

Introduction to ROS: Basics, Motion, and Vision

- 1 In general, we can use **`gdb`** for debugging. Also, memory leaks can be checked with **`valgrind`**. To see the graphical graph representation and performance analysis, **`callgrind`** and **`KCachegrind`**.
- 2 You can not use **`gdb rosrun ros_debugging gdb_hello_world`** for debugging the code. However, there are two ways you can debug your code

to debug with gdb

```
cd devel/lib/ros_debugging  
gdb gdb_hello_world  
> (gdb) r
```

However, this way we can not load parameters and other config files.

- 1 Second option is to define the gdb options under launch-prefix the launch file

to debug with gdb

```
<launch>  
<node pkg="ros_debugging" type="gdb_hello_world" name="gdb_hello_world"  
output="screen" launch-prefix="xterm -e gdb -args"/>  
</launch>
```

Depending on the situation, you can use different keys to control your program execution flow. r for running the program, then bt for backtracing if you got into some problems.

- 1 If there is problems with memory leaks or with performance, valgrind can be used

to debug with valgrind

```
<launch>  
<node pkg="ros_debugging" type="rqt_logging" name="rqt_logging"  
output="screen" launch-prefix="valgrind"/> </launch>
```

ROS Debugging: Callgrind + Kcachegrind



- 1 Callgrind is a profiling tool that records the call history among functions in UNIX process

to debug with valgrind

```
<node pkg="ros_debugging" type="rqt_logging" name="rqt_logging"
  output="screen" launch-prefix="valgrind --tool=callgrind
  --callgrind-out-file='callgrind.example1.%p'"/> </launch>
```

- 2 The profiling log is saved in the `/.ros` directory
- 3 `kcachegrind /.ros/callgrind.example1.xxxx` for visualizing the profile

- 1 Logging usually performance-wise expensive
- 2 However, log4cxx, which is a port of log4j logger library, has a null footprint on performance
- 3 Logging has several LEVELs: DEBUG, INFO, WARN, ERROR, and FATAL
- 4 Different between ERROR and FATAL is that if there is an error, but program can still run, logging should be defined as ERROR otherwise FATAL

to import logging functionality

```
#include <ros/ros.h>  
#include <ros/console.h>
```

1 How can we do logging?

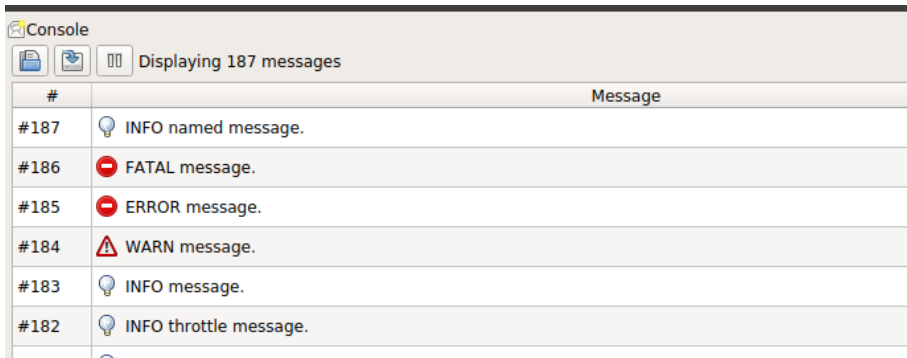
to define logging

```
ROS_<LEVEL>[_<OTHER>], e.g., ROS_INFO("INFO message");  
ROS_<LEVEL>[_STREAM]_NAMED, e.g.,  
ROS_INFO_STREAM_NAMED("class_name", "INFO stream message; val = "  
« val);  
ROS_<LEVEL>[_STREAM]_COND[_NAMED], e.g.,  
ROS_INFO_STREAM_COND(val < 0., "INFO stream message; val (" « val « "  
< 0");  
ROS_INFO_STREAM_ONCE("INFO stream message");
```

ROS Logging

to manipulate with logging messages

```
roslaunch rqt_console rqt_console
```



The screenshot shows the ROS rqt_console interface. At the top, it says "Console" with a folder icon. Below that are icons for saving, refreshing, and a list icon, followed by the text "Displaying 187 messages". The main area is a table with two columns: "#", representing the message index, and "Message", representing the log content. The table lists messages from index 182 to 187. Each message is preceded by a colored icon indicating its severity: a lightbulb for INFO, a red circle with a minus sign for FATAL, a red circle with a minus sign for ERROR, and a red triangle with an exclamation mark for WARN.

#	Message
#187	INFO named message.
#186	FATAL message.
#185	ERROR message.
#184	WARN message.
#183	INFO message.
#182	INFO throttle message.