UMAA SDK: Libraries User Guide

Contents

Purpose

The UMAA SDK Libraries are static C++11 libraries for sending and receiving UMAA messages via RTI Connext® DDS. Each library hosts one of six core functions of the UMAA standard. These libraries can be found in the package's `/lib` directory, and their headers in the `/include` directory.

Library	UMAA Core Function
libwgtddseo.a	Engineering Operations (EO)
libwgtddsmm.a	Mission Management (MM)
libwgtddsmo.a	Maneuver Operations (MO)
libwgtddssa.a	Situational Awareness (SA)
libwgtddssem.a	Sensor and Effector Management (SEM)
libwgtddsso.a	Support Operations (SO)

The purpose of this guide is to serve an explanation of how these libraries are used.

Introduction

When linking the UMAA SDK, be sure to link both the relevant UMAA IDL libraries (e.g., `libumaacommon.a`, `libumaaeo.a`, `libumaamm.a`, etc.) as well as the corresponding WGT DDS libraries (e.g., `libwgtddseo.a`, `libwgtddsmm.a`, etc.).

DDS Managers

Within each WGT DDS library is a suite of DDS "managers" - classes that are responsible for each of the UMAA core function's namespaces and their messages within, both sending and receiving. The managers are named similarly to the namespace of messages

they manage. For example,

Engineering Operations (EO) Namespace	DDS Manager (libwgtddseo.a)
UMAA::EO::AnchorControl	WGTDDS::EO::AnchorControlDdsMgr
UMAA::EO::AnchorSpecs	WGTDDS::EO::AnchorSpecsDdsMgr
UMAA::EO::AnchorStatus	WGTDDS::EO::AnchorStatusDdsMgr
UMAA::EO::BallastTank	WGTDDS::EO::BallastTankDdsMgr
…	…

For convenience, each of the UMAA core function directories within `/include` (e.g., `/include/EO`, `/include/MM`, etc.) contain three supplementary files for users who wish to utilize all of the DDS managers within (where <CF> can be EO, MM, MO, SA, SEM, or SO):

- `<CF>DdsMgrCommon.h` - A header to include all DDS managers of UMAA core function <CF>
- `<CF>DdsMgrDef.hx` - Source code to be included in a user header file to declare an instance of every DDS manager of UMAA core function <CF>
- `<CF>DdsMgrInit.cpx` - Source code to be included in a user source file to instantiate the instance of every DDS manager of UMAA core function <CF> (as declared in `<CF>DdsMgrDef.hx`), with the assumption that a *dds::domain::DomainParticipant* object named `domainParticipant` has been declared and initialized prior

DDS Message Observers

Also within each WGT DDS library is a suite of DDS "message observers" - base classes that are responsible for what to do with received messages. The message observers are named similarly to the namespace of messages they observe. For example,

Engineering Operations (EO) Namespace	DDS Message Observer (libwgtddseo.a)
UMAA::EO::AnchorControl	WGTDDS::EO::AnchorControlMsgObserver
UMAA::EO::AnchorSpecs	WGTDDS::EO::AnchorSpecsMsgObserver
UMAA::EO::AnchorStatus	WGTDDS::EO::AnchorStatusMsgObserver
UMAA::EO::BallastTank	WGTDDS::EO::BallastTankMsgObserver
…	…

These message observers are to be implemented by users (as needed) and registered with the managers to define what should be done with received messages.

Sending a Message

To send a message, the user must do three things:

1. Construct the message's respective DDS manager. For example, to send a *UMAA::EO::AnchorControl::AnchorCommandType* message,

```
int myDdsDomainId = 40;
WGTDDS::EO::AnchorControlDdsMgr* myDdsMgr = new WGTDDS::EO::AnchorControlDdsMgr(myDdsDomainId);
```

