Readme for Conversion of CRCL XML into ROS Message

Michaloski, John L.

7/21/2016 4:32:00 PM

This document presents the nistcrcl Robot Operating System (ROS) package for translating commands and status between Canonical Robot Control Language (CRCL) and ROS.

This is a test 1 of Bsubscripts123 and Asuperscripts456.

This is a test 2 of Bsubscripts789 and Asuperscriptsabc.

# Background

Firstly, the nistcrcl ROS package frames CRCL message. Detecting an CRCl message is not trivial as there is not an ending marker (i.e., "0" or linefeed) to detect. If a CRCl XML message is detected, it is placed onto a synchronized queue. Upon receipt of a message, a signal is sent to another thread to execute an "Action()" method. The Action method interprets this CRCl XML command message by translating CRCL robot data structures (e.g., joints and pose) and units of representation (e.g., millimeter and inches) into ROS standard representation. Finally, after the message has been decoded into a canonical ROS command, it is published as a ROS message on the crcl\_command topic. Handling of robot status is done similarly, only in reverse.

# Running

There is a launch file crclserver.launch which allows ip and port arguments:

roslaunch nistcrcl crclserver.launch port:=64444

Side effects include starting a roscore and selecting the Fanuc robot description.

# Canonical Robot Control Language (CRCL) Background

Canonical robot command language (CRCL) is part of the robot research at NIST. CRCL is a messaging language for controlling a robot. CRCL commands are executed by a low-level device robot controller. The usual source of CRCL commands is a plan/program execution system. CRCL is intended for use with devices typically described as industrial robots and for other automated positioning devices such as automated guided vehicles (AGVs). An AGV with a robotic arm attached may be regarded as a single robot responding to a single stream of CRCL commands or as two robots responding to two separate streams of CRCL commands.

Although CRCL is not a programming language, the commands are in the context of a session consisting of getting ready for activity, performing activities, and becoming quiescent. CRCL commands may be collected in files for testing purposes, but executing such files (by giving the commands in the order they occur in the file) is not be the normal operating mode of a robot. Because robots operate in uncertain and changing environment, the reliance on sensors to adjust for such disturbances makes canned scripts ineffective under real conditions.

CRCL models a status message from a low-level robot controller. Status includes the position and orientation (Poses) that are the subject of CRCL commands. If any joint status reporting is done, it is assumed that the system sending canonical commands and the system executing them both know the kinematics of the robot and have the same numbering system for the joints, starting with 1. The two systems also have the same understanding of where the zero point is and which direction is positive for each joint. Status items for joints must be configured using a CRCl ConfigureJointReports command. For each joint for which anything is to be reported, ConfigureJointReports specifies:

* whether joint position should be reported
* whether joint torque or force should be reported
* whether joint velocity should be reported

During a CRCL session, until a ConfigureJointReports command has been executed that sets the reporting status for a joint, default joint status is reported for that joint. The ConfigureJointReports command may be used more than once during a session to change joint status reporting.

## Nistcrcl package Architecture

ROS Topic: crcl\_command

Python CRCL  
Command Generator  
Status Reader

nistcrcl  
ROS Package  
Crcl2Ros

ROS Package

Port 64444  
CRCL XML

ROS Topic: crcl\_status

* nistcrcl Package – ROS C++ class to handle all socket communication with the CRCL clients. Relies on Boost Asio for low level IO handling. Within the Boost Asio framework, the class CAsioServer handles the intracies of socket connections and disconnections, while the class class CAsioSession handles initiating, reading, buffering and queing messages. CRCL XML commands are posted as complete strings on a message queue. The class CCrcl2Ros handles the translation of Crcl messages into ROS topic commands. The class CCrcl2Ros also handles status from ROS and updates Crcl world model for status replies.
* Boost.Asio – Boost.Asio is a cross-platform C++ library that was used for network I/O programming because it abstracts the low level socket handling functionality (Schäling, n.d.). Boost Asio is used for communication over sockets to the CRCL controller. Boost Asio is very robust with examples and issue solutions pervasively found on the Internet.
* Crcl2ROS relies on CodeSynthesis and the Xerces XML DOM parser. Xerces was used to parse the CRCL XML. CodeSynthesis provided support for translating CRCL XML into a C++ representation. Given the C++, it was then simple matter to translate the intent of CRCL status message into a corresponding ROS intent. Code Synthesis “XSD” tool was used to generate the corresponding C++ classes from the CRCL XSD.
* RCS implements a few utility classes: shared message queue, timer, timed thread, and message event–driven thread. ROS provides support for the timing utilities, but often the requirement to include the entire ROS middleware is a bloated solution.
* Python client Crcl testing program was created to send Crcl commands to the Crcl2Ros package (assume at minimum roscore is running), and also handles Crcl status messages. This Python program does not require ROS.
* Python client ROS testing program was created to send/receive ROS topics: crcl\_commands and ROS crcl\_status. This Python program does require ROS.

## How does the Crcl2Ros class work?

In the main C++ program, a Crcl2Ros class is declared (with a reference to the main ROS node handle reference, i.e., nh). Crcl2Ros reads framed Crcl XML messages, interprets them, and translates into ROS. Then, it publishes the Crcl command as a ROS command using the nistcrcl/crcl\_command topic.

CCrcl2RosMsg crcl2ros(nh);

// This thread handles new XML messages received from crcl asio socket.

CAsioCrclServer crclServer(myios,(CAsioMessageQueueThread\*)&crcl2ros);

session.Start(); // start the thread

CCrcl2RosMsg uses the RCS CMsgQueueThread template to run asynchronously. CMsgQueueThread combines a synchronized message queue with a threading model. CMsgQueueThread defines Init and Action methods that are virtual overloaded methods. CMsgQueueThread overrides the empty Init and Action methods to interpret and publish ROS messages in a sequence of operations.

