# Using InstVmeAdc

## Introduction

The InstVmeAdc library provides task-level I/O drivers for the Accelerator Division Instrumentation Department's family of VME waveform digitizer modules. The library is written in C++ for VxWorks and contains classes for configuring and reading the digitizer modules. Class VmeAdcMod supports programmed (software controlled) data reads while class VmeAdcModDma improves data transfer performance by adding support for the VME slot zero controller's block transfer DMA engine. Digitizer module hardware registers are logically grouped according to functionality and accessed through class mutators and accessors. A set of C callable wrapper functions is also provided to give C and VxWorks startup script programmers access to the library.

The two InstVmeAdc C++ digitizer classes will be described in detail and the C language wrapper functions, mapping one-to-one with the class methods, will be outlined. Various example programs coded in C++ demonstrate many of the possible schemes for configuring modules and collecting waveform data.

## Overview

To use the one of the supported digitizer modules the programmer must first read the appropriate header files to obtain module declarations. Establishing a typedef for the module provides a shorthand representation making subsequent code easier to manage and read:

```
#include "adc8x125.h"      // VME 8 Channel 125 MHz ADC Module
#include "vmeadcmod.h"     // generic digitizer with programmed data readout

typedef class InstADC::VmeAdcMod<InstADC::Adc8x125> tADC;
```

Notice that two header files are required to fully declare a digitizer module. In this case the file adc8x125.h contains declarations for the 8 channel 125 MHz digitizer module while vmeadcmod.h contains declarations for a generic digitizer that will represent any of the digitizer modules.

Next an A16 base address for the module's configuration registers and an A32 base address for the module's ADC sample data window must be established. The 16 bit ADC samples are packed 2 per 32–bit word within the data window. The window size is the

product of the number of active channels and the number of samples per trigger. By default all of the module's channels are active. The drivers use 'new with placement' to establish the A16 base address while a call to the ChannelSelect method establishes the A32 address:

```
#include "vmemap.h"       // Map VME Addresses to Processor Local Addresses

const int unsigned kDataChannels = 8;
const int unsigned kDataSamples = 1024;
void * const kAdcA16Base = reinterpret_cast<void * const>( 0x2000 );
void * const kAdcA32Base = reinterpret_cast<void * const>( 0x50000000 );

tADC::eStatusstatus;

tADC         *adcPtr;= new ( VmeA16ToLocal( kAdcA16Base ) ) tADC (
  &status,
  "TestADC",
  0 );

adcPtr->ChannelSelect(
  TADC::kChanMaskAll,
  kAdcA32Base );
```

After establishing the necessary base addresses the module's measurement mode, ADC clock sample period and number of data samples per trigger must be configured:

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

Finally digitizer data is collected:

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

// software triggering with sample period determined by task delays
while ( true ) {
  taskDelay( sysClkRateGet() / 15 );
  adcPtr->SoftwareTrigger();
  while ( adcPtr->IsRunning() ) {// the prudent programmer would add a timeout
      taskDelay( 1 );
  }
  adcPtr->Volts( buffer ); // read data in volts
}
```

This simple example established a software triggered measurement with programmed data readout of floating point values using default values for active channel mask and analog input range.

The example sections found later in this document show how the InstVmeAdc library will support several other triggering and data readout schemes. The library's class constructors and public methods are described in the "VmeAdcMod Class" and "VmeAdcModDma Class" sections below.

**Library Header and Object File Locations**

The InstVmeAdc library is maintained on the node nova.fnal.gov in a CVS repository named instvmeadc. The generic digitizer declarations can be found in:

/home/rfies/esd/rfiinst/inc/vmeadcmod.h
/home/rfies/esd/rfiinst/inc/vmeadcmoddma.h

for C++, and

/home/rfies/esd/rfiinst/inc/c_ vmeadcmod.h
/home/rfies/esd/rfiinst/inc/c_vmeadcmoddma.h

for the C language. Please study these headers for this is where the most detailed method documentation will be found.

The digitizer module specific declarations can be found in:

/home/rfies/esd/rfiinst/inc/adc12x5.h
/home/rfies/esd/rfiinst/inc/adc16x50.h
/home/rfies/esd/rfiinst/inc/adc8x125.h

Please study these module specific headers because they may contain detailed declarations and documentation for methods that extend the generic digitizer to handle unique module hardware capabilities. Other similarly named files may be added to the list as new digitizer modules become available. For each module specific header file there is a matching method parameter declaration file:

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

/home/rfies/esd/rfiinst/inc/adc16x50decls.h
/home/rfies/esd/rfiinst/inc/adc8x125decls.h

Again, to understand the full functionality of a given digitizer module the programmer should study vmeadcmod.h (or vmeadcmoddma.h if using DMA) and the adcXXX.h/adcXXXdecls.h pair for the digitizer module of interest.

The InstVmeAdc library object files may be found in a directory path of the form:

/fecode-bd/vxworks_boot/fe/rfiinst/lib/VERS/TARGET/libinstvmeadc-XXX.out

Where VERS is the VxWorks version (e.g., VW_55, VW_64), TARGET is the target CPU board (e.g., MVME2434, MVME5500) and XXX is the digitizer module identifier (e.g., adc12x5, adc8x125.)

**VmeAdcMod Class**

The VmeAdcMod class provides support for basic hardware configuration, data acquisition complete interrupts and programmed data readout. Please see the header file vmeadcmod.h for detailed method declarations and documentation. The VmeAdcMod class provides the following methods:

• Instantiation::
  - VmeAdcMod::VmeAdcMod constructs a module instance
  - new() specifies the A16 VME address for the module instance
- VmeAdcMod::~VmeAdcMod() deletes a 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
  - InterruptSelect() specifies values for the IRQ status ID and IRQ level registers
  - BaseClockSelect specifies internal or external ADC clock source
  - SoftwareTrigger() software triggers the module when in software trigger (arm) modes
  - SoftwareArm() arms the module when in hardware trigger (arm) modes
  - Disable() disables triggers and interrupts
• Data readout:

- Counts() copies raw digitizer data for all or specified channels to a buffer in user memory
- Volts() reads and converts raw digitizer data for all or specified channels into floating point values in units of volts in user memory
- Scaled() reads and converts raw digitizer data for all or specified channels into floating point values in user memory using the specified range and offset scaling factors

• Utility methods:
- IsRunning() returns true if the digitizer is actively making measurements
- SamplePeriod() returns the digitizer sample period in nSec
- ChannelCount() returns the number of active configured channels
- ChannelMask() returns the channel mask value for the specified channel number
- ChannelDatumCount() 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
- Name() return pointer to the module's name string
- Display() displays the module's register contents on the console
- ModuleID() returns the module's identification information
- Model() returns the module's model identification
- Version() returns the module's version identification
- Revision() returns the module's version revision level identification
- Frequency() returns the module's maximum base clock frequency in MHz
- Channels() returns the module's maximum channel count
- ModuleIDString() returns pointer to a string containing decoded module identification information
- InstanceCount() returns the total number of VmeAdcMod instances

**VmeAdcModDMA Class**

The VmeAdcModDma class builds on the VmeAdcMod class to provide support for DMA data collection with either callback function or message queue acquisition complete notification. Please see the header file vmeadcmoddma.h for detailed method declarations and documentation. The VmeAdcModDma class provides the following methods:

• Instantiation:
- VmeAdcModDma::VmeAdcModDma constructs a module instance

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

• Configuration:

- ChannelSelect() specifies values for the channel mask, sample count and A32 pointer registers
- 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
- InterruptSelect() specifies the hardware interrupt vector and level digitization complete interrupt processing, and a task priority for DMA data collection processing
- AutoRearm() - enables or disables automatic digitization rearm after data collection in hardware triggered (armed) modes
- Disable() disables any pending data read operation and further data collection
- DisableBLT() forces DMA single transfer mode
- EnableBLT() enables DMA block transfer mode

• Utility methods:

- N/A

**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 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 sysClkRateGet()
#include <iv.h>                  // for INUM_TO_IVEC
#include <taskLib.h>             // for taskDelay()

#include "logging.h"             // message logging declarations
#include "vmemap.h"              // Map VME Addresses to Processor Addresses
#include "binarysemaphore.h"     // for simple binary semaphore with timeout

#include “adc8x125.h”            // hardware module specific declarations
#include "vmeadcmod.h"           // high level VME digitizer declarations
```

```
/*
**  module type definitions  **
*/
typedef class InstADC::VmeAdcMod<InstADC::ADC_MODEL> tADC;

/*
**  module constant definitions  **
*/
static const unsigned int kDataInterruptVector = 80;
static const bool kDisplayErrors = true;

static const int unsigned kDataChannels = tADC::kNumChannels;
static const int unsigned kDataSamples = 1024;

static void * const kAdcA16Base = reinterpret_cast<void * const>( 0x2000 );
static void * const kAdcA32Base = reinterpret_cast<void * const>( 0x50000000 );

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

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

  // create ISR<-->task 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
  tADC::eStatus     status;
  tADC       *adcPtr = new ( VmeA16ToLocal( kAdcA16Base ) ) tADC(
    &status,
    "ADC00",
    0 );

  if ( tADC::kOk != status ) {
    LogError( "DataReadoutTask() - Could not create ADC!\n", 0,0,0,0,0,0 );
    return;
  }
```

```
  status = adcPtr->ChannelSelect(
    tADC::kChanMaskAll,
    kAdcA32Base );

  status = adcPtr->RangeSelect(
    tADC::kChanMaskAll,
    tADC::kRangeDefault );

  // Mode 5 - Ns samples, at Sr sample rate, software arm
  status = adcPtr->ModeSelect(
    tADC::kMultSampScopeSwArm,
    tADC::kSamplePeriodMin,
    tADC::eChannelSampleCount( kDataSamples ) );

  // create interrupt service routine
  if ( OK == intConnect(
               INUM_TO_IVEC( kDataInterruptVector ),
              VOIDFUNCPTR( DataReadoutISR ),
              int( semPtr ) ) ) {
    status = adcPtr->InterruptSelect( kDataInterruptVector, tADC::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() );// ~1 Hz acquisition
    adcPtr->SoftwareTrigger();
    semStatus = semPtr->Take( Semaphore::kWaitForever, kDisplayErrors );
    adcPtr->Volts( buffer );            // get data in volts
    LogMessage( "DataReadoutTask() got fresh data.\n" );
  }

}
/*
*/
```

