# micROS RT 0.20 User's Manual

*micROS RT, an implementation of ROS 1.x on DDS*

*Please contact us through bding@msn.com. Any feedback would be greatly appreciated!*



## Part I Introduction

### 1.1 What is micROS RT?

MicROS RT (micROS Real-Time) is a modified ROS C++ kernel which adopts OMG's DDS (Data Distribution Systems for Realtime Systems) as its underlying message transfer protocol. DDS is an Object Management Group's standard for pub/sub middleware (http://portals.omg.org/dds/). It supports high-performance, scalable and QoS-assuring message delivery. It has been applied into many industry-level systems.

In fact, the ROS official has also the plan to integrate ROS 2.0, the next generation of ROS, with DDS (http://design.ros2.org/articles/ros_on_dds.html). The major difference between micROS RT and the future ROS 2.0 includes (according to currently available information of ROS 2.0):

(1) MicROS RT keeps compatibility with existing ROS 1.x package and programming paradigm, because we believe that the protection of existing investments important to the ROS community. Existing programs need no modification and can easily benefit from DDS.

(2) MicROS RT lays strong emphasis on exploiting the QoS-features of the underlying DDS middleware (cf. Section 1.2), which are important to multi-physical-node and multi-robot settings.

### 1.2 Why micROS RT?

By replacing the original ROS message protocols (TCPROS & UDPROS) with DDS, we can support a set of advanced features in ROS message delivery process, such as UDP-based multicast and setting the QoS properties (transport priority, latency budget, etc.) of a specific topic. In concrete, we can achieve the benefits as follows.

**General benefits**
**(1) Enabling multi-cast.** The DDS middleware will automatically select the multi cast protocol when necessary. Therefore, when there are *n* listeners in a topic (*n*>=2), you can achieve significant performance benefit.
**(2) Robustness in unexpected settings.** Since DDS is a mature and industry-level message delivery system, it is behaves better in various complex communications settings. For example, it has better reconnection behavior when dropping out of wireless.

**QoS-related benefits**
**(1) Transport priority and latency budget of a topic**. With micROS RT, you can specify the transport priority and latency budget of a topic. It is useful in many real-time settings, such as multi-robot collaboration with limited wireless network bandwidth.
**(2) Expected message arriving deadline.** With micROS RT, you can set the expected message arriving period as well as the behavior when a message not arriving in this period. It is also useful in

real-time settings.
**(3) Time-based message filter.** You can filter unnecessary messages based on time properties. For example, if you need only update the state of a sensor per minute, you can ignore other redundant messages by simply setting a property while you subscribing the topic.
**(4) Other QoS-related features.** With micROS RT, you can specify other QoS-related properties provided by the underlying DDS middleware in a simplified way when advertising/subscribing a topic, such as reliability of message delivery, data-centric update, message order method, and message valid period.

**Compatibility**
**(1) Existing ROS packages need no modification.** Existing ROS packages can easily benefit from DDS by simply replacing the libroscpp.so in the ROS installation directory. Those packages needs no modification and re-compilation.
**(2) Existing ROS programming paradigm is kept.** Existing ROS programming paradigm is kept, except for several new APIs to set topic QoS are added.
**(3) Interoperable with official ROS kernel.** The micROS RT can smartly choose appropriate protocol when it communicate with a official C++ or Python ROS kernel. The preferred protocol of a topic can also be specified when advertising it.

# Part II Installation Guide

Currently, micROS RT is modified based on ROS Indigo. However, the kernel modified based on ROS Hydro will be provided soon. You can install micROS RT in two different ways: installation from binary library or installation from source code.

## 2.1 Installation from binary library

**Step1: Replacing official ROS C++ Kernel with micROS RT**
(1) Installing ROS Indigo.
(2) Installing OpenSplice DDS 6.4 community version (http://www.prismtech.com/opensplice/opensplice-dds-community).
(3) Please ensure that the DDS environment variables have been set correctly. Usually, you can achieve this goal by simply running "source %DDSInstallationPath%/release.com".
(4) Downloading libroscpp.so from here. Please select the appropriate directory according to you ROS version and OS bits (indigo 64bit or indigo 32bit).
(5) Replacing the libroscpp.so in the ROS library directory (usually /opt/indigo/lib/) with the file you downloaded.