CMsgQueueThread first calls Init() which is overridden to initialize CRCL delegate (which interprets XML message and if a legal command is found, the message is transformed and queued onto ROS canonical command topic.) And, Init also subscribes to the crcl\_status topic and advertises the crcl\_command topic for communication with other ROS packages.

void CCrcl2RosMsg::Init() {

Name() = "CCrcl2RosMsg";

crclinterface = boost::shared\_ptr<Crcl::CrclDelegateInterface>(

new Crcl::CrclDelegateInterface());

crclinterface->SetAngleUnits("DEGREE");

crcl\_pub = \_nh.advertise<nistcrcl::CrclCommandMsg>("crcl\_command", 10);

crcl\_sub = \_nh.subscribe("crcl\_status", 1000, crclstatusCallback);

}

The CMsgQueueThread method Start starts the thread which calls Init(), waits for a Stop method, and connects a signal to activate the Action() loop. The connect code uses the boost bind method to tie the signal to the current class method Action.

void Start() {

Threads().push\_back(this);

ThreadGroup().create\_thread(boost::bind(&CMsgQueueThread<T>::Cycle, this));

SigAction.connect(bind(&CMsgQueueThread<T>::Action, this));

}

When a CRCL message is received, the message is pushed onto the queue, and the SigAction signal causes the “Action” method to run. This sequence of operation was defined as an event driven so that the receipt of a message causes the Action method to be called immediately, with no latency between message receipt and message handling.

virtual void AddMsgQueue(T t) {

{

boost::mutex::scoped\_lock lock(m);

CMessageQueue<T>::xml\_msgs.push\_back(t);

}

SigAction();

}

Within the CCrcl2RosMsg class the Cycle method calls Init method, and then waits for a stop condition action to occur to end the thread. Before finishing the thread it calls the virtual method Cleanup().

void Cycle() {

\_bDone = false;

Init();

boost::mutex::scoped\_lock lock(stopMutex);

try {

stopCond.wait(lock); // wake up waiting action

} catch (...) { std::cout << "Unhandled exception - " << Name().c\_str() << std::endl; }

Cleanup();

\_bDone = true;

}

The CCrcl2RosMsg Action() method determines if new CRCL command has been placed on the message queue, even though a CRCL message must have been placed on the queue or the method would not have been called. Action pops the message and splits the message into two parts: the Crcl command and the pointer to the client session which initiated the command, in case the Crcl command requires a response be sent as part of the command from the client (e.g., status reply is reqruied for a Crcl GetStatus command).

CCrcl2RosMsg Action()

{

. . .

if (CAsioCrclSession::InMessages().SizeMsgQueue() > 0) {

CrclMessage msg = CAsioCrclSession::InMessages().PopFrontMsgQueue();

std::string crclcmd = boost::get<0>(msg);

\_pSession = boost::get<1>(msg);

The CRCL command (and CRCL data structures) is interpreted and if necessary translated into a ROS command (with ROS data structures). First the CRCL XML is translated into C++ representation with the DelegateCRCLCmd. Then, if a command DelegateCRCLCmd translates into ROS representation.

Crcl::CrclReturn ret = crclinterface->DelegateCRCLCmd(crclcmd);

If the return code indicates that the CRCL command is requesting a status message, the latest local CRCL world model is wrapped into a CRCL status reply and sent via a synchronous write to the client:

if (ret == Crcl::CANON\_STATUSREPLY) {

// Generate status reply message

std::string sStatus

= Crcl::CrclClientCmdInterface().GetStatusReply( &crclinterface->crclwm );

\_pSession->SyncWrite(sStatus);

}

}

Next these canonical “ROS” commands are translated into a ROS topic message. ROS requires standard units (meter, radians) while CRCL allows mm, angles, etc. So the Crcl commands are translated into ROS representation. Some Crcl commands such as assigning linear or angular units are self-contained and do not generate a ROS message, i.e., millimeter or inch linear units are translated locally into ROS meter length standard units. So, there Crcl commands are not passed across the crcl\_command topic.

First, the canonical “ROS” commands queue is checked for new commands.

if (RCS::cmds.SizeMsgQueue() > 0) {

RCS::CanonCmd cc = RCS::cmds.PopFrontMsgQueue();

If one is found then we know that it is not a local units setting but an actual motion command. If a ROS canonical message has been generated, then a new ROS command topic message is created :

nistcrcl::CrclCommandMsg rosmsg;

Currently, the contents o f CrclCommandMsg is a union of all CRCL motion related commands (excluding tolerance, velocity, acceleration, force parameters at this time, but are required for path planning.) Below is a sample of translating a Crcl actual joint (move joint) and a Crcl move to pose commands into the ROS crcl\_command topic:

if (cc.cmd == RCS::CANON\_MOVE\_JOINT)

{

rosmsg.crclcommand=actuatejoints;

rosmsg.joints=cc.joints;

}

else if (cc.cmd == RCS::CANON\_MOVE\_TO)

{

rosmsg.crclcommand=moveto;

rosmsg.finalpose.position.x=cc.pose.getOrigin().x();

rosmsg.finalpose.position.y=cc.pose.getOrigin().y();

rosmsg.finalpose.position.z=cc.pose.getOrigin().z();

rosmsg.finalpose.orientation.x=cc.pose.getRotation().x();

rosmsg.finalpose.orientation.y=cc.pose.getRotation().y();

rosmsg.finalpose.orientation.z=cc.pose.getRotation().z() ;

rosmsg.finalpose.orientation.w=cc.pose.getRotation().w();

}

. . .

