Abstract:

The base classes presented here provide an encapsulation of DDS and DDS mechanisms allowing not only Command/Response Data patterns, but also other such as Alarms, or Status (periodic or aperiodic). They are intended to speed up the DDS learning curve allowing a developer to become more productive quickly. They also factor out much of the common logic and mechanisms required by DDS[1].

DDS Entities Wrapper Classes:

This document is intended to describe the purpose, use, and APIs for the DDS Entities (ddsEnties.h/.hpp/.py) bases classes.

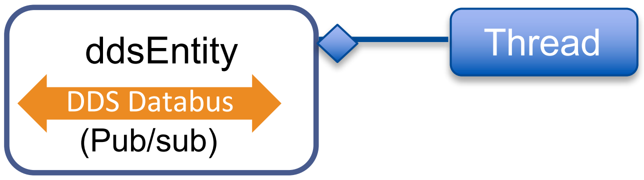

Figure 1 - Simplified ddsEntity base class composition

Purpose and Scope:

The DDS Entities (ddsEntities.h./.hpp/.py) base classes encapsulate common DDS reader and writer setup and configuration, simplifying the user application and initial detail knowledge of DDS APIs. This allows a friendlier and more productive DDS ramp-up in getting an application up and running while factoring out common DDS code. Currently, these base classes support the following three language bindings and DDS core mechanisms.

1. c++98 - Compiled Types, programmatic entity instantiation, Waitsets, reader thread, periodic writers, optional writer statuses thread or statuses callbacks, content filtering, and suppression of self-written topics the application has also subscribed (e.g., command_ack).

2. c++11 - Dynamic Types, XML Application Creation, Waitsets, reader thread, periodic writers, optional writer status thread or statuses call back, content filtering, and suppression of self-written topics the application has also subscribed to (e.g., command_ack).

3. Python - Dynamic Types, XML Application Creation, Waitsets, reader thread, periodic writers, optional writer status thread or statuses call back, content filtering, and suppression of self-written topics the application has also subscribed (e.g., command_ack)

[1] Because they are implemented as inheritance vs. composition, the code is still replicated by the compiler. Never-the-less there is only one instance of the logic (and one place to fix bugs)

By creating topic objects that inherit from one of the two base classes (reader or writer) and overloading the respective handler() and writer() functions, the user can easily create application-specific topics. The user topic classes then handle the read data in the context of the application topics and may be extended for application-specific functionality and properties as required.

Example, to create a writer topic in each of the supported language bindings at the application level would be done as follows (subsequent topics would be done immediately after, in like fashion):

1. c++98 - `DeviceStateWtr device_state_writer(participant, publisher);`

2. c++11 - `DeviceStateWtr device_state_writer(participant);`

3. Python - `device_dsw = topics.DeviceStateWtr(participant)`

APIs - Presented In Python - however, very similar for all three supported language bindings.

`Class Writer(threading.Thread)`:

Purpose:

The writer base class primarily provides access to a default write to write the data sample. It intended the user override the write() function to modify specific sample parameters and then write then write a sample in the context of an application.

Optionally, the writer base class allows a topic to be sent periodically and/or monitor writer statuses (via waitset or listener). Currently the only statuses monitored are the `PUBLICATION_MATCHED`. However, an application writer can write their own listener class to monitor any and all statuses, passing their function handle to the `set_listener()` call. This class encapsulates a thread for periodic writing of the sample, and/or event statuses via a waitset.

Class Signature (C'tor)

`def __init__(self, **participant**, **periodic**, **period**, **topic_type_name**, **writer_name**)`

participant - handle to DDS participant, as created from create_participant_from_config() call from the application (see example application)

periodic - Bool, indicates if the writer is periodic

period - float s.ms - period in seconds.milliseconds

topic_type_name - topic type name

writer_name - writer name (c++11/python - used to lookup writer, c++ non-XML Application Creation, a defined const string printed out as the writer is created.

c++ also takes a **QoS profile name** as defined in the XML file, and the **DDSPublisher *** as created int the application from a participant-> create_publisher()call.

Member functions:

`run(self)` - Overloaded function used by the treading library member function when a thread - user calls. foo_topic.start() to start the writer statuses monitoring and/or periodic writing

`write(self)`- Default writes the sample. Can be overloaded by the inheriting topic class to customize the sample or handling of the sample write.