**Step2: Adding micROS RT development suppport (optional)**
(6) If you want to use micROS-RT newly added APIs in you program (cf. Part III of this document), please download the micROS-RT header files from here (by pressing the “raw” button) and extract them into the %ROSInstallationPath%/include/ros directory (usually /opt/indigo/include/ros). You will be prompted that several files are replaced.

## 2.2 Installation from source code

(1) Installing ROS Indigo from source code (http://wiki.ros.org/indigo/Installation/Source)
(2) Installing OpenSplice DDS 6.4 community version (http://www.prismtech.com/opensplice/opensplice-dds-community).
(3) Please ensure that the DDS environment variables have been set correctly. Usually, you can achieve this goal by simply running "source %DDSInstallationPath%/release.com". And since you

will compile ROS cpp core with DDS devlopment support, the DDSInstallationPath should be the HDE directory of OpenSplice DDS.
(4) Downloading the modified roscpp source code from here (by pressing the “raw” button) and extracting it.
(5) Replacing %ROSInstallationPath%/src/ros_comm/roscpp directory with the directory you downloaded.
(6) Recompiling the roscpp package. You can add “--pkg roscpp” to the catkin_make_isolated command to achieve this goal.

If you installed micROS RT successfully, you can see the information “[DDS] DDS Ready!” in the console when launching a program which advertises a topic.

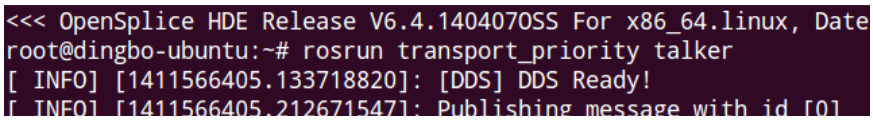

Figure 1 Information that indicates DDS launched

## Part III Developer's Guide

As we have mentioned, existing ROS packages can easily benefit from DDS by simply replacing the libroscpp.so in the ROS installation directory. After replacing it, all ROS messages will be automatically switched to the DDS middleware. However, if you want to set the QoS parameters of a topic, you should modify your program and use the newly added APIs in micROS RT.

Note that the QoS parameters you specified only take effects when you using micROS RT on both publisher and subscriber side. And the QoS parameters are only available to the topic-based pub/sub communication. In other words, they don't affect service invocations in ROS.

### 3.1 Setting QoS parameters

In micROS-RT, the QoS parameters are set on the node-topic level, which means you can specify different message delivery QoS parameters (e.g., transport priority and latency budget) for each topic on each node. Even two nodes both publish messages on the same topic, they can set different transport priority and other QoS parameters. Besides, the publisher and the subscriber should specify different QoS parameters for their actions (Table I).

**Table I QoS Parameters that micROS RT supports**

| **Message publisher QoS Parameters (Using advertiseWithQoS() method)** | |
|---|---|
| Transport priority | Message transport priority |
| Latency budget | Expected latency from publisher to the subscriber |
| Best effort delivery | Using the best effort message delivery protocol or not |
| Message valid period | The valid period (lifespan) of a message |
| Data centric update | Not supported yet |
| **Message subscriber QoS Parameters (Using advertiseWithQoS() method)** | |
| Message order method | Message is ordered by the sending time stamp or the arriving sequence |
| Time filter duration | Performing time-based message filterring |
| Best effort delivery | Using the best effort message delivery protocol or not |

| | |
|---|---|
| data_centric_update | Not supported yet |
| Deadline | Not supported yet |
| Behavior when deadline is not fulfilled | Not supported yet |

In the original ROS programming practice, we use the NodeHandle::advertise() method to create a Publisher which is used to publish on a topic, and the NodeHandle::subscribe() to subscribe the messages on a topic. In micROS-RT, we add two new methods, NodeHandle::advertiseWithQoS() and NodeHandle::subscribeWithQoS(). They can be regarded as the QoS available version of advertise() and subscribe(). If you want to specify the QoS parameters, you should use them instead of the original ones.

3.1.1 advertiseWithQoS()

Basically, advertiseWithQoS() adds some QoS-related parameters to the NodeHandle::advertise() method. Since in the offical ROS kernel, NodeHandle::advertise() has several variations, the advertiseWithQoS() has the following variations as well. The meaning of the parameter *topic*, *queue_size*, *latch* and *ops* are identical to the original advertise() method.

