Processor Local Addresses
#include "adc12x5.h"       // VME 12 Channel 5 MHz ADC Module
using InstADC::Adc12x5;

const int unsigned kDataChannels = 12;
const int unsigned kDataSamples = 1024;
void * const kAdcA16Base = reinterpret_cast<void * const>( 0x2000 );
void * const kAdcA32Base = reinterpret_cast<void * const>( 0x09000000 );

Adc12x5::eStatus    status;

class Adc12x5       *adcPtr;= new ( VmeA16ToLocal( kAdcA16Base ) ) Adc12x5(
```

```
  &status, "TestADC",
  Adc12x5::kBaseClockInternal,
  Adc12x5::kSampleAndHold, 0 );

adcPtr->ChannelSelect(
  Adc12x5::kChanMaskAll,
  Adc12x5::eChannelSampleCount( kDataSamples ),
  kAdcA32Base );
```

After establishing the two VME address spaces the programmer must configure the module's input range and measurement mode:

```
adcPtr->RangeSelect(
  Adc12x5::kChanMaskAll,
  Adc12x5::kHighBipolar );

adcPtr->ModeSelect(
  Adc12x5::kMultSampScopeSwArm,
  Adc12x5::kSamplePeriodMin,
  Adc12x5::eChannelSampleCount( kDataSamples ) );
```

Finally the programmer collects the data:

```
static float        buffer[ kDataChannels ][ kDataSamples ];

// software triggering with sample period determined by task delays
while ( true ) {
  taskDelay( sysClkRateGet() / 15 );
  adcPtr->SoftwareTrigger();
  adcPtr->Volts( buffer ); // read data in volts
}
```

This example established a very simple software-triggered measurement with programmed data readout of floating point values. The Adc12x5 library supports other triggering and data readout schemes that will be outlined in the example sections below. The various class constructors and methods will be described in the "Adc12x5 Class" and "Adc12x5Dma Class" sections below. Please read the adc12x5decls.h, adc12x5.h and adc12x5dma.h files for the detailed definition of all method parameter values.

The Adc12x5 library is developed and hosted on the node nova.fnal.gov in a CVS repository named adc12x5. The header files can be found in:

/home/rfies/esd/rfiinst/inc/ adc12x5decls.h
/home/rfies/esd/rfiinst/inc/adc12x5.h

/home/rfies/esd/rfiinst/inc/c_adc12x5.h
/home/rfies/esd/rfiinst/inc/adc12x5dma.h
/home/rfies/esd/rfiinst/inc/c_adc12x5dma.h

The library object files may be found in:

/fecode-bd/vxworks_boot/fe/rfiinst/lib/VW_55/MVME5500/libadc12x5.out
/fecode-bd/vxworks_boot/fe/rfiinst/lib/VW_55/MVME2434/libadc12x5.out
/fecode-bd/vxworks_boot/fe/rfiinst/lib/VW_55/MVME2401/libadc12x5.out
/fecode-bd/vxworks_boot/fe/rfiinst/lib/VW_55/MVME2301/libadc12x5.out

/fecode-bd/vxworks_boot/fe/rfiinst/lib/VW_61/MVME5500/libadc12x5.out
/fecode-bd/vxworks_boot/fe/rfiinst/lib/VW_61/MVME2434/libadc12x5.out
/fecode-bd/vxworks_boot/fe/rfiinst/lib/VW_61/MVME2401/libadc12x5.out

The working examples presented in this note are hosted on the node nova.fnal.gov in a CVS repository named adc12x5example.

Adc12x5 Class

The Adc12x5 class provides support for basic hardware configuration, data acquisition complete interrupt generation and programmed data readout functionality. Please see the header file adc12x5.h for complete method declarations. The class provides the following methods:

• Instantiation::
  - Adc12x5 ::Adc12x5() specifies values for the base clock and ADC sample mode registers
  - new() specifies the A16 VME address for the module instance
• Configuration:
  - ChannelSelect() specifies values for the channel mask, sample count and A32 pointer registers
  - RangeSelect() specifies values for any or all of the input range registers
  - ModeSelect() specifies values for the mode, sample rate, sample per trigger and trigger count registers
  - SamplePeriod configures the digitizer sample period
  - InterruptSelect() specifies values for the IRQ status ID and IRQ level registers
  - SoftwareTrigger() software triggers the module when in software trigger (arm) modes
  - Arm() arms the module trigger when in hardware trigger (arm) modes

• Data readout:
  - Counts() copies all raw digitizer data to a buffer in user memory
  - Volts() converts raw digitizer data previously stored in user memory with the above Counts method to floating point values in units of volts
  - Counts() (overloaded method) copies raw digitizer data for the specified channels to a buffer in user memory
  - Volts() (overloaded method) reads and converts all raw digitizer data into floating point values in units of volts in user memory
  - Volts() (overloaded method) reads and converts raw digitizer data for the specified channels into floating point values in units of volts in user memory
  - Scale() reads and converts raw digitizer data for the specified channels into floating point values in user memory using the specified range and offset scaling factors
• Utility methods:
  - Adc12x5::~Adc12x5() deletes a module instance
  - Disable() disables the module measurement and interrupt activity
  - Running() returns true if the digitizer is actively making measurements
  - ChannelCount() returns the number of active configured channels
  - ChannelSampleCount() returns the number of digitizer samples produced by each channel
  - DataPtr() returns the base address of the module's output data buffer as a pointer to volatile void
  - ChannelBytes() returns the size in bytes of the data for a single channel
  - ModuleBytes() returns the size in bytes of the data across all channels
  - PrintVersion() prints the class version information on the console
  - Display() displays the module's register contents on the console
  - InstanceCount() returns the total number of Adc12x5 instances

Adc12x5Dma Class

The Adc12x5Dma class builds on the Adc12x5 class to provide support for DMA data collection with either callback function or message queue data acquisition complete notification. Please see the header file adc12x5dma.h for complete method declarations. The class provides the following methods:
• Instantiation:
  - Adc12x5Dma::Adc12x5Dma() specifies values for the base clock and ADC sample mode registers
• Configuration:

- DataSelect() specifies the address of the user's data destination buffer, a data collection complete callback function and a parameter for that callback function
- DataSelect() (overloaded method) specifies the address of the user's data destination buffer, a data collection complete message queue pointer and a message pointer for that message queue
- AutoRearm() - enables or disables automatic digitization rearm after data collection in hardware triggered (armed) modes
- InterruptSelect() specifies the hardware interrupt vector and level digitization complete interrupt processing, and a task priority for DMA data collection processing

• Utility methods:

- Adc12x5::~Adc12x5() deletes a module instance

Example 1 - Software Triggered Interrupt Driven Programmed Data Acquisition

This example shows how to do interrupt driven data acquisition on software generated triggers with programmed data readout. The user creates a synchronization semaphore and an interrupt service routine that gives the semaphore for each Adc12x5 data acquisition complete interrupt. The data readout task takes the semaphore and reads the data under program control.

