Using ClassACNet

Introduction

The ClassACNet library provides an object based interface between front-end data structures and the global accelerator control system commonly referred to as ACNet. ClassACNet is a C++ namespace containing the definitions of two classes that form the basis for communication with ACNet: 1) class Portal which is layered on top of the MOOC and ACNet libraries to provide full-featured MOOC operability, and 2) class ObjAccessor which ties front-end objects to Portal instances. Class ObjAccessor is an abstract class and must be subclassed, so the ClassACNet library contains a collection of off-the-shelf accessor classes that address most common data access needs. Accessors are included for reading and setting memory based data in situ, for instantiating and manipulating scalars, vectors & matrices, for calling user provided C-style callback functions and for calling the accessor and mutator methods of user provided classes. In cases where special processing is required the user may want to create custom ObjAccessor derived classes. In that event the ClassACNet library provides a rich set of examples.

The file ClassACNetOverview.ppt contains a series of slides outlining the material presented in this note along with other more detailed information.

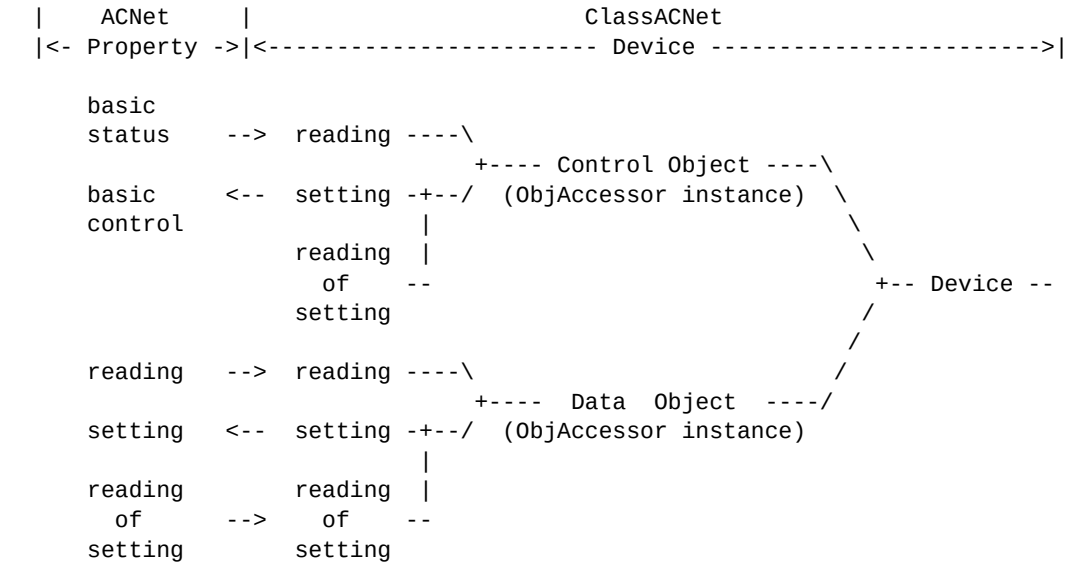
Overview

To attach front-end objects to ACNet the front-end user instantiates a Portal with the desired operating attributes and then instantiates one or more accessors that are connected to user objects. Upon instantiation ObjAccessors automatically attach themselves to the most recently instantiated Portal so the user code can look something like the following:

```
(void) new Portal(…)       // slow devices
(void) new Accessor(…)
(void) new Accessor(…)
          :
(void) new Portal(…)       // fast devices
(void) new Accessor(…)
(void) new Accessor(…)
          :
```

As indicated it is possible to instantiate multiple Portals with multiple accessors per portal.

ACNet devices have optional basic status, basic control, reading, setting and reading of setting properties. In ClassACNet a fully implemented 'Device' is composed of a pair of ObjAccessor instances -- one for status/control: the 'Control Object', and another for reading/setting: the 'Data Object'. The mapping from ACNet device property to ClassACNet device is depicted below:



Data and control objects each have a reading that can be read and a setting that can be both set and read. For any given device the control and data objects are optional so it is possible to have a control only or a data only device. For any given control or data object the reading and setting are optional so it is possible to have a reading and setting, reading only or setting only object. Devices may also support an optional fast read method that provides optimized access to object readings for fast plotting. ClassACNet devices map one-to-one feature-wise to ACNet devices. Note that the full ClassACNet device symmetry is partially lost because the ACNet device model does not support reading the basic control property.

