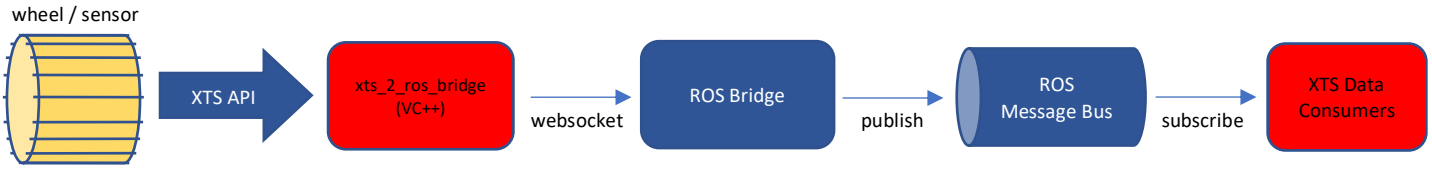
xts_2_ros_bridge : Xiroku Pressure Pad to ROS Bridge Application
Usage & Implementation Notes
jdyoung – 2019-04-18

In order to enable rapid ML prototyping using the Ubuntu-based ROBOT Operating System (ROS : http://ros.org) alongside the Windows-only Xiroku XTS Sensor Pad API, the Visual C++ application *xts_2_ros_bridge* was developed to capture sensor data through that API and issue web-socket calls to the ROS Bridge (http://wiki.ros.org/rosbridge_suite) to publish the data on the ROS message bus. The high-level process flow is depicted here:



Items in blue are third party components. Those in red are internally developed.

The development and testing of the *xts_2_ros_bridge* application were performed on an Intel *Next Unit of Computing* (NUC), which is the primary platform for current efforts involving the Xiroku pressure pad. This document presents development and the runtime in the context of the NUC. The VC++ solution and ROS files (i.e. the message definition and simple publisher and subscriber test scripts) are checked into the barefoot rover github-fn repository as subdirectories *xts_2_ros_bridge* and *barefoot_rover_ros* (respectively) under *src/CROSSBOW*.

Usage

These steps assume that the environment has already been set-up, see "Set-up" below for details. For the sake of this discussion, it is assumed that the catkin workspace is */home/barefoot/workspace/barefoot_rover/ros* (which will be referred to as *$(catkin_home)* for brevity, although this is not an actual environment variable) and the Barefoot Rover package is installed in the *src/barefoot_rover_ros* subdirectory (which will be referred to only as *barefoot_rover_ros*).

1. Start ROS (WSL)
   In this context, *roscore* provides the ROS message/topic bus and is the first component started. Before starting *roscore*, the setup script in *$(catkin_home)/devel* (whether *setup.bash, setup.sh, setup.zsh* depending on the shell being used) must be sourced/executed. This can be added to the shell initialization script (e.g. *~/.bashrc*)

   ```
   source /home/barefoot/workspace/barefoot_rover/ros/devel/setup.bash
   ```

   Then start roscore:

   ```
   roscore &
   ```

2. Start the ROS bridge (WSL)
   Use *roslaunch* to start the ROS bridge:

   ```
   roslaunch rosbridge_server rosbridge_websocket.launch
   ```

   This can also be run as a background process if preferred. The bridge is available once a message similar to the following is displayed:

   ```
   [INFO] [1556208729.987623]: Rosbridge WebSocket server started on port 9090
   ```

The bridge endpoint is *ws://localhost:9090*

3. Start all ROS subscriber applications (WSL)
   As an example, the test subscriber found in *barefoot_rover_ros/scripts* listens for and acknowledges receipt of data messages, and optionally displays the data, albeit w/ a caveat[1].

   ```
   rosrun barefoot_rover_ros xts_subscriber.py
   ```

   The *xts_subscriber* accepts an optional topic name as a command line argument, which is otherwise defaulted to "*XtsBus"* which is the same default used by the *xts_2_ros_bridge* application when sending messages to the ROS bridge.

4. Connect the sensor pad to the host. (FYI, the pad is powered through the USB connection.)

5. Start *xts_2_ros_bridge* application (Windows)
   The application has three primary operational modes:
   a. obtain data from the sensor pad, publish to the ROS topic via the ROS bridge

      ```
      .\xts_2_ros_bridge.exe <deviceId> <bridgeEndPointURI>
      ```

   b. obtain data from the sensor pad, write data to an output file

      ```
      .\xts_2_ros_bridge.exe -o <outputfileName> <deviceId>
      ```

   c. read data from an input file, publish to the ROS topic via the ROS bridge

      ```
      .\xts_2_ros_bridge.exe -f <inputfileName> <bridgeEndPointURI>
      ```

   In cases of (a) and (b), the scan is started immediately, i.e. data is captured from the sensor pads and continues until the application is terminated with a ctrl-C (signal handling is in place to assure a graceful shutdown). In the case of (c), the application terminates once the end of file is reached or is explicitly terminated with a ctrl-C.