`writer()` - @property - used to get the DDS writer handle to set properties in the DDS writer class (e.g., get the writer instance handle to block self written topics for a participant)

`join(self)` - thread join by the application to shut down threads upon exiting

Defined, but not currently used: **`handler()-`** specific topic context writing, **`enable()`** and **`disable()`** to turn on and off the periodic writing.

`Class Reader(threading.Thread):`

Purpose:

The reader base class is used to read samples and monitor reader statuses. Currently the only statuses monitored are the `SUBSCRIPTION_MATCHED`.
To be useful the user MUST override the handler() in the inheriting topic class to receive the data sample in topic / application context.

This class encapsulates a thread to run the reader waitset.

Class Signature (C'tor)

`def __init__(self, `**`participant, topic_type_name, reader_name`**`)`

`participant` `- handle to DDS participant, as created from create_participant_from_config() call from the application (see example application)`

`topic_type_name` `- topic type name`

`reader_name` `- reader name (c++11/python - used to lookup reader, c++ non-XML Application Creation, a defined const string printed out as the reader is created.`

`c++ also takes a `**`QoS profile name`**` as defined in the XML file, and the `**`DDSSubscriber *`**` as created int the application from a participant-> create_subscriber()call.`

Member functions:

`run(self)` - Overloaded function used by the treading library member function when a thread - user calls. foo_topic.start() to start the reader (reading data and monitoring statuses).

`handler(self, data)` - SHOULD BE OVERRIDDEN by the inheriting topic class. Default, just prints the read sample with the message to override. This is the topic reader specific read handler for a data sample that is taken.

`join(self)` - thread join by the application to shut down threads upon exiting

TODO - allow passing in the statuses monitoring mask, and add all possible statuses handling with overridable member functions by the inheriting topic classes.

`Class DefaultWriterListener(dds.DynamicData.NoOpDataWriterListener):`

Purpose:

Provides a default listener to print writer subscribers as they come and go. As noted above, the application writer can create their own WriterListener class and pass that in to the topic instantiation at the application level.

Example:

Replace:

```
controller_rrm_w.writer.set_listener(ddsEntities.DefaultWriterListener
(), dds.StatusMask.ALL)
```

With

```
controller_rrm_w.writer.set_listener(MyWriterListener(),
dds.StatusMask.ALL)
```

Using the ddsEntities Base Classes:

In the Command / Response c++11 and Python directories there are six key files (c++11 and c++ also have associated header files). For compiled types as provided in the c++98 example there are also a set of type spectific files. They include:

1. **ddsEntities.py/cxx** - as described above. For python and c++11, **this is the only required file**, and generally should not be modified. All other files are application specific. For c++98 (compiled vs. dynamic types) the topics_T.cxx file described below, is also required and should not be modified.

2. **Applications.py/cxx** - Optional example file to provide signal.SIGINT ^C shutdown.

3. **Constants.py /CommandResp.hpp** - Data model constants file (best practice is to generate this file from the idl file using rtiCodeGenerator utility.)

4. **topics.py/cxx** - your application specific topics - described in more detail below.

5. **device.py/cxx** - example ‘Device’ application - this files along with the complementary controller.py/cxx provides an example producer/consumer. Here, the device announces itself, the controller register the device, and sending it a command soliciting a response.

6. **controller.py/cxx** - examples ‘Controller’ application

7. **Compiled types ONLY (c++98 example**) - will have the following additional files.

    - Command/Response (your application name) - rtiCodeGen files from your data model IDL.

        - CommandResp.h/.cxx
        - CommandRespPlugin.h and .cxx
        - CommandRespSupport.h and .cxx

- **topics_T.h** - This file is a provided required file and generally should not be modified.

  The topic_T base classes take care of the type specific topic creation and tracking required by compiled types. It also handles the qos_profile and content filter installation required of programatic entity creation (vs. XML Application Creation used in the c++11 and python wrappers where qos and CFTs are handled in the XML).

  With compiled types, all of the application specific topics (in topics.cxx) will be type specific, and cannot directly inherit from the type agnostic ddsEntity base classes (In python/c++11 the dynamic typed topics are type agnostic and inherit directly from the ddsEntitie base classes). For compiled types, the application specific topics classes will first inherit from the topic_T base classes providing the templatized topic type information. The topic_T classes then inherit from the ddsEntity base classes.