Class Portal

The Portal class encapsulates the functionality of a MOOC class supporting the five basic data properties, fast plotting and alarms and limits. Portals also support double buffering of data if necessary to alleviate buffer alignment issues, and MOOC_WHACK_FORWARD if the user needs to modify incoming setting values. Normally the user need only instantiate the Portal class – there are no other required

methods. Class Portal's constructor defines the MOOC features that will be supported by all accessors attached to a given portal. The Portal class constructor prototype is:

```
Portal( ePortalStatus * const statusPtr,
        char const * const namePtr, unsigned int const portalID,
        unsigned int const maxAlarms = kNumDevices,
        bool const fastPlotEnable = k720HzPlot,
        unsigned int const maxPlots = kNumDevices,
        eDiagnosticControl const diagnosticControl = kDiagnosticDisable );
```

The constructor parameters are defined as follows:

• statusPtr – pointer to constructor status destination – a nonzero status value indicates an error condition

• namePtr – pointer to a text string that uniquely identifies this instance for use in diagnostic messages

• portalID – integer portal identifier identical in value to the second 16 bit word of the associated ACNet device's SSDN

• maxAlarms – integer number of alarm scans to be handled by this portal – the default is 256 – use of smaller values is recommended to save on memory and processing overhead

• fastPlotEnable – boolean true selects 720 Hz fast plot mode – false selects 15 Hz mode

• maxPlots - integer number of fast-time plots to be handled by this portal – the default is 256 – use of smaller values is recommended to save on memory and processing overhead

• diagnosticControl – nonzero values enable diagnostic messages – various options enable general messages, ACNet setting messages or ACNet reading messages.

The Portal class supports an option to allow the user to invoke the periodic fast plot data sampling handler. By default the Portal instance periodically samples via the UCD auxiliary interrupt. The Portal class' static method SetFastSamping(bool const enable) should be called prior to Portal instantiation to flag the user's intent to periodically call the fast data sampling routine ClassACNet::FtpHandler(class Portal * const portalPtr). A call to the SetFastSamping method affects all subsequent Portal instantiations. Note that when calling the FtpHandler method for a given Portal at interrupt context the user is responsible for saving and restoring the floating point context if any accessor attached to that Portal includes floating point code in its FastRead method.

Please see the file classacnet.h for detailed examples, definitions and declarations for the class Portal constructor and other methods.

Class ObjAccessor

Class ObjAccessor defines the interface between one or more user data objects and a Portal. As mentioned above ObjAccessor is an abstract class that cannot be directly instantiated. The casual user does not need to understand all ObjAccessor constructor parameters, however, several of the parameters appear in the constructors of derived classes so familiarity will help gain a deeper understanding of accessor operation. The ObjAccessor class constructor prototype is:

```
ObjAccessor( ePortalStatus * const statusPtr,
             char const * const namePtr, unsigned int const deviceID,
             bool const controlObject = false,
             unsigned long int const maxReadBuffer = 0,
             unsigned long int const maxSetBuffer = 0,
             eDiagnosticControl const diagnosticControl = kDiagOff );
```

The constructor parameters are defined as follows:

• statusPtr – pointer to constructor status destination – a nonzero status value indicates an error condition

• namePtr – pointer to a text string that uniquely identifies this instance for use in diagnostic messages

• deviceID – integer device identifier identical in value to the low byte of the first 16 bit word of the associated ACNet device's SSDN

• controlObject – boolean true indicates that this accessor connects to the basic status/control of the device – false indicates reading/setting

• maxReadBuffer – integer number of bytes required to double buffer the largest possible reading addressed by this accessor – if zero double buffering is disabled

• maxSetBuffer - integer number of bytes required to double buffer the largest possible setting addressed by this accessor – if zero double buffering is disabled

• diagnosticControl – nonzero values allow diagnostic messages – possible values enable general messages, ACNet setting messages or ACNet reading messages.