Again, numerous ROS representations of pose position and orientation exist. To use existing ROS message structures, the geometry pose (with point and quaternion sub definition) was used. Readers are referred to <https://github.com/ros/common_msgs> to view a list of common ROS messages.

Finally, if a ROS message was generated it is published on the topic (nistcrcl/ crcl\_command):

// publish ros message if found corresponding crcl command

if(rosmsg.crclcommand!=noop)

{

ROS\_INFO("ROS command: [%s] ", rosmsg.crclcommand);

crcl\_pub.publish(rosmsg);

}

## CRCL Socket Communication

As discussed, there is also no terminating character (such as zero) in a CRCL message. Also, CRCL messages can also be of different buffer sizes. So framing the message requires buffering each message, such that the end of a status message is detected with a closing XML tag, and this message may be divided with some of the buffer belonging to the previous or next message. Unfortuneately, the last write of the CRCL XML message need not satisfy an asynchronous condition - such as buffer full or matching character. Since there is no CRCL message termination condition, a deadline timer was used to stop asynchronous reading and cancel the read. Likewise, often two CRCL will be combined into one aynchronous read operation, so that these two message must be separated by the CRCL streaming reader.

The CRCL client establishes a connection using an assigned socket and port number with the CRCL server once. The method to send CRCL command from a client is shown in the (b) portion of Figure 1. Upon receipt of a Crcl command that requires a status reply, then the nistcrcl package responds with the CRCL XML status message as shown in the (c) portion of Figure 1.

Write CRCL status

Over socket

CRCL

XML

Client

nistcrcl

ROS package

Connect CRCL socket

nistcrcl

ROS package

Write CRCL command

Over socket

(a)

(b)

CRCL

XML

Client

nistcrcl

ROS package

CRCL

XML

Client

(c)

Figure CRCL Communication to nistcrcl ROS package

We will assume that the CRCL Client has connected to the nistcrcl package socket. Figure 2 shows the components involved in communication.The CRCL Client class initiates communication to first send a command to the nistcrcl package. Using Boost Asio, the nistcrcl package has the class CAsioSession which asynchronously reads the command from the CRCL client. When the CAsioSession has read a complete message it queues this message onto the message queue. Since the message queue is a shared resource, and multiple threads share this resource, a mutex is used to lock the contents for one thread at a time access. The latest CRCL command message is retrieved and then the CRCL data expressed in XML can be reinterpreted before storing into the ROS crcl\_command topic. The translation uses CodeSynthesis to parse the CRCL XML message and translate into C++. Once in C++, it is translated into an equivalent ROS representation.

Python CRCL  
Command Generator  
Status Reader

Crcl2Ros

Boost ASIO  
CAsioServer  
(Handle connections)  
CAsioSession   
(Handle Messages)

Asynchronous Read

CCrcl2Ros  
Unqueue Message  
Convert C++ Rep.  
Translate into ROS  
Signal Action()

crcl\_command  
ROS topic

Action()

RCS  
Message  
Queue

Figure Communication Sequence with ROS crcl\_command topic

A major portion of the CAsioSession was the handling of the socket stream interface to CRCL client. The Boost Asio library was used. Boost Asio uses the Proactor design pattern for event handling in which long running activities are running in an asynchronous part (Pyarali, Harrison, Schmidt, & Jordan, 1997). The major complaint with Boost Asio is that it is more difficult to due to the separation in time and space between operation initiation and completion. Applications may also be harder to debug due to the inverted flow of control and difficulty in understanding when a problem occurred. In the Proactor model, a completion handler is called after the asynchronous event has triggered. In this software pattern, the connect asynchronous event is registered in Boost Asio, and when the connect event happens a “callback” handler method is invoked. Within this connection “callback” handler, the asynchronous read event is setup. This asynchronous read event setup function is used to specify how a stream will be asynchronously read a certain number of bytes of data from a stream. The asynchronous setup itself returns immediately. But, the asynchronous event detection will continue until one of the following conditions is true:

* The supplied buffer is full. That is, the bytes transferred are equal to the sum of the buffer size.
* An error occurred. Errors include socket disconnection, or asynchronous read cancelation by the deadline timer. Socket disconnects cause a discontinuation of reading and the asynchronous connection trigger is then restarted.

Then the asynchronous read event handler will be called. Within the asynchronous read event handler, the asynchronous read event setup must be done again, or no more bytes will be read from the stream. Of note, often the condition of completely filling the buffer is impossible as receiving the exact amount of buffer size is impractical. Because of this, deadline timers are incorporated into Boost Asio to terminate asynchronous reads even if the triggering event (full buffer) has not occurred. So a deadline timer is used to cancel a socket asynchronous read when it expires.

Boost Asio provides io\_service, which is a singleton class for servicing I/O. Every program based on Boost.Asio uses an object of type io\_service. There is one Asio “io\_service” per application program and from our effort it helps if all the asynchronous operations are run in the same thread as this io\_service object. Each asynchronous call in Boost Asio is enabled by the io\_service methods run, run\_one or poll. After all the asynchronous operations were placed in the same thread, the io\_service communication responded better. However, Boost Asio was too efficient so an io\_service run\_one method was combined with a sleep and yield to allow other threads to run and to slow down the Boost Asio operation.

Figure 3 shows the call sequence involved with the CRCL status reading. Of note, are the synchronous write to send the “Init” during the connection handling and the “GetStatus” command messages sent at the start of every new read of the CRCL simulator status. These are the only Boost Asio operations that were synchronous. Overall, the Boost Asio asynchronous connect operation could wait indefinitely upon startup of the CRCL simulator listener. This is as intended. It is unclear how often Boost Asio tests the socket for the CRCL listener. Upon connection of the socket, the handler for the async connect event is called and it calls the async read and async periodic timer (2 seconds). Either 1) the socket stream is read and the periodic timer is canceled, or 2) the periodic timer expires and the socket async read is canceled, which calls the read handler to see if it has read any bytes or is just waiting for a termination condition. If bytes have been read, there are buffered. In either case, the asynchronous read is called again.