```
/*
**  module include files  **
*/
#include <vxWorks.h>      // for the following...
#include <intLib.h>       // for intConnect() and related stuff
#include <sysLib.h>       // for sysIntEnable() & sysClkRateGet()
#include <iv.h>                 // for INUM_TO_IVEC

#include "logging.h"      // message logging declarations
#include "vmemap.h"       // Map VME Addresses to Processor Local Addresses
#include "binarysemaphore.h" // for simple binary semaphore with timeout
#include "adc12x5.h"      // VME 12 Channel 5 MHz ADC Module

using InstADC::Adc12x5;

/*
**  module constant definitions  **
*/
static const unsigned int kDataInterruptVector = 170;
static const bool kDisplayErrors = true;

static const int unsigned kDataChannels = 12;
static const int unsigned kDataSamples = 1024;

void * const kAdcA16Base = reinterpret_cast<void * const>( 0x2000 );
void * const kAdcA32Base = reinterpret_cast<void * const>( 0x09000000 );
```

```
/*
Data readout interrupt service routine.
Informs interrupt handler of the availability of data via semaphore.
*/
static void DataReadoutISR( class Semaphore *semPtr ) {
  semPtr->Give( kDisplayErrors );
  return;
}
/*
*/

/*
Data readout interrupt handler.
Waits for interrupt service routine to announce
data arrival and then reads data to local memory.
*/
static void DataReadoutTask( void ) {