The ObjAccessor class supports an offset multiplier for use when the user data will be addressed as a large byte array via length and offset. The offset multiplier default value is one and may be modified by calling OffsetScale(unsigned long int const scaleFactor) after accessor instantiation. ACNet offsets are multiplied by the offset multiplier to allow the range limited offset value to address large data 'pages'.

The double buffering feature is intended for use when the user data contains floating point values which will be copied with floating point instructions. This is a concern

because the MOOC/ACNet message buffers are not always long word aligned. If the selected accessor class will not try to pass values as floating point (e.g., the bcopy based in situ accessors) then the associated maxXxxBuffer parameter values should be set to zero.

Please see the file classacnet.h for detailed examples, definitions and declarations for the class ObjAccessor constructor and other methods.

Fast Plotting Support

The ACNet continuous and snapshot plot protocols are supported by cooperative code elements in the ObjAccessor and Portal classes. If a given device has a data object then its accessor will be used for fast plotting otherwise the control object's accessor will be used. To enable fast plotting the FastReadByteCount method of the accessor must return a nonzero value. If the accessor does not enable fast plotting in this manner a MOOC fast read not supported error will be returned at plot setup time. When fast plotting is enabled the accessor's FastRead method will be called at the fast plot data collection rate. If the accessor does not provide a FastRead method the ObjAccessor base class will call the normal Read method if available, otherwise it will call the ReadSetting method.

The ACNet SSDN

The ACNet SSDN must be configured properly to address user accessors. Five special fields are defined within 8 byte SSDN as follows:

- portalID: SSDNHX property (0023/**0021**/0000/0124)
- deviceID: SSDNHX property (00**23**/0021/0000/0124)
- deviceChan: SSDNHX property (0023/0021/0000/01**24**)
- deviceType: SSDNHX property (0023/0021/0000/**01**24)
- misc: SSDNHX property (0023/0021/**0000**/0124)

• The portalID field identifies the Portal instance number and must match the portalID parameter of the Portal constructor.

• The deviceID field identifies the device instance number attached to the given portal and must match the deviceID parameter of the ObjAccessor constructor.

• The deviceChan field is used with the proper deviceType value to address user data by channel number.

• The deviceType field determines how deviceChannel and length & offset will be interpreted to address user data as follows:

- kArrayDevice (0) – length, offset and data byte count are used to calculate an index and count to address one of n identically sized data elements.

- kChannelDevice (1) – the deviceChan field is used to address one of n potentially different sized data elements. Length must equal the data element size and offset must be zero.
- kLengthOffsetDevice (2) – length, offset and the offset scale factor are used as count and index to address a portion of the data elements treated as an array of char: data[offset * offsetScale] for length bytes.

• The misc field is a 16 bit value that is passed to all read and set methods as user defined extra information.

Class ObjAccessor Derivatives

The first and most general accessor type is the 'in situ accessor' which reads/sets user data where it resides in memory. Use of this group of accessors is limited to cases where ACNet settings or readings simply need to be transferred to or from the front-end's memory with no special synchronization or control flow requirements. This is ideal for reading slowly updating data values or setting parameters that will be picked-up as needed by the front-end software. In order to optimize code space the in situ support is provided by several classes that are tailored to the nature of the user data:

• ReadSetInSitu – supports a reading, setting and reading of setting for scalars, vectors or matrices

• ReadInSitu – supports reading only for scalars, vectors or matrices

• SetInSitu - – supports setting and reading of setting only for scalars, vectors or matrices

• WriteOnlyInSitu - – supports setting and reading of setting (image) only for scalars, vectors or matrices when the target resides in write only hardware

• ReadBndsSetInSitu – supports a reading, setting and reading of setting for scalars, vectors or matrices with setting bounds checking

• BndsSetInSitu - – supports setting and reading of setting only for scalars, vectors or matrices with setting bounds checking