  While requiring an additional layer of inheritance and templatized Type information, the steps to create application specific topics is very similar to dynamic types and is mostly transparent to the user. This document will give explicit guidance and examples of how to create compiled type topics.

Creating your own application specific topics for applications using Dynamic Data and XML Application Creation (topics.py/cxx file)

The information in this section, while specific to the python/c++11 wrappers (i.e. Dynamic data and XML Application Creation), also generally applies to the c++98 (Compiled types and programatic DDS entity instantiation). An additional section to explicitly explain the c++98 wrappers will also be provided.

All topics in this example, used by both the producer and consumer applications (in this example, "controller" and "device") are placed in a single topics.py/cxx file. Nothing in the ddsEntities wrappers requires this organization (i.e., you could create any named files and separate out producer and application topics so long as they correctly inherit from the ddsEntities reader and writer classes).

Also note that Quality of Service (QoS) using XML Application Creation is specified in the system XML file and not a part of these wrappers.

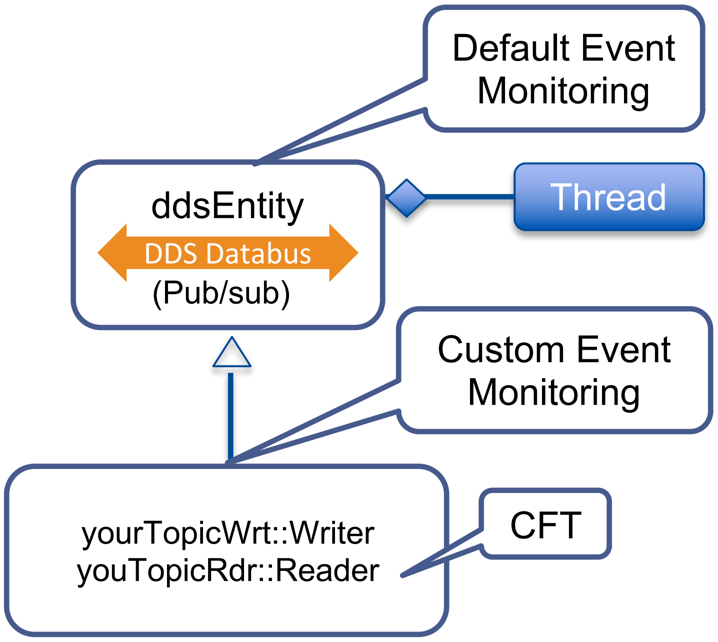

Figure 2, Inheritance and composition of Application Topics (w/XML Dynamic Data types) to ddsEntity classes

Writer Topic Definition

This example shows a non-periodic topic definition. By Defining `periodic = True` and providing a desired period a periodic topic may optionally be created (you must also start the thread via `foo_wtr.start()` in your application (see section, Putting All Together, an Example Application, below)

```
class FooTopicWtr(ddsEntities.Writer):
    def __init__(self, participant, periodic=False, period=0.0):
        ddsEntities.Writer.__init__(self, participant, periodic,
                                    period,
                                    constants.FOO_TOPIC_TYPE_NAME,
                                    constants.FOO_TOPIC_WRITER)
```

You may optionally overload the write() function to customize the writer. The default write() function simply sends the writer property _sample. Here is an example of an overloaded write() where the topic modifies the (application specific data model) topic 'state' field)

```
   def write(self, state):
        print("Writing DeviceState Sample")
        self._sample["state"] = state
        self._writer.write(self._sample)
```

Initialize static writer sample data

The Writer Constructor is also a good place to initialize any data in sample fields that will not change - such as deviceId. Here is an example, where we have an application object that has a function (`setDevIdInSample()`) to correctly load the topic writer _sample property (_sample is a ddsEntities property)

```
        self._app_state_obj.setDevIdInSample(self._sample, "deviceId")
```

The writer topic definition may also include any other application specific member functions and properties needed.

