ROS/ROCKWELL INTERFACE

Ashley Killian
Advisor: Dr. Wyatt Newman
Sponsors: Rockwell Automation
29 April 2015

Executive Summary

The goal of this project was to create a way for a ROS (Robot Operating System) system to send joint trajectories to a Rockwell Automation controller. The project was sponsored by Rockwell Automation, as part of their initiative to work more collaboratively with Case Western Reserve University. Rockwell Automation is interested in expanding its market, specifically in industrial robots. Incorporating ROS with their industrial automation equipment was of interest to Rockwell Automation because of its free, open framework, current and growing capabilities, and supportive simulation tools.

This system uses a Linux computer running ROS (using C++), Rockwell controller and two Ethernet modules, proprietary programming software package for the controller, a drive, and a motor. All equipment other than the Linux computer and Windows computer are all manufactured by Rockwell Automation. The interface relies on sockets connection (TCP/IP) to send information from a running ROS script to the controller.

Introduction

Right now industrial customers cannot integrate ROS with commercial automation equipment. ROS is used over other robotic programming frameworks because the primary goal of ROS is to support code *reuse* in robotics research and development (ROS.org). This is attractive to robotic users in industries because, since code is already available for application, it saves the customer time to start innovating their robotic systems. It also encourages collaboration, creating a file sharing system that allows multiple people to work on a project. Most engineers work in teams, so this makes ROS a positive feature. It is easy to test ROS programs, which helps going from development to real applications faster. ROS also runs on Linux Ubuntu, a free operating system. ROS ends up being cost effective and easy to start developing robotic systems.

Rockwell Automation provides industrial automation solutions, with customers all over the world. The growth of robot sales has increased by 15% from 2013 to 2014, and this trend is expected to continue. This is a market that Rockwell Automation can tap into and provide industrial robotic solutions for automation companies. Combining Rockwell Automations' top of the line products with the ever growing ROS community can give industry users more of an edge in assembling and distributing their products to the market. Combining free robotic software that supports code reuse and collaboration, and hardware from a company known for its innovative solutions can be a game changer in the industrial robotic industry.

Background

ROS is an open source software framework for robotic applications. ROS-Industrial is a program that leverages capabilities and functionality within ROS for industrial robots, creating industry-related ROS software (rosindustrial.org). It allows for industrial users to use already available ROS technology, such as tools for development, simulation, and visualization and 2-D and 3-D perception to create new applications for their industrial robots. The ROS-Industrial core also simplifies robot programming at the task level by eliminating path planning and teaching, and applying abstract programming principles to similar tasks. It is also free and standardizes interfaces. Current companies that are part of the ROS-Industrial Consortium Americas are ABB, Fraunhofer IPA, Siemens, and EWI, which are also companies that are in the industrial automation industry. The promises in ROS-Industrial makes companies like Rockwell Automation worth it to start looking into how to integrate their technology with the ROS-Industrial software framework.

Current research at Rockwell Automation for incorporating robotic programming and planning is using Simumechanics in Matlab with the ControlLogix controllers. To do this, a robotic simulation running in Simumechanics outputs movements to the controller by compiling into different bits of code and sends it to a Special Applications Module (SAM). The SAM compiles this code and sends the movements through the backplane to the controller. The limits with this approach is Matlab is expensive, the set-up for this system is complex, and there

is a learning curve with learning Matlab. Using an approach like this but using ROS, and eventually using the ROS-Industrial core, is one approached looked at for this project and should still be looked at for further research. However, because of time constraints and learning curve for this approach, using sockets was focused on for the majority of the project. In addition, ROS-Industrial has packages that use sockets and TCP/IP (Transmission Control Protocol/Internet Protocol) for sending joint trajectories to robots. Looking into the future, it seemed smart to use this same method with the project.

Project Goals/Technical Specifications

The main goals of this project can be outlined by the following list:

- Establish socket communication between ROS and the controller
- Send float values to the controller
- Use the data from ROS to move motor

Further notes regarding these goals below.

The primary goal of this project was for the Rockwell controller to read data from a running ROS node and be able to move a motor with the data ROS publishes. Due to the nature of ROS and robotic programming in general, update rates need to have low latency (delay time of 1 msec or lower) and high sampling rates (100Hz – 1kHz). Thus, it was important to come up with a method to move data between ROS and the controller with this in mind.