   In cases of (b) and (c), the format of the file is identical to that produced by the Xiroku *LLtest* application. Hence files produced by *LLtest* can also be used as input for (c).

   The <*deviceId*> is typically 'VD000000000001' – see 'Virtual Devices and the XtsApi.ini file' below. As noted above, the <*bridgeEndPointURI*> is typically 'ws://localhost:9090'.

Virtual devices and the XtsApi.ini file

When multiple sensor pads are chained together, they collectively constitute what Xiroku terms a *virtual device*. In all cases, the ID of a virtual device can be used interchangeably with the ID of a single pad. However, the virtual device must be specified in the *XtsApi.ini* file which must be located in the same directory as the loaded *XtsApi.dll*. Virtual devices and

[1] As noted in the usage output of this script, displaying all received data can cause messages to be lost since the ROS message bus has a maximum capacity after which messages are dropped. Refer to the ROS topic documentation for more details (http://wiki.ros.org/Topics).

the XtsApi.ini file are described in *XtsApi.docx* file found in the Barefoot_Rover repository in the *tools/Xiroku/XtsApi-2.13* directory. This document is a Google Japanese to English translation of the Xiroku user guide and the virtual device guide. The checked in version of the XtsApi.ini file contain the virtual device specification of the current (initial) wheel and pressure pad setup. Since individual sensor pads have internally coded device ID's, the contents of the *XtsApi.ini* file will need to be updated once additional rigs become available.

Options
There are a couple of items that may be useful in other development, debugging and testing efforts.

- There is a ROS script *xts_test_publisher* which accepts an input file (having the format as produced by *xts_2_ros_bridge* and/or Xiroku's *LLtest*) and publishes it to the ROS topic. This is provided for testing of topic subscribers in a manner isolated to ROS (i.e. w/o using the ROS bridge). Once the subscriber is running (including *roscore*), the publisher can be started:

  ```
  rosrun barefoot_rover_ros xts_test_publisher <filename> {<ros_topic>}
  ```

  The topic is defaulted to *'XtsBus'* and publishing of data begins upon script invocation.

- There is a fourth operational mode of *xts_2_ros_bridge* in which data is obtained from an input file and written to an output file. This very limited functionality is for debugging and testing the output formatting and does not appear in the usage output from the application.

  ```
  .\xts_2_ros_bridge -of <outputfilename> <inputfilename>
  ```

Known Issue
When xts_2_ros_bridge disconnects from the ROS bridge, the following message can be observed:

```
[WARN] [1556230474.034352]: Could not process inbound connection: [/rosbridge_websocket] is not a publisher of
[/XtsBus]. Topics are [['/client_count', 'std_msgs/Int32'], ['/rosout',
'rosgraph_msgs/Log']]{'message_definition': 'uint64 frameNumber\nuint32 xCoils\nuint32 yCoils\n\nXtsXCoil[]
frameData\n\n================================================================================\nMSG:
barefoot_rover_ros/XtsXCoil\nfloat32[] xCoil\n', 'callerid': '/listener_235_1556230338222', 'tcp_nodelay': '0',
'md5sum': '48f19b737e2d55d508bf701f69886089', 'topic': '/XtsBus', 'type': 'barefoot_rover_ros/XtsScan'}
```

While shown as a warning, this can prevent subsequent ROS bridge clients from publishing to the same topic. This has been reported as an issue with rospy's management of topic subcribers but can be worked around by commenting out the following lines (at or around line 321):

```
def _unregister_impl(self, topic):
      # Work-around for annoying WARNING message
      # see https://github.com/RobotWebTools/rosbridge_suite/issues/138
      # if not self._publishers[topic].has_clients():
            # self._publishers[topic].unregister()
            # del self._publishers[topic]
      del self.unregister_timers[topic]
```

from *publishers.py* in the rosbridge_library package (the file is typically found in */opt/ros/kinetic/lib/python2.7/dist-packages/rosbridge_library/internal*).

Setup
The following steps are necessary when setting up a new system to run the application and ROS.