The above working example is maintained on the node nova.fnal.gov in a file named example1.cpp within the CVS repository named instvmeadcexample.

**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 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 sysClkRateGet()
#include <iv.h>                  // for INUM_TO_IVEC
#include <taskLib.h>             // for taskDelay()

#include "logging.h"             // message logging declarations
#include "vmemap.h"              // Map VME Addresses to Processor Addresses
#include "binarysemaphore.h"     // for simple binary semaphore with timeout

#include “adc8x125.h”            // hardware module specific declarations
#include "vmeadcmod.h"           // high level VME digitizer declarations

/*
**  module type definitions  **
*/
typedef class InstADC::VmeAdcMod<InstADC::ADC_MODEL> tADC;

/*
**  module constant definitions  **
*/
static const unsigned int kDataInterruptVector = 80;
static const bool kDisplayErrors = true;

static const int unsigned kDataChannels = tADC::kNumChannels;
static const int unsigned kDataSamples = 1024;

static void * const kAdcA16Base = reinterpret_cast<void * const>( 0x2000 );
static void * const kAdcA32Base = reinterpret_cast<void * const>( 0x50000000 );

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

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

  // create ISR<-->task 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
  tADC::eStatus     status;
  tADC       *adcPtr = new ( VmeA16ToLocal( kAdcA16Base ) ) tADC(
    &status,
    "ADC00",
    0 );

  if ( tADC::kOk != status ) {
    LogError( "DataReadoutTask() - Could not create ADC!\n", 0,0,0,0,0,0 );
    return;
  }

  status = adcPtr->ChannelSelect(
    tADC::kChanMaskAll,
    kAdcA32Base );

  status = adcPtr->RangeSelect(
    tADC::kChanMaskAll,
    tADC::kRangeDefault );

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

  // create interrupt service routine
  if ( OK == intConnect(
               INUM_TO_IVEC( kDataInterruptVector ),
              VOIDFUNCPTR( DataReadoutISR ),
              int( semPtr ) ) ) {
    status = adcPtr->InterruptSelect( kDataInterruptVector, tADC::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() );// ~1 Hz acquisition
    adcPtr->SoftwareArm();       // arm the digitizer for the next trigger
    semStatus = semPtr->Take( Semaphore::kWaitForever, kDisplayErrors );
    adcPtr->Volts( buffer );            // read data in volts
    LogMessage( "DataReadoutTask() got fresh data.\n" );
  }

}
```

```
/*
*/
```

The above working example is maintained on the node nova.fnal.gov in a file named example2.cpp within the CVS repository named instvmeadcexample.

**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 data acquisition complete interrupt. Interrupt processing is handled within the InstVmeAdcDma class. The data readout task takes the semaphore and requests the data in floating point form. The data are read by DMA into a holding buffer within the InstVmeAdcDma class. The holding buffer is then converted to floating point values in user memory.