1. *Publisher advertiseWithQoS(const std::string& topic, uint32_t queue_size, TransportPriority priority, bool latch = false) //setting transport priority of messages on this topic directy*
2. *Publisher advertiseWithQoS(const std::string& topic, uint32_t queue_size, Duration latency_budget, bool latch = false) //setting latency budget of messages on this topic directly*
3. *Publisher advertiseWithQoS(const std::string& topic, uint32_t queue_size, AdvertiseQoSOptions& qos_ops, bool latch = false)*
4. *Publisher advertiseWithQoS(AdvertiseOptions& ops, AdvertiseQoSOptions& qos_ops);*

**The first one** just added a *priority* parameter to the original advertise() method. Its type, *TransportPriority*, is an enum type which is defined as follows:

```
enum TransportPriority
  { ExtremelyLow, VeryLow, Low, Normal, High, VeryHigh, ExtremelyHigh };
```

In other words, you can set the transport priority of the messages of this topic by selecting one from those above seven levels. For example, you can set the transport priority of the topic “chatter” to the low level by the following statement.

```
ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000, ros::Low);
```

**The second one** just added a *latency_budget* parameter to the original advertise() method. It can set the delivery latency budget (from the publisher to the listener) of the message on this topic. It is a hint to the underlying DDS middleware, which will automatically adapt its behavior to meet the requirements of the shortest delay if possible. It is of the ros::Duration type.

**The third one and the fourth one** involves a new structure named qos_ops. This structure are of the type of AdvertisQoSOptions, which is defined as follows:

```
struct AdvertiseQoSOptions
{
  TransportPriority transport_priority; //message transport priority of this topic
  Duration latency_budget;  //message latency budget of this topic
  bool using_best_effort_protocol; //using best effort transport protocol or not
  bool data_centric_update; //using data centric update or not
```

```
  Duration msg_valid_period; // message valid period
};
```

This structure controls the QoS beahvior when this node publishes a message on this topic. The meaning of its members are as follows:

- *transport_priority.* We have explained it earlier. The default value of this member is ros::Normal.
- *latency_budget.* We have explained it earlier. The default value of this member is ros::DURATION_MIN, which tells the underlying DDS middleware to deliver the message as soon as possilbe.
- *using_best_effort_protocol.* It tells the underlying DDS middleware to use the best effort transport protocol or the reliable one. When using the unreliable transport protocol, DDS will only attempt to deliver the data, and no arrival-checks are being performed and any lost data is not re-transmitted (non-reliable). When using the reliable transport protocol, extra arrival-checks are performed and data may get re-transmitted in case of lost data. However, it may cause some extra performance cost. The default value of this member is *false*, which means using the reliable protocol.
- *data_centric_update.* Not formally supported in this version.
- *msg_valid_period.* It tells the underlying DDS middleware the valid period (lifespan) of a message. When this time period has expired and the subscriber doesn't receive it yet, the message will be automatically discard. It is useful to reduce network traffic under certain circumstances. Its default value is ros::DURATION_MAX, indicating that the message does not expire.

Note that if you set *using_best_effort_protocol* to true, you have to set the QoS parameter *using_best_effort_protocol* of the subscriber to true as well. Or else you subscriber cannot receives the messages published by this node, because they adopt different protocols.

The following code is an example to use the third variation of advertiseWithQoS(). Note that when declaring a *AdvertiseQoSOptions*, its members will have defualt values and you can only modify the members you want change.

```
ros::AdvertiseQoSOptions adv_qos_ops;
adv_qos_ops.using_best_effort_protocol=true;
ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000, adv_qos_ops);
```

### *3.1.2 subscribeWithQoS()*

The subscribeWithQoS() has the following variations. The meaning of the parameter *topic*, *queue_size*, <callback> and *ops* are identical to the original advertise() method. We only add a new parameter *qos_ops* whose type is *SubscribeQoSOptions*.

1. *Subscriber subscribeWithQoS(const std::string& topic, uint32_t queue_size, SubscribeQoSOptions& qos_ops, <callback> )*
2. *Subscriber subscribeWithQoS(SubscribeOptions& ops, SubscribeQoSOptions& qos_ops);*

SubscribeQoSOptions are defined as follows:

```
struct SubscribeQoSOptions
{
  Duration deadline;
  DeadlineMissedCallback deadline_cb;
```

```
  bool ordered_by_sending_timestamp;
  Duration time_filter_duration;
  bool using_best_effort_protocol;
  bool data_centric_update;
};
```

This structure controls the QoS beahvior realted to the action of this node subscribing this topic. The meaning of its members are as follows:

- *deadline.* It set the period within which this node expects a new message on this topic. Its default value is ros::DURATION_MAX. And it is not formally supported in this version.
- *deadline_cb.* It is a callback function which will be invoked when no new message is arrived before the deadline. It is not formally supported in this version as well.
- *ordered_by_sending_timestamp.* It controls the message is ordered according to the sending time stamp (publisher side) or the arriving time stamp (local side). Its default value is *true*.
- *time_filter_duration*. It specifies the time period to filter the messages. In a time period, only one message will be received and other message will be discarded. For example, if a sensor node publishes its state at a 100hz frequency and another node only want to process the state 1 times per second, you can use this parameter to filter unnecessary messages. It is of ros::Duration typoe and its default value is ros::DURATION_MIN, which means no filter.
- *using_best_effort_protocol.* It tells the underlying DDS middleware to use the best effort transport protocol or the reliable one. When using the unreliable transport protocol, DDS doesn't support message arrival-checks are being performed and any lost data is not re-transmitted (non-reliable). When using the reliable transport protocol, extra arrival-checks are performed and data may get re-transmitted in case of lost data. However, it may cause some extra performance cost. The default value of this member is *false*, which means using the reliable protocol.
- *data_centric_update.* Not formally supported in this version.

Note that the *subscribeWithQoS()* method doesn't have the *transport_hints* parameter in contrast with the original subscribe() method, since when you specify a QoS requirement the kernel will choose DDS protocol automatically.

### 3.2 Selecting the preferred protocol of a topic

At runtime, the micROS RT kernel has the ability to negotiate the message delivery protocol with other nodes. It will choose the DDS protocol in the first place. However, if the remote node is an official ROS kernel withous DDS protocol support, it will smartly switched to ROSTCP or ROSUDP. This ability is realized based an enhancement of the official ROS's protocol negotiate framework.

If you don't want to use DDS on a topic, you can specify the preferred protocol when you subscribe the topic using the *transport_hints* parameter of the subscribe() method. The following statement specifies that the preferred protocol is DDS.

```
ros::Subscriber sub = nh.subscribe("my_topic", 1, callback, ros::TransportHints().dds());
```

And the following statement specifies that the preferred protocol is TCPROS, the tcp protocol built in the ROS official kernel.

```
ros::Subscriber sub = n.subscribe("chatter", 1000, callback, ros::TransportHints().tcp());
```

Please refer to here for more information about the *transport_hints* parameter.

### 3.3 Examples

3.3.1 Writing a Publisher with QoS

The following program create three topics with different transport priority. The only difference between it and a official ROS-based program is that it use *advertiseWithQoS()* instead of *advertise()* to advertise a topic.

```
#include "ros/ros.h"
#include "std_msgs/String.h"

int main(int argc, char **argv)
{
  ros::init(argc, argv, "talker");
  ros::NodeHandle n;
  ros::Publisher high_pub, normal_pub, low_pub;

  //using  advertiseWithQoS() instead of advertise()
  //the last parameter of  advertiseWithQoS() is transport priority. See Section 3.1.1
  high_pub = n.advertiseWithQoS<std_msgs::String>("high", 1000, ros::High);
  normal_pub = n.advertiseWithQoS<std_msgs::String>("normal", 1000, ros::Normal);
  low_pub = n.advertiseWithQoS<std_msgs::String>("low", 1000, ros::Low);

  ros::Rate loop_rate(20);

  while (ros::ok())
  {
    //Publishing messages on each topic here
    …...

    ros::spinOnce();
    loop_rate.sleep();
}

  return 0;
}
```

3.3.1 Writing a Subscriber with QoS

The following program subscribe a topic with a time-based filter. We use *subscribeWithQoS()* instead of *subscribe()* to subscribe a topic and tells the kernel to only allow one message per second.

```
#include "ros/ros.h"
#include "std_msgs/String.h"

void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
  ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "listener");
  ros::NodeHandle n;
```

```
  // using subscribeWithQoS() instead of subscribe()
  // the first parameter of the constructor of ros::Duration is second, and the second on is nanosecond
  ros::Duration filter_duration(1, 0);
  ros::SubscribeQoSOptions qos_ops;
  qos_ops.time_filter_duration=filter_duration;
  ros::Subscriber sub = n.subscribeWithQoS("chatter", 1000, qos_ops, chatterCallback);

  ros::spin();

  return 0;
}
```

More examples can be found here.