Figure Asio Communication Sequence

## CRCL Communication Code Review

The nistcrcl package uses a two thread model 1) one thread to handle communication with CRCL client(s) and 2) the other thread to handle the ROS interface. Figure 4 shows the main thread spawns thread 2, which handles the ROS message streaming. It was found that if all the Boost Asio operation were not in the same thread, unpredictable results occurred. Unfortunately, it is very difficult to debug problems in Boost Asio, as most of the operation is hidden in a thread, and when no events occur, there is nothing to debug.

Figure 4 shows the code that run in the two threads. Thread 1 spawns Boost Asio event callbacks, and then loop running Boost Asio io\_service to handle all the asynchronous communication events, as well as frame and queue any CRCL XML messages. Thread 2 handles the ROS communication over topics. This framework forms the basis of the nistcrcl code, which will be described herein.

Thread 1

Thread 2

async ASIO  
listen connection  
connect  
async read  
queue message  
(notify Crcl2ROS)

Crcl2ROS  
parse XML -> C++  
translate Crcl into ROS   
publish crcl\_command

Figure Threads – Boost Asio and Crcl2ROS

The Boost Asio thread is responsible for handling the asynchronous reads of CRCL status messages. The first step, is setting up the asynchronous connect event. This is done by calling the CAsioSession::Connect() routine. The connect routine uses a previously stored Internet Protocol (IP) address and socket number, for which it now uses as an endpoint in which to connect. The IP can be a name or a physical address.

void CAsioSession::Connect() {

tcp::resolver resolver(\*\_io\_service);

tcp::resolver::query query(\_ipv4.c\_str(), \_port.c\_str());

tcp::resolver::iterator endpoint\_iterator = resolver.resolve(query);

\_socket->async\_connect(\*endpoint\_iterator, boost::bind(&CAsioSession::HandleConnect,

this,

boost::asio::placeholders::error));

}

Upon detection of a socket connect, the routine CAsioSession::HandleConnect is called. This routine checks for any errors, if the error has been determined to be connection refused, the Connect routine is called again. There must always be asynchronous events for Boost Asio to detect and handle, or it will terminate. Assuming the socket has been successfully connected, the StartAyncRead() routine will be called for Boost Asio to detect an asynchronous read event and call the handle as defined in StartAyncRead(). In order for the CRCL to operate properly, a CRCL “Init” command is sent first, and usually expects a robot status reply.

void CAsioSession::HandleConnect(const boost::system::error\_code& error) {

if (!\_socket->is\_open()) {

Disconnect();

return;

}

if (error == boost::asio::error::connection\_refused) {

Globals::Sleep(\_nMSServerConnectRetry);

Connect();

return;

} else if (error == boost::asio::error::already\_started) {

// not really an error?

}

// On error, return early.

else if (error) {

Disconnect();

return; // giving up

}

StartAyncRead();

SyncWrite(CrclInterface().CRCLInitCanonCmd());

}

The StartAyncRead routine handles the setting up the asynchronous read event for Boost Asio to the complete filling of the read buffer provided to boost::asio::async\_read. Because the conditional event of complete filing of the read buffer may not happen (only if the each status message requires multiple reads can the buffer be assured to be full, and the case where the boundary of the read buffer and the status CRCL read are of the same length is not guaranteed), a deadline periodic timer is setup with TimerReset() to time out the read after a certain amount of time and will cancel the asynchronous read. The routine boost::asio::async\_read is called with the TCP socket, the read buffer, the condition, and then a callback method is provided using the Boost bind mechanism (which calls this class CAsioSession method bytesToRead with parameters for error and number of read bytes transferred by Boost Asio.)

void CAsioSession::StartAyncRead() {

try {

Globals::DebugMessage("StartAyncRead\n");

if (!\_socket->is\_open()) {

DebugBreak();

}

TimerReset();

boost::asio::async\_read(\*\_socket,

boost::asio::buffer(data\_, max\_length),

boost::asio::transfer\_at\_least(1),

boost::bind(&CAsioSession::bytesToRead,

this,

boost::asio::placeholders::error,

boost::asio::placeholders::bytes\_transferred));

if(\_bSendGetStatus)

{

std::string statcmd = CrclInterface().CRCLGetStatusCmd();

SyncWrite(statcmd);

\_bSendGetStatus=false;

}

} catch (boost::exception & ex) {

std::cerr << boost::diagnostic\_information(ex);

Disconnect();

} catch (...) {

Disconnect();

}

}

CAsioSession::TimerReset() sets a deadline time to implement the timeout after 2000 milliseconds or 2 seconds. Again, a callback method is provided using the Boost bind mechanism (which calls this class CAsioSession method wait\_callback with parameters for error and the socket that timed out.)

void CAsioSession::TimerReset() {

boost::system::error\_code ec;

int n = \_timer->expires\_from\_now(boost::posix\_time::milliseconds(2000), ec);

\_timer->async\_wait(boost::bind(&CAsioSession::wait\_callback, this, \_1, \_socket));

}

The periodic timer event handler (i.e., the wait\_callback method) checks to see if the timer expired, a general error occurred, or the asynchronous read handler canceled this timer event. If an error occurred, the method forward the error to the waiting asynchronous reader.

void CAsioSession::wait\_callback(const boost::system::error\_code& error, socket\_ptr \_socket) {

if (error == boost::asio::error::operation\_aborted) {

return; // canceled by asynchronous read event handling

} else if (error) {

// Unclear why here – maybe disconnect?