Writer Topic Declaration

In the application a writer topic would then be declared as follows[2]:

```
  foo_writer = topics.FooTopicWriter(participant)
```

To apply a status monitor:

Threaded waitset: `foo_writer.start()`

Or,

[2] With XML Application creation, Entities already exist after the `create_participant_from_config()` call. Here, the writer is looked up via a `find_by_name()` API

Callback listener: (note the user can define their own non-default listener and pass the reference to the set_listener call.

```
foo_writer.writer.set_listener(ddsEntities.DefaultWriterListener(),
dds.StatusMask.ALL)
```

Reader Topic Definition

The topic reader class needs to provide an overload to the `handler()` member function (the default `handler()` in ddsEntities simply prints the sample, but has no capability to 'understand' it.) By overloading the `handler()` in the topic specific class, the sample is passed to a `handler()` that understands the topic and sample context and can perform meaningful action. Of course this class may optionally contain application specific functions and might contain a Content Filter Topic (CFT) member function and capability have readers ignore writer instances of topics both the consumer and producer might write and read). The CFT and ignore writer instances are described below.

```
class FooTopicRdr(ddsEntities.Reader):
    def __init__(self, participant):
        ddsEntities.Reader.__init__(self, participant,
                                    constants.FOO_TOPIC_TYPE_NAME,
                                    constants.FOO_TOPIC_READER)


    def handler(self, data):
        print("Foo Topic Reader Handler Executing")
        print(data)
        # handle the sample in the context of the FooTopic
```

Example CFT

There are a number of uses for Content Filters placed on an existing Topic (so called Content Filter Topic or CFT) . CFTs and their use are really application specific, and not directly supported in the ddsEntities base classes. Because CFTs are type specific, they are supported in the c++ compiled types example below. For completeness, we'll also discuss them for the c++11 and python Dynamic Type example here.

In the Command Response example, the Content Filter Topic (CFT) is placed on Device reader topics to so that the Device application need not concern itself with commands and responses directed from the Controller to other devices (determined by the deviceID). In an actual device, the deviceID would be read from non-volatile RAM or FLASH memory device. In this example, we configure a content filter expression with the last four digits of a 32 byte deviceID to 0000 in the XML Application Creation file, and modify the specific topic CFT with the devices ID in the constructor. This represents a real world use case. This is done as follows.

```
        # Install the content filter for the devices Id, so we only get
        # Command Requests to this device
             cft_topic = dds.DynamicData.ContentFilteredTopic.find ( \
                  self._participant, constants.FOO_CMD_REQ_TOPIC_CFT)
```

```
            cft_topic.filter_parameters = [str(device_id[28]), \
            str(device_id[29]),str(device_id[30]), str(device_id[31])]
```

Reader Topic Declaration[3] with Example ignore writer instance

Often, an application my both publish and subscribe to the same topic, but only want to receive the topic from other devices or applications. Using a CFT to avoid receiving the self sourced samples may not be enough (for example, say for a command response topic that is both written and received by the same application, a command sent FROM a device my have the source deviceId in it, allowing a command response to be sent back to the device using the deviceID. A command sent TO the device, may also require a command response from the device set with the deviceID. In this case we want to only received the command response sent TO the device but ignore the command response sent from the Device. DDS allows this by setting the instance handle in the ingore_datawriter property of the participant. This is done as follows in the specific reader topic constructor :

```
      participant.ignore_datawriter(ignore_wtr_instance_hndl)
```

The writer instance handle parameter is obtained from the specific writer, at the application level, as follows (`writer` is a ddsEntities Writer class property, which provides the writer associated with the specific inheriting writer topic)

```
      foo_topic_rdr = topics.FooTopicRdr(participant, app_state_obj, \
                                   foo_topic.writer.instance_handle)
```

Creating your own application specific topics using compiled types and programatic entity creation (the c++98 example)

The information in this section, is specific to compiled data types and programatic entity (publisher, subscriber, writers and readers creations) as exemplified in the c++98 example.

The process is to create application specific topics is generally the same as shown above with the c++11 and python above, with the addition of a template Topics_H.h (so familiarization with the section above is a prerequisite). With programatic entity creation, all of the DDS container Entities, Participant, Subscriber and Publisher must be created in the application.

Compiled type topics, are type specific, requiring the logic to programmatically instantiate and track writers and readers to be templatized to the specific types. The Topics_H.h provided file is used to perform this function, and should not need to be modified. Rather than the Application (user) topics inheriting directly from the ddsEntites base classes, as done with Dynamic data, they will instead inherit from the reader and writer Topics_H.h templatized topics passing in the topicTypeSupport and other template parameters as shown below. The templatized TopicRdr<T> and TopicWtr<T> in turn inherit from the ddsEntities writer and reader base classes as shown in Figure 2 below.