  // create ISR<-->handler synchronization semaphore
  ::eStatus         semStatus;
  class Semaphore   *semPtr = new Semaphore( &semStatus, Semaphore::kEmpty );
  if ( ::kError == semStatus ) {
    LogError( "DataReadoutTask() - Could not create semaphore!\n",0,0,0,0,0,0);
    return;
  }

  // create digitizer instance
  Adc12x5::eStatus  status;
  class Adc12x5     *adcPtr = new ( VmeA16ToLocal( kAdcA16Base ) ) Adc12x5(
    &status, "ADC0", Adc12x5::kBaseClockInternal, Adc12x5::kSampleAndHold, 0 );
  if ( Adc12x5::kOk != status ) {
    LogError( "DataReadoutTask() - Could not create Adc12x5!\n", 0,0,0,0,0,0 );
    return;
  }

  adcPtr->ChannelSelect(
    Adc12x5::kChanMaskAll,
    Adc12x5::eChannelSampleCount( kDataSamples ),
    kAdcA32Base );

  adcPtr->RangeSelect(
    Adc12x5::kChanMaskAll,
    Adc12x5::kHighBipolar );
  // Mode 5 - Ns samples, at Sr sample rate, software arm

  adcPtr->ModeSelect(
    Adc12x5::kMultSampScopeSwArm,
    Adc12x5::kSamplePeriodMin,
    Adc12x5::eChannelSampleCount( kDataSamples ) );

  // create interrupt service routine
  if ( OK == intConnect(
              INUM_TO_IVEC( kDataInterruptVector ),
```

```
            VOIDFUNCPTR( DataReadoutISR ),
            int( semPtr ) ) ) {
    adcPtr->InterruptSelect( kDataInterruptVector, Adc12x5::kIntLevel1 );
    sysIntEnable( Adc12x5::kIntLevel1 );
  } else {
    LogError( "DataReadoutTask() - Could not create ISR!\n", 0,0,0,0,0,0 );
    return;
  }

  static float      buffer[ kDataChannels ][ kDataSamples ];

  while ( true ) {
    taskDelay( sysClkRateGet() / 15 ); // ~15 Hz acquisition
    adcPtr->SoftwareTrigger();
    semPtr->Take( Semaphore::kWaitForever, kDisplayErrors );
    adcPtr->Volts( buffer ); // read data in volts
  }

}
/*
*/
```

Example 2 - Hardware Triggered Interrupt Driven Programmed Data Acquisition

This example shows how to do interrupt driven data acquisition on external hardware generated triggers with programmed data readout. The user creates a synchronization semaphore and an interrupt service routine that gives the semaphore for each Adc12x5 data acquisition complete interrupt. The data readout task takes the semaphore and reads the data under program control.

```
/*
**  module include files  **
*/
#include <vxWorks.h>       // for the following...
#include <intLib.h>        // for intConnect() and related stuff
#include <sysLib.h>        // for sysIntEnable() & sysClkRateGet()
#include <iv.h>                  // for INUM_TO_IVEC

#include "logging.h"       // message logging declarations
#include "vmemap.h"        // Map VME Addresses to Processor Local Addresses
#include "binarysemaphore.h" // for simple binary semaphore with timeout
#include "adc12x5.h"       // VME 12 Channel 5 MHz ADC Module

using InstADC::Adc12x5;

/*
**  module constant definitions  **
*/
static const unsigned int kDataInterruptVector = 170;
static const bool kDisplayErrors = true;

static const int unsigned kDataChannels = 12;
```

```
static const int unsigned kDataSamples = 1024;

void * const kAdcA16Base = reinterpret_cast<void * const>( 0x2000 );
void * const kAdcA32Base = reinterpret_cast<void * const>( 0x09000000 );

/*
Data readout interrupt service routine.
Informs interrupt handler of the availability of data via semaphore.
*/
static void DataReadoutISR( class Semaphore *semPtr ) {
  semPtr->Give( kDisplayErrors );
  return;
}
/*
*/

/*
Data readout interrupt handler.
Waits for interrupt service routine to announce
data arrival and then DMAs data to local memory.
*/
static void DataReadoutTask( void ) {

  // create ISR<-->handler synchronization semaphore
  ::eStatus         semStatus;
  class Semaphore   *semPtr = new Semaphore( &semStatus, Semaphore::kEmpty );
  if ( ::kError == semStatus ) {
    LogError( "DataReadoutTask() - Could not create semaphore!\n",0,0,0,0,0,0);
    return;
  }