std::cout << "read\_timeout Error - " << error.message() << std::endl;

return;

}

\_socket->cancel(); // will cause read\_callback to fire with an timeout error

}

When event signifying that the Java CRCL status socket has finished reading, either thru fulfilling the buffer condition or the periodic timer canceling the socket read, then the routine bytesToRead() is called. This routine first cancels the periodic timer. If there was an error, the socket is Disconnected, which will cause Boost Asio to start listening for a new connection event. Next the read buffer is appended to the current string buffer. If no end tag has been found so far, using the FindLeadingElement(\_current) routine with the current buffer (includes all characters in the current read) is called. Finally, the BufferHandler() is called to determine if a complete message has been read (by looking for the matching end tag, in this case, "</CRCLStatus>"). Unless an error has occurred that is not the deadline timer, the StartAyncRead() is called again to insure that Boost Asio continues reading.

size\_t CAsioSession::bytesToRead(const error\_code& error, size\_t bytes\_read) {

size\_t result = bytes\_read;

\_timer->cancel(); // cancel periodic timer

if (error == boost::asio::error::eof || boost::asio::error::connection\_reset == error) {

Disconnect();

return 0;

} else if (error == boost::asio::error::operation\_aborted) {

} else if (error) {

return 0;

}

if (bytes\_read > 0) {

AppendBuffer(std::string(data\_, data\_ + bytes\_read));

if (\_endtag == NonsenseTag()) {

\_endtag = FindLeadingElement(\_current);

}

BufferHandler(\_endtag);

}

StartAyncRead();

return result;

}

The routine does detection of a complete status message. It searches the buffer string for the ending tag (which is not a dummy tag if a starting tag has been detected.). If it find the ending tag, it divides the message, appending the completed status message onto the queue, and then saving the remaining part of status message for the next read. It is at this point, the end tag is reset to a dummy value, and the signal to send a GetStatus to the CRCL simulator is set.

bool CAsioSession::BufferHandler(std::string endtag) {

std::size\_t found;

if ((found = \_current.find(endtag)) != std::string::npos) {

found = found + endtag.size();

\_next = \_current.substr(found);

\_current = \_current.substr(0, found);

AddMsgQueue(\_current);

\_current.clear();

TagReset();

\_bSendGetStatus=true;

return true;

}

return false;

}

The routine FindLeadingElement(std::string xml) is being documented only because it uses Boost regex. Given an XML string, this routine will find the matching element tag after the “<?xml” header. If found, it returns the ending tag corresponding to the starting tag, otherwise it return a Nonsense tag.

std::string CAsioSession::FindLeadingElement(std::string xml) {

boost::match\_results<std::string::const\_iterator> matchResult;

bool found;

boost::regex e("<[A-Za-z0-9\_]+");

found = boost::regex\_search(xml, matchResult, e);

if (found) {

std::string elem(matchResult[0]);

elem.insert(1, 1, '/');

elem = Globals::Trim(elem);

elem.append(">"); // append closing tag symbol

return elem;

}

return NonsenseTag();

}

# Installing Prerequisites

## Installing XercesC with Ubuntu

You can use apt-get to install the packages for the library and the dev files. Then to use them in your C/C++ programs you simply #include the appropriate headers and link with the library when compiling/linking.

sudo apt-get update

apt-cache search libxerces

sudo apt-get install libxerces-c3.1 libxerces-c-dev

Need to include the file path for searching include headers in the CMakeLists.txt file (required in ROS):

include\_directories(/usr/include/xercesc)

Link library in CMakeLists.txt:

link\_directories(/usr/lib/x86\_64-linux-gnu/)

Need to link against libxerces.a in CMakeLists.txt:

target\_link\_libraries(nistcrcl

libxerces-c.a

${catkin\_LIBRARIES}

${Boost\_LIBRARIES}

)

## Installing CodeSynthesis XSD

http://www.codesynthesis.com/products/xsd/download.xhtml 1. Chose the linux deb install file that matches your computer (below 64 bit amd). 2. Download xsd4.0.0-1amd64.deb and it will say open with Ubuntu Software Center 3. Click to install, authenticate and add /usr/include/xsd/cxx/xml as include path.

Need include file path in CMakeLists.txt:

include\_directories(/usr/include/xsd/cxx/xml)

If you cannot run Ubuntu software centerto install CodeSynthesis, you can download the source and install it. You need to go to the web page: http://www.codesynthesis.com/products/xsd/download.xhtml and select:

xsd-4.0.0-x86\_64-linux-gnu.tar.bz2

It will be saved into /usr/local/downloads, but you can save it anywhere. Then cd to where you saved it, and do this:

tar --bzip2 -xvf xsd-4.0.0-x86\_64-linux-gnu.tar.bz2 (dash-dash bzip2, dash-xvf)

It will create a directory xsd-4.0.0-x86\_64-linux-gnu.

Make a symbolic link:

ln -s <path/to/xsd-4.0.0-x86\_64-linux-gnu/libxsd/xsd /usr/local/include/xsd

e.g., ln -s /usr/local/xsd-4.0.0-x86\_64-linux-gnu/libxsd/xsd /usr/local/include/xsd

# Testing

A testing scenario was developed that is similar to typical deployment. A minimalist approach was taken, which can often be difficult in ROS. Initially, only roscore was to be spawned to integrate ROS functionality, but it was determined that a robot\_description parameter was required in order to establish names for the joints. (CRCL only used sequential numbered actuator indexes, while ROS uses names to identify links and joints using the robot description.) As such a roslaunch file was used to start roscore and establish two ROS parameters: robot\_description and controller\_joint\_names as defined below:

<launch>

<param name="robot\_description" command="$(find xacro)/xacro.py $(find fanuc\_lrmate200id\_support)/urdf/lrmate200id.xacro" />

<rosparam command="load" file="$(find fanuc\_lrmate200id\_support)/config/joint\_names\_lrmate200id.yaml" />

</launch>

The roslaunch utility starts roscore which starts up:

* ROS Master
* ROS Parameter Server
* rosout logging node

The roscore can run indefinitely. At the same time a crcl\_client Python program was started that generated CRCL XML commands to the nistcrcl package to receive and interpret. The nistcrcl package communicates with another Python test program to read and write "robot" status/commands.