```
/*
**  module include files  **
*/
#include "logging.h"             // message logging declarations
#include "vmemap.h"              // Map VME Addresses to Processor Addresses
#include "binarysemaphore.h"     // for simple binary semaphore with timeout

#include “adc8x125.h”            // hardware module specific declarations
#include "vmeadcmoddma.h"        // high level VME digitizer declarations

/*
**  module type definitions  **
*/
typedef class InstADC::VmeAdcModDma<InstADC::ADC_MODEL> tADC;

/*
**  module constant definitions  **
*/
static const int unsigned kDataInterruptVector = 80;
static const int unsigned kReadoutTaskPriority = 33;
static bool const kNasty = false;
static const bool kDisplayErrors = true;

static const int unsigned kDataChannels = tADC::kNumChannels;
static const int unsigned kDataSamples = 1024;

static void * const kAdcA16Base = reinterpret_cast<void * const>( 0x2000 );
static void * const kAdcA32Base = reinterpret_cast<void * const>( 0x50000000 );

/*
// Data collection complete callback function.
```

```
// Informs data readout task of the arrival of new data via a semaphore.
*/
static void Callback( tADC::eStatus const status, void * const paramPtr ) {
  class Semaphore *semPtr = static_cast<class Semaphore *>( paramPtr );
  semPtr->Give( kDisplayErrors );
  return;
}
/*
*/

/*
// Data readout task.
// Waits for callback function to announce
// data arrival and then reads data to local memory.
*/
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;
  }

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

  if ( tADC::kOk != status ) {
    LogError( "DataReadoutTask() - Could not create ADC!\n", 0,0,0,0,0,0 );
    return;
  }

  status = adcPtr->ChannelSelect(
    tADC::kChanMaskAll,
    kAdcA32Base );

  status = adcPtr->RangeSelect(
    tADC::kChanMaskAll,
    tADC::kRangeDefault );

  status = adcPtr->ModeSelect(
    tADC::kMultSampScopeExtArm,
    tADC::kSamplePeriodMin,
    tADC::eChannelSampleCount( kDataSamples ) );

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

```
  status = adcPtr->InterruptSelect( kDataInterruptVector,
    tADC::kIntLevel6,
    kReadoutTaskPriority );

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

  static float      buffer[ kDataChannels ][ kDataSamples ];

  while ( true ) {
    semStatus = semaphore.Take( Semaphore::kWaitForever, kDisplayErrors );
    adcPtr->Volts( buffer );
    LogMessage( "DataReadoutTask() got fresh data.\n" );
  }

}
/*
*/
```

The above working example is maintained on the node nova.fnal.gov in a file named example3.cpp within the CVS repository named instvmeadcexample.

**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 and defines a message that the InstVmeAdcDma class places in the queue when data acquisition is complete. All interrupt processing is handled by the InstVmeAdcDma class. The data readout task receives the queued message and requests the data in floating point form. The data are read into a holding buffer within the InstVmeAdcDma 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 within the data acquisition loop. This feature is useful in cases where the external trigger signal rate may be 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 Addresses
```

```
#include “adc8x125.h”            // hardware module specific declarations
#include "vmeadcmoddma.h"        // high level VME digitizer declarations

/*
**  module type definitions  **
*/
typedef class InstADC::VmeAdcModDma<InstADC::ADC_MODEL> tADC;

/*
**  module constant definitions  **
*/
static const int unsigned kDataInterruptVector = 80;
static const int unsigned kReadoutTaskPriority = 33;
static bool const kNice = true;
static const bool kDisplayErrors = true;
static const int unsigned kQueueDepth = 2;

static const int unsigned kDataChannels = tADC::kNumChannels;
static const int unsigned kDataSamples = 1024;

static void * const kAdcA16Base = reinterpret_cast<void * const>( 0x2000 );
static void * const kAdcA32Base = reinterpret_cast<void * const>( 0x50000000 );
/*
// Data readout task.
// Waits for message queue to announce
// data arrival and then reads data to local memory.
*/
static void DataReadoutTask( void * ) {

  tADC::eStatus     status;
  tADC       *adcPtr = new ( VmeA16ToLocal( kAdcA16Base ) ) tADC(
    &status,
    "ADC00",
    kNice,
    0 );

  if ( tADC::kOk != status ) {
    LogError( "DataReadoutTask() - Could not create ADC!\n", 0,0,0,0,0,0 );
    return;
  }