When using Programmatic entity creation (vs. XML Application Creation), while the QoS profiles are defined in the XML file, they are assigned at writer and reader entity creation and passed to

[3] With XML Application creation, Entities already exist after the `create_participant_from_config()` call. Here, the reader is looked up via a `find_by_name()` API

the TopicT base classes. As such, QoS profile assignment is supported in the example c++ wrappers.

Note: c++11 and Python natively support templatized Types in the reader and writer APIs. This should significantly reduce complexity in creating compiled data types and allow the <T-Type> to be directly incorporated into the ddsEntities; removing the need for a topics_H.hpp type conversion file as shown in the c++98 example here.

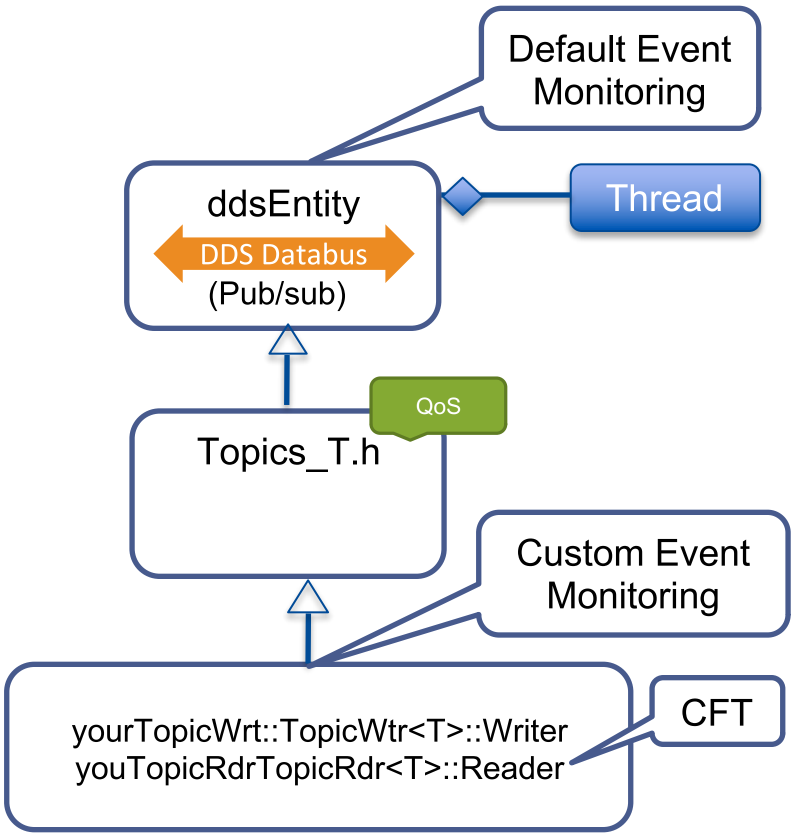

Figure 3, Inheritance and composition of Application Topics (w/XML Compiled Data types) to ddsEntity classes

Writer Topic Definition

This example shows a non-periodic topic definition. By Defining `periodic = True` and providing a desired period a periodic topic may optionally be created (you must also start the thread via `foo_wtr.runThread()` in your application (see section, Putting All Together, an Example Application, below). Like the c++11/python example above, static sample data should be set in the c'tor, with any non-static data handled by overloading the `write()` member function.

Writer Template TopicWtr class in Topics_T.h:

```
template<class T_Topic, class T_TopicSupport, class T_Writer>
Class TopicWtr : public Writer { .. // Writer is a ddsEntity base
class
```