# 

The crcl\_client Python program contains code to prevent it from proceeding until a socket connection with the CRCL server has been established.

## Coordinated Testing of nistcrcl package Bash Script

A bash script was developed to test the CRCL command communication through nistcrcl executable that is then read as a ROS crcl\_command in a ROS python package. The location of the script is:

. . ./nistcrcl\_ws/src/nistcrcl/scripts/runmultiterm.bash

The script assumes you have done a “catkin build” of the nistcrcl\_ws to compile the executable and have sourced the devel/setup.bash script to setup the ROS environment variables. The script uses gnome-terminal to bring up 4 terminals:

1. roscore which launches the ROS master program as well as set the robot\_description and joint names parameters
2. nistcrcl which executable to handle communication between CRCL and ROS topics
3. cannedcrclclient.py is a python program which sends CRCL XML commands to the nistcrcl (and receives CRCL status)
4. crclfeedbacktest.py under the ROS testcrcl\scripts is a ROS python program which reads crcl\_command topic ROS message and writes robot status out the crcl\_status topic ROS message.

### Bash script to coordinate multiple nistcrcl shells - runmultiterm.bash

A shell script to open gnome terminal with multiple tabs, with each tab running a separate shell command.

#!/bin/bash

source /opt/ros/indigo/setup.bash

source /usr/local/michalos/nistcrcl\_ws/devel/setup.bash

# Determines whether to use netbeans or ros to run nistcrcl package

#crcl="run"

cmd=( gnome-terminal )

# This roslaunch will load a robot description so there are actual joint names. Otherwise no joint names!

# It also loads the controller\_joint\_names yaml file

cmd+=( --tab-with-profile=Default --title="roslaunch" -e "/opt/ros/indigo/bin/roslaunch testcrcl simplelaunch.launch " )

# This is the ROS package to translate CRCL to/from ROS message topics

# For debugging, netbeans is used as a IDE

if [ "$crcl" = "run" ]

then

cmd+=( --tab-with-profile=Default --title="nistcrcl" -e "/usr/local/michalos/nistcrcl\_ws/devel/lib/nistcrcl/nistcrcl " )

fi

# This canned python script will keep attempting to connect until connected,

# then will move joint 0 +90 to -90 every 6 seconds

cmd+=( --tab-with-profile=Default --title="python canned test" -e 'python /usr/local/michalos/nistcrcl\_ws/src/testcrcl/scripts/cannedcrclclient.py' )

# This python program will read ROS crcl command and echoa status

#cmd+=( --tab-with-profile=Default --title="python feedback test" -e 'python /usr/local/michalos/nistcrcl\_ws/src/testcrcl/scripts/crclfeedbacktest.py' )

"${cmd[@]}"

This script does the following:

1. opens gnome-terminal
2. appends tab command which opens a tab executes a script following the -e option
3. for example, --title=" roslaunch " opens a tab with title " roslaunch " that runs the ROS master program roslaunch, -e "/opt/ros/indigo/bin/roslaunch".

The script is best exited by hitting ^C in each shell and then can exit the gnome-terminal window by closing the window which kills the nistcrcl command which hangs on a control C.

### Python CRCL Test Program - cannedcrclclient.py

A Python program was written to act as a CRCL client. That is, the Python code establishes a TCP/IP socket connection to a CRCL server (in the test case it is the nistcrcl ROS package executable). This Python program is not an exhaustive test and does not have coverage for all potential CRCL commands. It repeatedly attempts to connect to the server, and once a connection is established it sends a CRCL ActuateJoint command to move Joint 1 (numbered from 1) from -90° to +90°.

A Python class CrclClientSocket handles the connection, synchronous sending and synchronous receiving of CRCL XML socket communication. CrclClientSocket repeatedly attempts to connect to the server as defined by a host and port (typically 127.0.0.1 and port 64444). Upon failure to connect, the CrclClientSocket class will keep recursively calling connect until the server listener has been established.

import rospy

import socket

import sys

import math

#from mathutils import Matrix

import numpy as np

from numpy import matrix

import time

from xml.dom import minidom

import os.path

class CrclClientSocket:

def \_\_init\_\_(self, host, port):

self.host=host

self.port=port

self.stopconnecting=False

# quit when find </CRCLStatus>

self.End='</CRCLStatus>'

self.nextdata=''

def connect(self):

try:

if(self.stopconnecting):

return

self.sock = socket.socket(

socket.AF\_INET, socket.SOCK\_STREAM)

self.sock.connect((self.host, self.port))

except socket.error, msg:

print 'Failed to create socket. Error code: ' + str(msg[0]) + ' , Error message : ' + msg[1]

time.sleep( 5 )

self.connect()

def disconnect(self):

self.sock.close()

def syncsend(self, msg):

print msg

sent = self.sock.send(msg)

if sent == 0:

self.disconnect()

self.connect()

# raise RuntimeError("socket connection broken")

# http://code.activestate.com/recipes/408859/

def syncreceive(self, end):

self.End=end # '</CRCLStatus>'

data=''

alldata=self.nextdata

while True:

data=self.sock.recv(8192)

if data == 0:

alldata='' # empty string

return

alldata=alldata+data

if self.End in alldata:

alldata=alldata[:alldata.find(self.End)]

self.nextdata=data[data.find(self.End)+1:]

break

return alldata # ''.join(total\_data)