  status = adcPtr->ChannelSelect(
    tADC::kChanMaskAll,
    kAdcA32Base );

  status = adcPtr->RangeSelect(
    tADC::kChanMaskAll,
    tADC::kRangeDefault );

  status = adcPtr->ModeSelect(
    tADC::kMultSampScopeExtArm,
    tADC::kSamplePeriodMin,
    tADC::eChannelSampleCount( kDataSamples ) );
```

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

  // DAQ complete message queue
  ::eStatus queueStatus;
  class MessageQueue<tADC::DaqCompleteMessage>
    completedQueue( &queueStatus, kQueueDepth );

  const int  kBoard1 = 1;
  void       *rawPtr;
  // NULL tells class to allocate a raw buffer for us
  rawPtr = adcPtr->DataSelect(
    NULL,
    &completedQueue,
    &kBoard1 );

  status = adcPtr->InterruptSelect(
    kDataInterruptVector,
    tADC::kIntLevel1,
    kReadoutTaskPriority );

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

  while ( true ) {
    taskDelay( sysClkRateGet() );// ~1 Hz acquisition
    adcPtr->SoftwareArm();       // rearm the digitizer
    completedQueue.Receive(
      message,
      MessageQueue<tADC::DaqCompleteMessage>::eWait( 100 ),
      kDisplayErrors );
      if ( tADC::kOk == message.status ) {
       adcPtr->GetVolts( rawPtr, buffer, tADC::kChanMaskAll );
        LogMessage( "DataReadoutTask() got fresh data.\n" );
      } else {
       LogError( "DataReadoutTask() - DAQ failed, status = %d, board = %d\n",
                 message.status,
                 *static_cast<int *>( message.messagePtr ), 0, 0, 0, 0 );
      }
  }

}
```

The above working example is maintained on the node nova.fnal.gov in a file named example4.cpp within the CVS repository named instvmeadcexample.

**Example 5 – Use With Two Modules of Differing Model**

This example shows how to use two ADC modules of different model type in one source code file. The key is to read the declarations and create a typedef for the first ADC module model and then do the same for the second ADC module model.

```
/*
**  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

/*
**  module constant definitions  **
*/
static void * const kAdc1Base = reinterpret_cast<void * const>( 0x2000 );
static void * const kAdc2Base = reinterpret_cast<void * const>( 0x3000 );

static void * const kAdc1Data = reinterpret_cast<void * const>( 0x50000000 );
static void * const kAdc2Data = reinterpret_cast<void * const>( 0x50010000 );

static const int unsigned kDataChannels = 2;
static const int unsigned kDataSamples = 1024;

// read declarations and create typedef for 12 channel 5 MHz digitizer
#include "adc12x5.h"             // hardware module specific declarations
#include "vmeadcmod.h"           // high level VME digitizer declarations
typedef class InstADC::VmeAdcMod<InstADC::ADC_MODEL> tADC1;

// read declarations and create typedef for 8 channel 125 MHz digitizer
#include "adc8x125.h"            // hardware module specific declarations
#include "vmeadcmod.h"           // high level VME digitizer declarations
typedef class InstADC::VmeAdcMod<InstADC::ADC_MODEL> tADC2;

/*
// Data readout task.
// Software triggers the digitizers, polls digitizer status until
// digitization is complete and then reads data to local memory.
*/
static void DataReadoutTask( void * ) {

  tADC1::eStatus    status1;

  tADC1             *adc1Ptr = new ( VmeA16ToLocal( kAdc1Base ) ) tADC1(
    &status1,
    "ADC01",
    0 );
```

```
  if ( tADC1::kOk != status1 ) {
    LogError( "DataReadoutTask() - Could not create ADC #1!\n", 0,0,0,0,0,0 );
    return;
  }

  status1 = adc1Ptr->ChannelSelect(
    tADC1::kChanMask0,
    kAdc1Data );

  status1 = adc1Ptr->ModeSelect(
    tADC1::kMultSampScopeSwArm,
    tADC1::kSamplePeriodMin,
    tADC1::eChannelSampleCount( kDataSamples ) );

  //adc1Ptr->Display( true );


  tADC2::eStatus    status2;

  tADC2             *adc2Ptr = new ( VmeA16ToLocal( kAdc2Base ) ) tADC2(
    &status2,
    "ADC02",
    0 );

  if ( tADC2::kOk != status2 ) {
    LogError( "DataReadoutTask() - Could not create ADC #2!\n", 0,0,0,0,0,0 );
    return;
  }

  status2 = adc2Ptr->ChannelSelect(
    tADC2::kChanMask0,
    kAdc2Data );

  status2 = adc2Ptr->ModeSelect(
    tADC2::kMultSampScopeSwArm,
    tADC2::kSamplePeriodMin,
    tADC2::eChannelSampleCount( kDataSamples ) );

  //adc2Ptr->Display(true );

  static float             buffer[ kDataChannels ][ kDataSamples ];

  // software triggering with sample period determined by task delays
  while ( true ) {
    taskDelay( sysClkRateGet() / 2 );   // ~2 Hz acquisition
    adc1Ptr->SoftwareTrigger();
    adc2Ptr->SoftwareTrigger();
    while ( adc1Ptr->IsRunning() || adc2Ptr->IsRunning() ) {
      // the prudent programmer would add a timeout
      taskDelay( 1 );
    }
    adc1Ptr->Volts( &buffer[ 0 ] ); // read data in volts
    adc2Ptr->Volts( &buffer[ 1 ] ); // read data in volts
    LogMessage( "DataReadoutTask() got fresh data.\n" );
```

```
  }
}
```