Your topic defined in Topics.h (with different ways to write default and changing sample data):

```
class FooTopicWtr : public TopicWtr<MODULE::FooTopic,
MODULE::FooTopicTypeSupport, MODULE::FooTopicDataWriter> {
    public:
        FooTopicWtr(const DDSDomainParticipant * participant,
                    const DDSPublisher * publisher,
                    const bool periodic=false,
                    const int period = 4 ) :
            TopicWtr(
                participant,
                publisher,
                periodic,
                period,
                MODULE::FOO_QOS_LIBRARY,
                MODULE::FOO_TOPIC_QOS_PROFILE,
                MODULE::TOPIC_FOO_TOPIC,
                MODULE::FOO_TOPIC_WRITER
                )
            // initialize static sample fields and topic vars in C'tor
            {
              this->getTopicSample()->myStaticTopicField=bar;
            };

        // Default writer of sample
        // No override of write() - Provided in Topics_T.h

        // write() overload , writer modifies the sample prior
         // to writing non-static field(s)
        void write(void)
        {
            # modify the sample
            this->topicSample->myNonStaticField1 = Global_foo_bar;
            this->topicSample->myNonStaticField2 = \
                                 this->member_func_get_foo_bar();
            this->write(*this->topicSample, \
            DDS_HANDLE_NIL);
        }

        // write foo_bar specific function
        void FooTopicWtr::writeFoobarData(const FooBar_DataType\
                                           foo_bar_data)
        {
            // modify the sample
            this->topicSample->bar_data_member = foo_bar_data;
            this->write();
        }
```

```
  * MODULE is the name space the user has defined.
```

Parameters TopicWtr (Topics_T.h):

Template parameters:

- `T_Topic` - `Topic` Class from codegen defined in .h file
- `T_TopicSupport` - `TopicTypeSupport` forward from codegen forward referenced in .h file (`TTypeSupport`)
- `T_Writer` - `TopicDataWriter` forward from codegen forward referenced in .h file (`TDataWriter`)

Constructor for TopicWtr.

- `participant` is your application participant handle returned from `create_participant` call in your application
- `publisher` is your application publisher handle returned from `create_publisher` call in your application
- `Periodic - bool` - True to run a periodic writer
- `Period - int` seconds
- `MODULE::FOO_QOS_LIBRARY` - QOS Library name in XML file
- `MODULE::FOO_TOPIC_QOS_PROFILE` - QOS profile name in XML file
- `MODULE::TOPIC_FOO_TOPIC` – string Topic Name (string you may define, used in printout only)
- `MODULE::FOO_TOPIC_WRITER` – string Topic Writer name (string you may define, used in printout only)

Writer Topic Declaration

In the application a writer topic would then be declared as follows:

```
  foo_writer = topics.FooTopicWriter(participant, publisher)
```

`participant` and `publisher` are created in the application.

To apply a status monitor:

Threaded waitset: `foo_writer.runThread()`

Listener:

The default listener is defined in the ddsEntities.h file. It generally just prints out on the console that an event triggered.

```
// Create a listener if we'd rather use vs. event waitset thread.
// Use a Default listener we created, but you can create your own
// listener(s) (and as many as you need if topic specific)
    DefaultDataWriterListener * listener = \
                                new DefaultDataWriterListener();
    topic_writer.getMyDataWriter()->set_listener(listener);
```

Reader Topic Definition

The topic reader class needs to provide an overload to the `handler()` member function (the default `handler()` in ddsEntities simply prints the sample, but has no capability to 'understand' it.) By overloading the `handler()` in the topic specific class, the sample is passed to a `handler()` that understands the topic and sample context and can perform meaningful action. Of course this class may optionally contain application specific functions and might contain a Content Filter Topic (CFT) member function and capability have readers ignore writer instances of topics both the consumer and producer might write and read). The CFT and ignore writer instances are described below.

Reader Template TopicRdr class in Topics_T.h:

```
template<class T_Topic, class T_TopicSupport, class T_Reader, class
T_TopicDataSeq>
Class TopicRdr : public Reader { .. // Reader is a ddsEntity base
class
```

Your topic defined in Topics.h (with different ways to write default and changing sample data):

```
class FooTopicRdr : public TopicRdr<MODULE::FooTopic,
MODULE::FooTopicTypeSupport, MODULE::FooTopicDataReader,
MODULE::FooTopicSeq> {
    public:
        FooTopicRdr(const DDSDomainParticipant * participant,
                    const DDSSubscriber * subscriber,
                    const Cft filter):
            TopicRdr(
                participant,
                publisher,
                filter,
                MODULE::FOO_QOS_LIBRARY,
                MODULE::FOO_TOPIC_QOS_PROFILE,
                MODULE::TOPIC_FOO_TOPIC,
                MODULE::FOO_TOPIC_READER
                )
            // initialize topic variables in C'tor
            {
              this->variable=bar;
            };
```