CrclClientSocket handles the socket communication of CRCL code. For example, the CRCL actuate joints command is encapsulated by the CrclActuateJoints method, calling it with a command number, joint number, joint position, joint velocity and joint acceleration. Unfortunately, only one joint at a time can be commanded by CRCL XML with this method; however, this is adequate for testing purposes. Below Python format statement is used to build and return a CRCL XML string.

def CrclActuateJoints(cmd, num, pos, vel, acc):

return '''<?xml version="1.0" encoding="UTF-8"?>

<CRCLCommandInstance

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="../xmlSchemas/CRCLCommandInstance.xsd">

<CRCLCommand xsi:type="ActuateJointsType">

<CommandID>{}</CommandID>

<ActuateJoint>

<JointNumber>{}</JointNumber>

<JointPosition>{}</JointPosition>

<JointDetails xsi:type="JointSpeedAccelType">

<JointSpeed>{}</JointSpeed>

<JointAccel>{}</JointAccel>

</JointDetails>

</ActuateJoint>

</CRCLCommand>

</CRCLCommandInstance>'''.format(cmd,num, pos, vel, acc)

There is a command line interpreter, which can read a command line such as:

"j 1 1.7 0 0"

And then call the CrclActuateJoints method with joint 1 at position 1.7, velocity and acceleration of zero. The command number (i.e., cmd) is handled separately and incremented with each command by the interpreter.

s=CrclActuateJoints(str(cmd), tokens[1], tokens[2], 4.0, 1.0)

mysocket.syncsend(s)

The Python code below creates a CrclClientSocket class which does the connection host and port (localhost or 127.0.0.1 and port 64444), then connects to this TCP/IP socket, and then enter a loop to sends a CRCL ActuateJoint command to move Joint 1 (numbered from 1) from -90° (-1.7 radians) to +90° (+1.7 radians).

mysocket = CrclClientSocket("localhost", 64444)

mysocket.connect()

while not rospy.is\_shutdown():

parseline("j 1 1.7 0 0")

time.sleep(6)

parseline("j 1 -1.7 0 0")

time.sleep(6)

mysocket.disconnect()

### Python CRCL Test Program - crclfeedbacktest.py

The feedback ros client reads the translated CRCL commands and simulates status by publishing to the crcl\_status topic.

import sys

import copy

import rospy

import moveit\_commander

import geometry\_msgs.msg

from sensor\_msgs.msg import JointState

from moveit\_msgs.msg import KinematicSolverInfo, PositionIKRequest

from moveit\_msgs.srv import GetKinematicSolverInfo, GetKinematicSolverInfoResponse, GetPositionFK, GetKinematicSolverInfo, GetPositionFKRequest, GetPositionFKResponse

from geometry\_msgs.msg import Pose

from std\_msgs.msg import Header

from nistcrcl.msg import CrclCommandMsg

from nistcrcl.msg import CrclStatusMsg

import time

import genpy

import roslib.message

import thread

import time

import subprocess

########################################

# This uses the Kinematic services which are now hidden in moveit

class Robot():

def \_\_init\_\_(self):

self.joint\_state = JointState()

self.linknames=[]

rospy.wait\_for\_service('get\_fk')

#rospy.wait\_for\_service('testkin/GetKinematicSolverInfo')

self.get\_fk\_proxy = rospy.ServiceProxy('get\_fk', GetPositionFK, persistent=True)

self.get\_fk\_solver\_info\_proxy = rospy.ServiceProxy('get\_fk\_solver\_info', GetKinematicSolverInfo)

solver\_info = self.get\_fk\_solver\_info\_proxy()

rospy.loginfo(solver\_info)

for name in solver\_info.kinematic\_solver\_info.joint\_names:

self.joint\_state.name.append(name)

rospy.loginfo("Adding joint " + str(name))

for name in solver\_info.kinematic\_solver\_info.link\_names:

self.linknames.append(name)

rospy.loginfo("Adding link " + str(name))

#self.joint\_state.position = [0] \* len(self.joint\_state.name)

for name in self.joint\_state.name:

self.joint\_state.position.append(0.0)

def JointIndex(self, jointname):

return self.joint\_state.name.index(jointname)

def Update(self, jointvals) :

for i in range(0,len(jointvals.name)):

self.joint\_state.position[self.JointIndex(jointvals.name[i])]=jointvals.position[i]

def FK(self,jointvals):

self.request = GetPositionFKRequest()

self.request.fk\_link\_names = self.linknames

self.request.robot\_state.joint\_state = self.joint\_state # JointState()

self.request.robot\_state.joint\_state.header.frame\_id = 'base\_link'

# self.request.robot\_state.joint\_state.name = self.joints

# self.request.robot\_state.joint\_state.position = [0] \* len(self.joints)

# self.request.robot\_state.joint\_state.position[0] = jointvals.position[0]

# self.request.robot\_state.joint\_state.position[1] = jointvals.position[1]

# self.request.robot\_state.joint\_state.position[2] = jointvals.position[2]

# self.request.robot\_state.joint\_state.position[3] = jointvals.position[3]

# self.request.robot\_state.joint\_state.position[4] = jointvals.position[4]

# self.request.robot\_state.joint\_state.position[5] = jointvals.position[5]

self.request.header.frame\_id = "base\_link"

try:

response = self.get\_fk\_proxy(self.request)

#rospy.loginfo(response)

return response

except rospy.ServiceException, e:

print "Service call failed: %s" % e

self.FK(jointvals)