  // create digitizer instance
  Adc12x5::eStatus  status;
  class Adc12x5     *adcPtr = new ( VmeA16ToLocal( kAdcA16Base ) ) Adc12x5(
    &status, "ADC0", Adc12x5::kBaseClockInternal, Adc12x5::kSampleAndHold, 0 );
  if ( Adc12x5::kOk != status ) {
    LogError( "DataReadoutTask() - Could not create Adc12x5!\n", 0,0,0,0,0,0 );
    return;
  }

  adcPtr->ChannelSelect(
    Adc12x5::kChanMaskAll,
    Adc12x5::eChannelSampleCount( kDataSamples ),
    kAdcA32Base );

  adcPtr->RangeSelect(
    Adc12x5::kChanMaskAll,
    Adc12x5::kHighBipolar );

  // Mode 6 - Ns samples, at Sr sample rate, external SYNC arm
  adcPtr->ModeSelect(
    Adc12x5::kMultSampScopeExtArm,
    Adc12x5::kSamplePeriodMin,
    Adc12x5::eChannelSampleCount( kDataSamples ) );
```

```
  // create interrupt service routine
  if ( OK == intConnect(
              INUM_TO_IVEC( kDataInterruptVector ),
             VOIDFUNCPTR( DataReadoutISR ),
             int( semPtr ) ) ) {
    adcPtr->InterruptSelect( kDataInterruptVector, Adc12x5::kIntLevel1 );
    sysIntEnable( Adc12x5::kIntLevel1 );
  } else {
    LogError( "DataReadoutTask() - Could not create ISR!\n", 0,0,0,0,0,0 );
    return;
  }

  static float      buffer[ kDataChannels ][ kDataSamples ];

  while ( true ) {
    adcPtr->Arm();  // arm the digitizer for the next trigger
    semPtr->Take( Semaphore::kWaitForever, kDisplayErrors );
    adcPtr->Volts( buffer ); // read data in volts
  }

}
/*
*/
```

Example 3 - Hardware Triggered Interrupt Driven DMA Data Acquisition with Callback Notification

This example shows how to do interrupt driven data acquisition on external hardware generated triggers with DMA data readout. The user creates a synchronization semaphore and a callback routine that gives the semaphore for each Adc12x5 data acquisition complete interrupt. Interrupt processing is handled within the Adc12x5Dma class. The data readout task takes the semaphore and requests the data in floating point form. The data are read into a holding buffer within the Adc12x5Dma class by DMA. The holding buffer is then converted to floating point values in user memory.

```
/*
**  module include files  **
*/
#include <vxWorks.h>      // for the following...
#include <taskLib.h>      // for taskDelay()
#include <sysLib.h>       // for sysClkRateGet()

#include "logging.h"      // message logging declarations
#include "vmemap.h"       // Map VME Addresses to Processor Local Addresses
#include "binarysemaphore.h" // for simple binary semaphore with timeout
#include "adc12x5dma.h"   // VME 12 Channel 5 MHz ADC Module

using InstADC::Adc12x5;

/*
```

```
**  module constant definitions  **
*/
static const int unsigned kDataInterruptVector = 170;
static const int unsigned kReadoutTaskPriority = 33;

static const int unsigned kDataChannels = 12;
static const int unsigned kDataSamples = 1024;

static bool const kNasty = false;
static const bool kDisplayErrors = true;

void * const kAdcA16Base = reinterpret_cast<void * const>( 0x2000 );
void * const kAdcA32Base = reinterpret_cast<void * const>( 0x09000000 );

using InstADC::Adc12x5Dma;

/*
data collection complete callback function
*/
void Callback( Adc12x5Dma::eStatus const status, void * const paramPtr ) {
  class Semaphore *semPtr = static_cast<class Semaphore *>( paramPtr );
  semPtr->Give( kDisplayErrors );
  return;
}
/*
*/

static void DataReadoutTask( void ) {

  // create callback<-->handler synchronization semaphore
  ::eStatus         semStatus;
  class Semaphore   semaphore( &semStatus, Semaphore::kEmpty );
  if ( ::kError == semStatus ) {
    LogError( "DataReadoutTask() - Could not create semaphore!\n",0,0,0,0,0,0);
    return;
  }