• BndsWriteOnlyInSitu - – supports unchecked setting and reading of setting (image) only for scalars, vectors or matrices with setting bounds checking when the target resides in write only hardware

Other standard derived accessor classes include:

• classacnetobject.h – instantiate and access scalars

• classacnetvector.h – instantiate and access vectors (2D arrays)

• classacnetmatrix.h – instantiate and access matrices (3D arrays)

• classacnetconstruction.h – reading via object construction

• classacnetfunction.h – call user provided reading, setting and reading of setting C style callback functions

• classacnetmethod.h – call user class provided accessor and mutator methods

• classacnetstring.h – read 2D character arrays (array of C strings) properly byte swapped

• classacnetfiledirectory.h – construct and read a 2D array of characters (array of C strings) containing the specified file names in the specified front-end file system directory, properly byte swapped

Please see the various header files identified above for detailed examples, definitions and declarations for the various forms of these accessors.

Some Examples

The following code examples not only show how to instantiate many of the standard accessors, but also provide a look at the structure of typical ClassACNet based front-end programs. In many cases the user simply instantiates a Portal and then a series of accessors.

• In Situ Accessor Example:

```
#include "classacnetinsitu.h"  // ClassACNet stuff

using namespace ClassACNet;

void Foo( void ) {

  typedef enum {  // data type & legal setting range
    kFooMin = -100,
    kFooMax = 100
  } tFooType;

  const unsigned long int kRows = 5;
  const unsigned long int kCols = 5;

  float         reading[kRows][kCols];
  tFooType        setting[kRows][kCols];
  ePortalStatus   status;

  class Portal
    myPortal( &status, "MyPort", 0x0020, 10, k720HzPlot, 10, kDiagOff );
        :

  // data device with buffering alignment critical
  // because of the use of a float reading type
  // supports matrix reading and setting with setting limits
  class ReadBndsSetInSitu<float, tFooType, kRows, kCols>
    myAccessor( &status, "m:xxTEST", 0x0001, kData, kAlign,
      &reading[0][0],
      &setting[0][0], kFooMin, kFooMax,
      kDiagOff );
        :

  while( true ) {      // do front-end processing
        :
  }

}
```

• Matrix Object Accessor Example:

```
#include "classacnetmatrix.h"  // ClassACNet stuff

using namespace ClassACNet;

void Foo( void ) {

  const unsigned long int kRows = 5;
  const unsigned long int kCols = 5;

  ePortalStatus status;
  float         reading = 123.0f;
  int           setting = 456;

  class Portal
    myPortal( &status, "MyPort", 0x0020, 10, k720HzPlot, 10, kDiagOff );
        :
  ReadSetMatrix<float,int, kRows, kCols>
    object3( &status, "Test", 0x0000, kData, kAlign, kDiagOff );
  object3[ 4 ][ 4 ].reading = reading;     // 2D array set reading
  object3[ 4 ][ 4 ].setting = setting;     // 2D array set setting
  reading = object3[ 4 ][ 4 ].reading;     // 2D array get reading
  setting = object3[ 4 ][ 4 ].setting;     // 2D array get setting

  ReadMatrix<float, kRows, kCols>
    object1( &status, "Test", 0x0001, kData, kAlign, kDiagOff );
  object1[ 2 ][ 2 ] = reading;             // 2D array set reading
  reading = object1[ 2 ][ 2 ];             // 2D array get reading

  SetMatrix<int, kRows, kCols>
    object5( &status, "Test", 0x0002, kData, kNoAlign, kDiagOff );
  object5[ 3 ][ 3 ] = setting;             // 2D array set setting
  setting = object5[ 3 ][ 3 ];             // 2D array get setting

 ReadBndsSetMatrix<float,int, kRows, kCols>
    object0( &status, "Test", 0x0003, kData, kAlign, 0, 999, kDiagOff );
  object0[ 1 ][ 1 ].reading = reading;     // 2D array set reading
  object0[ 1 ][ 1 ].setting = setting;     // 2D array set setting
  reading = object0[ 1 ][ 1 ].reading;     // 2D array get reading
  setting = object0[ 1 ][ 1 ].setting;     // 2D array get setting

  BndsSetMatrix<int, kRows, kCols>
    object2( &status, "Test", 0x0004, kData, kNoAlign, 0, 999, kDiagOff );
  object2[ 3 ][ 3 ] = setting;             // 2D array set setting
  setting = object2[ 3 ][ 3 ];             // 2D array get setting
     :
  return;

}
```