# This handles accepting a crcl command via subscribe to set goal joints and pose

class CrclCmd:

def \_\_init\_\_(self):

self.crclcommand=0

self.joints=JointState()

self.crclcommandnum=0

self.pose=Pose()

self.robot=Robot()

def pplist(self,list):

return ' '.join(['%5.3f'%x for x in list])

def callback(self, data):

self.joints=data.joints

print["{0}={1:0.4f}".format(self.joints.name[i], self.joints.position[i]) for i in range(0,len(self.joints.name))]

self.crclcommand=data.crclcommand

self.robot.Update(data.joints)

response=self.robot.FK(self.joints.position)

self.pose.position=response.pose\_stamped[0].pose.position

self.pose.orientation=response.pose\_stamped[0].pose.orientation

p= response.pose\_stamped[0].pose.position

print "Service success!!! Pose Position=" + self.pplist([p.x,p.y,p.z])

########################################

# This runs as a separate thread and simulates the update of the robot status

def updateStatusThread(pub, crcl):

status = CrclStatusMsg()

status.statuspose.position.x = 0.465

status.statuspose.position.y = 0.0

status.statuspose.position.z = 0.695

status.statuspose.orientation.x = 0.0

status.statuspose.orientation.y = 0.0

status.statuspose.orientation.z = 0.0

status.statuspose.orientation.w = 1.0

status.eepercent=1.0

while not rospy.is\_shutdown():

status.statusjoints=crcl.joints

status.statuspose=crcl.pose

status.crclcommandnum=crcl.crclcommandnum

pub.publish(status)

time.sleep(1)

def runProcess(exe):

p = subprocess.Popen(exe, stdout=subprocess.PIPE, stderr=subprocess.STDOUT)

while(True):

retcode = p.poll() #returns None while subprocess is running

line = p.stdout.readline()

yield line

if(retcode is not None):

break

if \_\_name\_\_ == '\_\_main\_\_':

print '============ Start ROS feedback of crcl topics...'

rospy.init\_node('crclfeedback', anonymous=True,log\_level=rospy.DEBUG)

crcl=CrclCmd()

for line in runProcess('rostopic list'.split()):

print line

rospy.Subscriber("crcl\_command", CrclCommandMsg, crcl.callback)

pub = rospy.Publisher('crcl\_status', CrclStatusMsg, queue\_size=10)

updateStatusThread(pub, crcl)

updateStatusThread(pub, crcl)

#thread.start\_new\_thread(updateStatusThread(pub, crcl))

# spin() simply keeps python from exiting until this node is stopped

rospy.spin()

FIXME: Echoing a true forward kinematics of the joint positions would be nice, and a ROS service has been developed to perform this task. Like many ROS version issues, the GetPositionFK service was built on "Deprecated" ROS technology – arm\_manipulation, so it requires moveit to be used. This was done, but it might be easier to use moveit commander in Python.

Using a the Spyder Python IDE in ROS was not successful. Python programs run fine when invoked from command line –due to environment variables, but without ROS environment setup could not find custom messages, even though source devel/setup.bash was invoked before starting Spyder.

import os

import subprocess

def PipedCmd(cmd):

task = subprocess.Popen(cmd, shell=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE )

data = task.stdout.read()

# data1 = task.stderr.read()

return data

dict = {}

cmd = '/bin/bash -i /usr/local/michalos/nistcrcl\_ws/src/nistcrcl/scripts/setup.bash'

envs = ExecuteShellCommand(cmd);

lines = envs. splitlines ()

for line in lines:

key, val = line.strip().split('=')

dict[key.strip()] = val.strip()

for key in dict:

os.environ[key]=dict[key]

# Appendix I nistcrcl Package Version Dependencies

|  |  |
| --- | --- |
| Package | Version |
| actionlib\_msgs | 1.11.8 |
| catkin | 0.6.16 |
| class\_loader | 0.3.1 |
| cpp\_common | 0.5.6 |
| eigen\_conversions | 1.11.7 |
| eigen\_stl\_containers | 0.1.4 |
| fcl | 0.3.2 |
| gencpp | 0.5.3 |
| genlisp | 0.4.15 |
| genmsg | 0.5.6 |
| genpy | 0.5.7 |
| geometric\_shapes | 0.4.3 |
| geometry\_msgs | 1.11.8 |
| kdl\_parser | 1.11.8 |
| libccd | 1.5.0 |
| message\_generation | 0.2.10 |
| message\_runtime | 0.4.12 |
| moveit\_core | 0.7.0 |
| moveit\_msgs | 0.7.0 |
| object\_recognition\_msgs | 0.4.1 |
| octomap | 1.6.9 |
| octomap\_msgs | 0.3.2 |
| orocos\_kdl | 1.3.0 |
| pluginlib | 1.10.1 |
| random\_numbers | 0.3.0 |
| resource\_retriever | 1.11.6 |
| rosbag\_migration\_rule | 1.0.0 |
| rosbuild | 1.11.11 |
| rosconsole | 1.11.16 |
| rosconsole\_bridge | 0.4.2 |
| roscpp | 1.11.16 |
| roscpp\_serialization | 0.5.6 |
| roscpp\_traits | 0.5.6 |
| rosgraph\_msgs | 1.11.1 |
| roslib | 1.11.11 |
| rospack | 2.2.5 |
| rostime | 0.5.6 |
| sensor\_msgs | 1.11.8 |
| shape\_msgs | 1.11.8 |
| srdfdom | 0.3.0 |
| std\_msgs | 0.5.9 |
| trajectory\_msgs | 1.11.8 |
| urdf | 1.11.8 |
| urdf\_parser\_plugin | 1.11.8 |
| urdfdom\_py | 0.3.0 |
| visualization\_msgs | 1.11.8 |
| xmlrpcpp | 1.11.16 |