  Adc12x5Dma::eStatus status;
  Adc12x5Dma *adcPtr= new ( VmeA16ToLocal( kAdcA16Base ) ) Adc12x5Dma(
    &status,
    "ADC00",
    kNasty,
    Adc12x5Dma::kBaseClockInternal,
    Adc12x5Dma::kSampleAndHold,
    0 );

  adcPtr->ChannelSelect(
    Adc12x5Dma::kChanMaskAll,
    Adc12x5Dma::eChannelSampleCount( kDataSamples ),
    kAdcA32Base );

  adcPtr->RangeSelect(
    Adc12x5Dma::kChanMaskAll,
    Adc12x5Dma::kHighBipolar );
```

```
  adcPtr->ModeSelect(
    Adc12x5Dma::kMultSampScopeExtArm,
    Adc12x5Dma::kSamplePeriodMin,
    Adc12x5Dma::eChannelSampleCount( kDataSamples ) );

  // tell class to allocate a raw buffer
  adcPtr->DataSelect( NULL,
    Callback,
    &semaphore );

  adcPtr->InterruptSelect( kDataInterruptVector,
    Adc12x5Dma::kIntLevel1,
    kReadoutTaskPriority );

  static float      buffer[ kDataChannels ][ kDataSamples ];

  // arm the digitizer for the first trigger
  // auto rearm is enabled by default
  adcPtr->Arm();

  while ( true ) {
    semaphore.Take( Semaphore::kWaitForever, kDisplayErrors );
    adcPtr->GetVolts( buffer );
  }

  return;

}
/*
*/
```

Example 4 - Hardware Triggered Interrupt Driven DMA Data Acquisition with Message Queue Notification

This example shows how to do interrupt driven data acquisition on external hardware generated triggers with DMA data readout. The user creates a synchronization message queue that is passed to the Adc12x5Dma class which will announce data acquisition complete on the message queue. Interrupt processing is handled within the Adc12x5Dma class. The data readout task receives the message and requests the data in floating point form. The data are read into a holding buffer within the Adc12x5Dma class by DMA. The holding buffer is then converted to floating point values in user memory. In this example the digitizer is manually rearmed under software control in the data acquisition loop. This feature is useful in cases where the external trigger signal rate is higher than the useful data readout rate.

```
/*
**  module include files  **
*/
```

```
#include <vxWorks.h>      // for the following...
#include <taskLib.h>      // for taskDelay()
#include <sysLib.h>       // for sysClkRateGet()

#include "logging.h"      // message logging declarations
#include "vmemap.h"       // Map VME Addresses to Processor Local Addresses
#include "binarysemaphore.h" // for simple binary semaphore with timeout
#include "adc12x5dma.h"   // VME 12 Channel 5 MHz ADC Module

using InstADC::Adc12x5;

/*
**  module constant definitions  **
*/
static const int unsigned kDataInterruptVector = 170;
static const int unsigned kReadoutTaskPriority = 33;

static const int unsigned kDataChannels = 12;
static const int unsigned kDataSamples = 1024;

static bool const kNice = true;
static const bool kDisplayErrors = true;
static const int unsigned kQueueDepth = 2;

void * const kAdcA16Base = reinterpret_cast<void * const>( 0x2000 );
void * const kAdcA32Base = reinterpret_cast<void * const>( 0x09000000 );

using InstADC::Adc12x5Dma;

static void DataReadoutTask( void ) {

  Adc12x5Dma::eStatus adcStatus;
  class Adc12x5Dma *adcPtr = new ( VmeA16ToLocal( kAdcA16Base ) ) Adc12x5Dma(
    &adcStatus,
    "ADC00",
    kNice,
    Adc12x5Dma::kBaseClockInternal,
    Adc12x5Dma::kSampleAndHold,
    0 );

  adcPtr->ChannelSelect(
    Adc12x5Dma::kChanMaskAll,
    Adc12x5Dma::eChannelSampleCount( kDataSamples ),
    KAdcA32Base );

  adcPtr->RangeSelect(
    Adc12x5Dma::kChanMaskAll,
    Adc12x5Dma::kHighBipolar );

  adcPtr->ModeSelect(
    Adc12x5Dma::kMultSampScopeExtArm,
    Adc12x5Dma::kSamplePeriodMin,
    Adc12x5Dma::eSampleCount( kDataSamples ) );
```