The above working example is maintained on the node nova.fnal.gov in a file named example5.cpp within the CVS repository named instvmeadcexample.

## C Language Wrappers

A set of wrapper functions provides full functionality for C programmers. Wrappers are provided for both the VmeAdcMod and the VmeAdcModDma classes. In each case 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 associated class methods.

## InstVmeAdc Wrapper Functions

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

- int VmeAdcMod_New()
- void VmeAdcMod_Delete()
- eStatus VmeAdcMod_ChannelSelect()
- eStatus VmeAdcMod_RangeSelect()
- eStatus VmeAdcMod_ModeSelect()
- eStatus VmeAdcMod_InterruptSelect()
- void VmeAdcMod_SoftwareTrigger()
- void VmeAdcMod_SoftwareArm()
- void VmeAdcMod_Disable()
- void VmeAdcMod_CountsAll()
- eStatus VmeAdcMod_Counts()
- void VmeAdcMod_VoltsAll()
- eStatus VmeAdcMod_Volts()
- eStatus VmeAdcMod_ScaleAll()
- eStatus VmeAdcMod_Scale()
- int VmeAdcMod_IsRunning()
- eStatus VmeAdcMod_SamplePeriod()
- unsigned long int VmeAdcMod_ChannelCount()
- eChannelSampleCount VmeAdcMod_ChannelSampleCount()

- const volatile void *VmeAdcMod_DataPtr()
- unsigned long int VmeAdcMod_ChannelBytes()
- unsigned long int VmeAdcMod_ModuleBytes()
- void VmeAdcMod_Name()
- void VmeAdcMod_Display()
- void VmeAdcMod_ModuleID()
- unsigned int VmeAdcMod_InstanceCount()

**VmeAdcModDma Wrapper Functions**

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

- int VmeAdcModDma_New()
- void VmeAdcModDma_Delete()
- eStatus VmeAdcModDma_ChannelSelect()
- void *VmeAdcModDma_DataSelect()
- eStatus VmeAdcModDma_RangeSelect()
- eStatus VmeAdcModDma_ModeSelect()
- eStatus VmeAdcModDma_InterruptSelect()
- void VmeAdcModDma_AutoRearm()
- void VmeAdcModDma_SoftwareTrigger()
- void VmeAdcModDma_SoftwareArm()
- void VmeAdcModDma_Disable()
- void VmeAdcModDma_VoltsAll()
- eStatus VmeAdcModDma_Volts()
- eStatus VmeAdcModDma_ScaleAll()
- eStatus VmeAdcModDma_Scale()
- int VmeAdcModDma_IsRunning()
- eStatus VmeAdcModDma_SamplePeriod()
- unsigned long int VmeAdcModDma_ChannelCount()
- eChannelSampleCount VmeAdcModDma_ChannelSampleCount()
- const volatile void *VmeAdcModDma_DataPtr()
- unsigned long int VmeAdcModDma_ChannelBytes()
- unsigned long int VmeAdcModDma_ModuleBytes()
- const char *VmeAdcModDma_Name()
- void VmeAdcModDma_Display()

- void VmeAdcModDma_ModuleID()
- unsigned int VmeAdcModDma_InstanceCount()

End.