Disabling IPv6

To eliminate some connectivity and latency headaches (particularly w/ Remote Desktop Connections and acquiring software through *apt* and *git*), it is advisable to disable IPv6:

1. Open the Windows Control Panel
2. Select Network & Internet
3. Select Wi-Fi on the left bar
4. Click on "Change Adapter Options", a new window containing available network connections will pop up
5. Right click Wi-Fi icon and select "Properties"
6. In the Networking tab of Wi-Fi Properties window, uncheck 'Internet Protocol Version 6 (TCP/IPv6)'

Setup steps

- Install the Windows git client: https://gitforwindows.org
- Install the *vcpkg* library manager for Windows (need to install cpprestsdk): http://gitub.com/Microsoft/vcpkg
- Install WSL : open the Microsoft Store, search for "Ubuntu 16.04", install and set-up.
- In a WSL window
    - Install git :
      ```
      sudo apt-get install git
      ```
    - Install ROS Kinetic : see http://wiki.ros.org/kinetic/Installation/Ubuntu
    - Install ROS-bridge :
      ```
      sudo apt-get install ros-kinetic-rosbridge-server
      ```
    - Create the catkin workspace :
      ```
      mkdir -p ~/workspace/barefoot_rover/ros/src
      ```
    - Checkout the barefoot rover repo into a separate directory and copy the ROS package *barefoot_rover_ros* found under *src/CROSSBOW* into the catkin workspace
    - Add the setup scripts to *.bashrc*
      ```
      echo -e "source /opt/ros/kinetic/setup.bash\n" \
         "source /home/barefoot/workspace/barefoot_rover/ros/devel/setup.bash" \
         >> ~/.bashrc
      ```

VC++ Development

[WIP]

Visual Studio 2017 was used for development.

The following are the external dependencies of *xts_2_ros_bridge*:

- Microsoft cpprest (https://microsoft.github.io/cpprestsdk/index.html): The web::websockets::client class is used to communicate with the ROS Bridge.
- NLohmann JSON library (https://github.com/nlohmann/json) : Used to compose the JSON messages sent to the ROS Bridge.
- Hyperrealm libconfig (https://github.com/hyperrealm/libconfig) : Configuration file support.
- XtsAPI (checked into the *tools/Xiroku* directory of the Barefoot Rover repo) : The Xiroku pressure sensor API. The distributed documentation is in Japanese but translations of the key documents (and code comments) have been checked in. Note that the *matrix.h* file which defines the data structure (*CMatrix* template) for the pressure pad data has been copied from the Xiroku sample application.

Note that there is an issue w/ packages that import Boost (i.e. cpprest) which cause Windows.h functions to become undefined (see https://stackoverflow.com/questions/38201102/including-boost-network-libraries-makes-windows-h-

functions-undefined), which is remedied by some explicit *#include* directives made prior to the offending packages. This is noted in the code.

Solution Directory/File Organization

The root directory of the solution is (not at all coincidentally) *xts_2_ros_bridge* and is organized as follows:

- `xts_2_ros_bridge`
  - `xts_2_ros_bridge.sln` – The VC++ solution file
  - `xts_2_ros_bridge.docx` – This document in Word format.
  - `xts_2_ros_bridge.pdf` – This document in pdf format.
  - `Release` – The directory containing the release build of the solution.
  - `Debug` – The directory containing the debug build of the solution.
  - `xts_2_ros_brige` – the source sub-directory, containing:
    - `dependencies` – third party dependencies not installed elsewhere in the repo
      - `json` – Nlohmann's JSON library
      - `libconfig` – hyperrealms libconfig
    - `matrix.h` – the data structure for scan data, borrowed from Xiroku's sample app.
    - `xts_2_ros_bridge.cfg` – the configuration file for the application
    - `xts_2_ros_bridge.cpp` – application source code
    - `xts_2_ros_bridge.vcxproj` – the application VC++ project file
    - (there are some other files generated by Visual Studio that have been omitted)

Dependency directory properties file

In order to reference the locations of external dependencies with Visual Studio macros, the file *DependencyDirs.props* has been specified in the solution and located in the *xts_2_ros_bridge* source code directory. This file contains the following macro definitions:

- XtsDir – the Xts API
  `\Users\jplba\workspace\barefoot_rover\Barefoot_Rover\tools\Xiroku\XtsApi-2.13\`
- DependencyDir – the directory in which other dependencies are located:
  `$(ProjDir)\dependencies\`
- libConfigDir – location of the libConfig package
  `$(DependencyDir)\libconfig\`
- NLohmanJsonDir
  `$(DepenencyDir)\json\`

Post-Build Event

A post-build event fires at the end of each build to copy the dll's for *XtsApi* and *libConfig*, the *XtsApi.ini* file and the *xts_2_ros_bridge.cfg* file to the build directory.