```
  // we will rearm the digitizer to govern the DAQ rate
  adcPtr->AutoRearm( false );

  // DAQ complete message queue
  ::eStatus queueStatus;
  class MessageQueue<Adc12x5Dma::DaqCompleteMessage>
    completedQueue( &queueStatus, kQueueDepth );
  const int kBoard1 = 1;

  // tell class to allocate a raw buffer
  adcPtr->DataSelect(
    NULL,
    &completedQueue,
    &kBoard1 );

  adcPtr->InterruptSelect(
    kDataInterruptVector,
    Adc12x5Dma::kIntLevel1,
    kReadoutTaskPriority );

  static float      buffer[ kDataChannels ][ kDataSamples ];
  Adc12x5Dma::DaqCompleteMessage message;

  while ( true ) {
    taskDelay( sysClkRateGet() / 15 ); // ~15 Hz acquisition
    adcPtr->Arm(); // rearm the digitizer
    completedQueue.Receive(
      message,
      MessageQueue<Adc12x5Dma::DaqCompleteMessage>::eWait( 100 ),
      kDisplayErrors );
    adcPtr->GetVolts( buffer );
  }

}
```

C Language Wrappers

A set of wrapper functions provides full functionality for C language programmers. Wrappers are provided for the Adc12x5 and Adc12x5Dma classes. In both cases the ‘constructor’ function returns an instance index that is passed as the first parameter of all other related functions. All other wrapper function parameters map one-to-one with those of the underlying class methods.

Adc12x5 Wrapper Functions

Please see the class Adc12x5 discussion above and the header file c_adc12x5.h for complete function declarations. The wrapper library provides the following functions:

- int Adc12x5New()
- void Adc12x5Delete()
- eStatus Adc12x5ChannelSelect()

- eStatus Adc12x5RangeSelect()
- eStatus Adc12x5ModeSelect()
- eStatus Adc12x5SamplePeriod()
- eStatus Adc12x5InterruptSelect()
- void Adc12x5SoftwareTrigger()
- void Adc12x5Arm()
- void Adc12x5Disable()
- int Adc12x5Running()
- unsigned long int Adc12x5ChannelCount()
- eChannelSampleCount Adc12x5ChannelSampleCount()
- const volatile void *Adc12x5DataPtr()
- unsigned long int Adc12x5ChannelBytes()
- unsigned long int Adc12x5ModuleBytes()
- void Adc12x5PrintVersion()
- void Adc12x5Display()
- void Adc12x5CountsAll()
- eStatus Adc12x5ToVolts()
- void Adc12x5VoltsAll()
- eStatus Adc12x5Volts()
- eStatus Adc12x5Scale()
- eStatus Adc12x5Counts()
- unsigned int Adc12x5Count()

Adc12x5Dma Wrapper Functions

Please see the class Adc12x5Dma discussion above and the header file c_adc12x5dma.h for complete function declarations. The wrapper library provides the following functions:

- int Adc12x5DmaNew()
- void Adc12x5DmaDelete()
- eStatus Adc12x5DmaChannelSelect()
- eStatus Adc12x5DmaRangeSelect()
- eStatus Adc12x5DmaModeSelect()
- void *Adc12x5DmaDataSelect()
- void Adc12x5DmaAutoRearm()
- eStatus Adc12x5DmaSamplePeriod()
- eStatus Adc12x5DmaInterruptSelect()
- void Adc12x5DmaSoftwareTrigger()

- void Adc12x5DmaArm()
- void Adc12x5DmaDisable()
- int Adc12x5DmaRunning()
- unsigned long int Adc12x5DmaChannelCount()
- eChannelSampleCount Adc12x5DmaChannelSampleCount()
- const volatile void *Adc12x5DmaDataPtr()
- unsigned long int Adc12x5DmaChannelBytes()
- unsigned long int Adc12x5DmaModuleBytes()
- void Adc12x5DmaPrintVersion()
- void Adc12x5DmaDisplay()
- eStatus Adc12x5DmaToVolts()
- void Adc12x5DmaGetVolts()
- unsigned int Adc12x5DmaCount()

End.