2. Construct the message. The contents of a message are defined by their respective IDL file (and furthermore, by the IDL's *rtiddsgen* files). Continuing the example,

```
UMAA::EO::AnchorControl::AnchorCommandType myMessage;

myMessage.action() = UMAA::Common::MaritimeEnumeration::AnchorActionEnumType::AnchorActionEnumType::STOP;

std::string mySourceString = "85FB969910E144EC";
for (int i = 0; i < 16; i++) myMessage.source()[i] = mySourceString.at(i);

std::string myDestiationString = "7DCC528471604BC2";
for (int i = 0; i < 16; i++) myMessage.destination()[i] = myDestiationString.at(i);

std::string mySessionIDString = "C4AE762A181F4AF8";
for (int i = 0; i < 16; i++) myMessage.sessionID()[i] = mySessionIDString.at(i);

myMessage.timeStamp().seconds() = 1663602912;
myMessage.timeStamp().nanoseconds() = 123456;
```

3. Use the `SendMessage` function of the message's respective DDS manager (from step 1). Continuing the example,

```
myDdsMgr->SendMessage(myMessage);
```

And that's it!

Receiving a Message

Receiving a message is a bit more in-depth than sending. The user must do four things:

1. Implement a DDS message observer class for the message to receive. This class must be derived from the message's respective observer (see DDS Message Observers). In the user's implementation, the `ReceiveMessage` function for that message's type should be overriden. For example, to implement an observer for *UMAA::EO::AnchorControl::AnchorCommandType* messages,

```
class MyDdsMsgObserver : public WGTDDS::EO::AnchorControl::AnchorControlMsgObserver
{
public:
    MyDdsMsgObserver() {}
    ~MyDdsMsgObserver() {}

    // Here is where I decide what to do when I receive the message.
    // I want to print the "timeStamp" field.
    void ReceiveMessage(const UMAA::EO::AnchorControl::AnchorCommandType& msg) override
    {
        std::cout << "I got an AnchorCommandType message! ";
        std::cout << "It has the timeStamp: " << msg.timeStamp().seconds() << "." << msg.timeStamp().nanoseconds() << "\n";
    }
};
```

Note, the base class *WGTDDS::EO::AnchorControl::AnchorControlMsgObserver* defines multiple `ReceiveMessage` functions - one for each of the messages within the UMAA::EO::AnchorControl namespace. The user need only override the function(s) relevant to the message(s) they want to receive.

2. Construct the message's respective DDS manager. Continuing the example,

```
int myDdsDomainId = 40;
WGTDDS::EO::AnchorControlDdsMgr* myDdsMgr = new WGTDDS::EO::AnchorControlDdsMgr(myDdsDomainId);
```

3. Construct the message's observer implementation (from step 1) and register it with the message's respective manager (from step 2) using the manager's `RegisterDdsMsgsObserver` function. Continuing the example,

```
MyDdsMsgObserver* myObserver = new MyDdsMsgObserver();
myDdsMgr->RegisterDdsMsgsObserver((WGTDDS::EO::AnchorControl::AnchorControlMsgObserver*)myObserver);
```

4. Use the message's respective manager's `StartListenDdsMsgs` function to begin listening for messages. Continuing the example,

```
myDdsMgr->StartListenDdsMsgs();
```

And it's all set up! Whenever the message is received, the user's `ReceiveMessage` function from step 1 will be called.

To stop receiving a message, use the message's respective manager's `StopListenDdsMsgs` function.

Specifying DDS QoS

To specify a DDS Quality of Service (QoS) for applications that use the UMAA SDK libraries, the user has two options:

1. Place the QoS in the directory the application will be ran from.

2. Provide the QoS as an object to a DDS domain participant. This is done using the DDS manager constructor that allows the domain participant to be provided by the user. For example,

```
int myDdsDomainId = 40;
dds::core::QosProvider myQosProvider("path/to/my/qos.xml");
dds::domain::DomainParticipant myDomainParticipant(myDdsDomainId, myQosProvider.participant_qos());
WGTDDS::EO::AnchorControlDdsMgr* myDdsMgr = new WGTDDS::EO::AnchorControlDdsMgr(myDomainParticipant);
```

To learn more about what DDS QoS is, visit this site.

Monitoring Messages

To monitor messages, the *rtiddsspy* debugging tool from RTI Connext® provides a look into DDS traffic. When running this tool, ensure its options align with any that may have been provided to the UMAA SDK library (e.g., domain ID, QoS). It is recommended that *rtiddsspy* is ran with the `-printSample` option so that the fields within messages can be seen.

Version 1.0.301a (Sept 19, 2022)

175 Admiral Cochrane Dr., Suite 302, Annapolis, Maryland 21401, United States
Site