• Constructor Accessor Example:

```
#include "classacnetconstruction.h"  // object construction accessor stuff

   class RawMultiturn {    // raw measurement data
              :
   }

   class Orbit {
     public:

       // constructor
       Orbit( class RawMultiturn const &rawData );
       // NOTE: destructor will never be called!

       // new with placement operator
       inline void operator new( size_t size, voidp ) {
         return( p );
       }
              :
   }


   using namespace ClassACNet;

   ePortalStatus    status;

   static RawMultiturn rawData( ... );
                  :

   class Portal
     myPortal( &status, "MyPort", 0x0020, 10, k720HzPlot, 10, kDiagOff );
            :
   // build Orbit data from most recent raw data
   class ObjConstructor<class Orbit, class RawMultiturn>
     myAccessor( &status,
                 "m:TEST", 0x0002, kData, kAlign,
                 rawData,
                 kNoDiagnostic );
                   :
```

• Method Accessor Example:

```
class Foo {
  public:
    typedef enum {  // data type & legal setting range
      kFooMin = -100,
      kFooMax = 100
    } tFooType;

    // accessors
    int Read( float *destPtr, unsigned long int const misc ) const {
      *destPtr = _reading; return( ClassACNet::kDeviceOk );
    };
    int Get( tFooType *destPtr, unsigned long int const misc ) const {
      *destPtr = _setting;
      return( ClassACNet::kDeviceOk );
    };

    // mutator
    int Set( tFooType const &value, unsigned long int const misc ) {
      _setting = value;
      return( ClassACNet::kDeviceOk );
    };

  private:
    float       _reading;
    tFooType    _setting;
};
        :

#include "classacnetmethod.h"  // ClassAcnet stuff

using namespace ClassACNet;

void Bar( void ) {

  class Foo        myFoo;
  ePortalStatus    status;

  class Portal
    myPortal( &status, "MyPort", 0x0020, 10, k720HzPlot, 10, kDiagOff );
        :
  // data device with critical buffering alignment due to use of float type
  // supports reading and setting with setting limits
  ReadBndsSetMethod<class Foo, float, Foo::tFooType>
    myAccessor( &status, "m:xxTEST", 0x0001, kData, kAlign,
      &myFoo,
      &Foo::Read,
      &Foo::Get, &Foo::Set, Foo::kFooMin, Foo::kFooMax, kDiagOff );
        :
  return;

}
```

ClassACNet Remote

The file classacnetremote.h contains declarations for classes that are designed to access ACNet devices in remote ACNet nodes. Three classes are defined:

- class ACNetState - supports setting the state device associated with the current front-end
- class ACNetRead – supports reading ACNet devices in remote nodes
- class ACNetSet – supports setting ACNet devices in remote nodes

Please see the file classacnetremote.h for detailed examples, definitions and declarations for the various forms of these accessors.

ClassACNet Model

The file classacnetmodel.cpp contains stub code for implementing a full-featured ClassACNet accessor. The model code appears to be quite lengthy, but recall that it is a subclass of ObjAccessor so any of the method stubs that are not required in the user's application may simply be deleted! The base class will 'cover for' any methods that are not provided by the user. For example if the user does not provide a FastRead method the base class will call the Read method in its place. So in this case the user can have fast plotting support simply by providing the FastReadByteCount method to tell the ClassACNet Port that fast plotting is desired without writing any special fast plot code.

ClassACNet Examples

The file classacnetexample.cpp contains examples of the use of various standard accessors and also the remote ACNet classes. Also each of the accessor header files begin with an example that uses the code contained in the header.