Override Topic reader `handler` to receive each data valid sample in topic context:

```
            void FooTopicRdr::handler(const MODULE::FooTopic * data) {
                 // your topic reader handler here
            }
```

Parameters TopicRdr (Topics_T.h):

Template parameters:
- `T_Topic` - `Topic` Class from codegen defined in .h file
- `T_TopicSupport` - `TopicTypeSupport` forward from codegen forward referenced in .h file (`TTypeSupport`)
- `T_Reader` - `TopicDataReader` forward from codegen forward referenced in .h file (`TDataReader`)
- `T_TopicDataSeq` - `TopicDataSeq` forward from codegen forward referenced in .h file (`TDataSeq`)

Constructor for TopicRdr.
- `participant` is your application participant handle returned from `create_participant` call in your application
- `subscriber` is your application subscriber handle returned from `create_subscriber` call in your application
- `filter - bool` - True to run a periodic writer
- `MODULE::FOO_QOS_LIBRARY` - QOS Library name in XML file
- `MODULE::FOO_TOPIC_QOS_PROFILE` - QOS profile name in XML file
- `MODULE::TOPIC_FOO_TOPIC` - string Topic Name (string you may define, used in printout only)
- `MODULE::FOO_TOPIC_READER` - string Topic Reader name (string you may define, used in printout only)

Reader Topic Declaration

In the application a writer topic would then be declared as follows:

```
foo_reader = topics.FooTopicWriter(participant, \
                                   subscriber, \
                                   topic_cft);
```

`participant` and `subscriber` are created in the application.

For `topic_cft` (see Example CFT section below)

To start the reader thread:

```
foo_reader.runThread()
```

Readers only support `waitsets()` since they require a thread anyway.

Example CFT

There are a number of uses for Content Filters placed on an existing Topic (so called Content Filter Topic or CFT) . In the Command Response example, the Content Filter Topic (CFT) is placed on Device reader topics to so that the Device application need not concern itself with commands and responses directed from the Controller to other devices (determined by the deviceID). While CFTs and their use are really application specific, for compiled types, a general purpose Cft class is provided by the topics_T.h file. This class accepts a param_list and filter expression and will install a CFT.

```
class Cft {
    public:
        Cft(const char ** param_list=NULL, \
            const char * filter_expression_p=NULL)
```

NOTE: All readers must install a CFT. If your application reader does not need one, you can create a NULL CFT (which is automatically disabled) as follows:

```
  // Reader API take a filter, but controller does not need one
    Cft foo_topic_cft;      // create a disabled cft
```

At the application level, the following will show how to define and install an enabled CFT . Below is an example of installing a device ID filter (note it is user application topic specific, so your implementation may be slightly different)

```
   // Device filters ConfigureDeviceRequests to it's deviceID
  std::string s1 = \
       std::to_string(device_state_writer.getTopicSample()\
                    ->myDeviceId.resourceId);
  std::string s2 = \
       std::to_string(device_state_writer.getTopicSample()\
                    ->myDeviceId.id);
  const char *param_list[] = { s1.c_str(), s2.c_str(), NULL };

  Cft cdr_cft(param_list, "targetDeviceId.resourceId = %0, \
               targetDeviceId.id=%1" );
```

Example ignore writer instance (this is still python specific)

Often, an application my both publish and subscribe to the same topic, but only want to receive the topic from other devices or applications. Using a CFT to avoid receiving the self sourced samples may not be enough (for example, say for a command response topic that is both written and received by the same application, a command sent FROM a device my have the source deviceId in it, allowing a command response to be sent back to the device using the deviceID. A command sent TO the device, may also require a command response from the device set with the deviceID. In this case we want to only received the command response sent TO the device but ignore the command response sent from the Device. DDS allows this by setting the instance handle in the ingore_datawriter property of the participant. This is done as follows in the specific reader topic constructor :

```
      participant.ignore_datawriter(ignore_wtr_instance_hndl)
```

The writer instance handle parameter is obtained from the specific writer, at the application level, as follows (`writer` is a ddsEntities Writer class property, which provides the writer associated with the specific inheriting writer topic)

```
foo_topic_rdr = topics.FooTopicRdr(participant, app_state_obj, \
                                   foo_topic.writer.instance_handle)
```