Functions of ROS Code:

- Generate Float Values
- Create Client and Socket for Client
- Convert Float to ASCII String
- Establish communication to the controller's server socket

Functions of Logix Code:

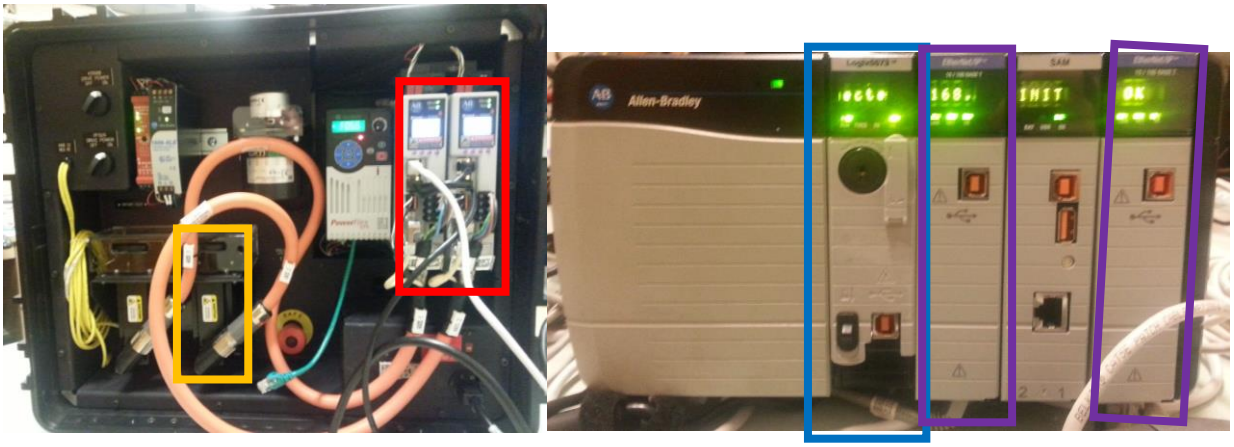
- Create Socket as Server
- Accept Data from Client (ROS node)
- Read Data
- Parse Data and Store Parsed Values as Floats
- Use Floats to Move Motor

Methodology

Equipment Used

- Ubuntu 12.04 w/ROS Hydro Computer

- Studio 5000 V24
- L73x ControlLogix Controller w/4 Slot Chassis
- 2 EN2T (Ethernet Modules)
- Kinetix 5500 Drive
- SERVO Motor



First Method: SAM and API

When looking at different methods of successful communication between ROS and the controller, cost, usability, and update rates were the most important factors in this decision. Two different methods were introduced in this project. First, using an existing API as guidance to create a similar system for ROS. Using infrastructure libraries in ROS and Logix, code running in ROS would compile in the Linux computer and converted to Binary using Toolchain. The Binary code would be sent to a Special Applications Module (SAM) and the SAM will process the Binary code and send the results to the controller. This method was at first ideal because it has already been done before for Simumechanics. The complexity of this method did not make it a suitable solution. The different processes that had to happen to make this work would most likely not be completed within the time allotted for this project. In addition, it was uncertain if this method will have the desired update rate requirements.

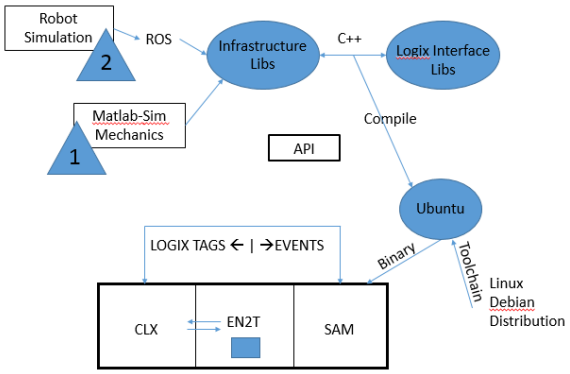


Figure 1: Original System Using API and SAM

Second Method: Sockets

The second method looked at, and used for this project, is using sockets. Sockets are used for interprocess communication. Both Rockwell Automation controllers and Linux supports this type of communication protocol. Usually, Ethernet/IP Socket Interface is used for Ethernet devices that do not comply with EtherNet/IP application protocol (EtherNet/IP is Rockwell Automation's proprietary network interface for its system), and TCP/IP was used for this type of communication. A TCP connection is a byte stream between two application entities. Hence, the type of sockets used for this method is stream sockets, treating communication as a continuous stream of characters. To establish a communication channel, a client and a server needs to create their own socket. The client instantiates the connection to the server, like a person making a phone call to another person. The client must be given the IP Address and Port Number of the server, but the server does not need to know the address of the client. Below is the list of steps involved for establishing a socket on the client side and the server side for Linux, provided by LinuxHowtos.org:

Client:

1. Create a socket with the socket() system call
2. Connect the socket to the address of the server using the connect() system call
3. Send and receive data. There are a number of ways to do this, but the simplest is to use the read() and write() system calls

Server:

1. Create a socket with the socket() system call
2. Bind the socket to an address using the bind() system call. For a server socket on the Internet, an address consists of a port number on the host machine.
3. Listen for connections with the listen() system call
4. Accept a connection with the accept() system call. This call typically blocks until a client connects with the server.
5. Send and receive data

The controller is acting as the server, and it is not a Linux machine, but the steps above is the same logical steps happening in the controller. Below is a diagram showing the Socket TCP communication protocol steps for a Logix controller acting as a server from the "EtherNet/IP Socket Interface" Application Technique Manual.

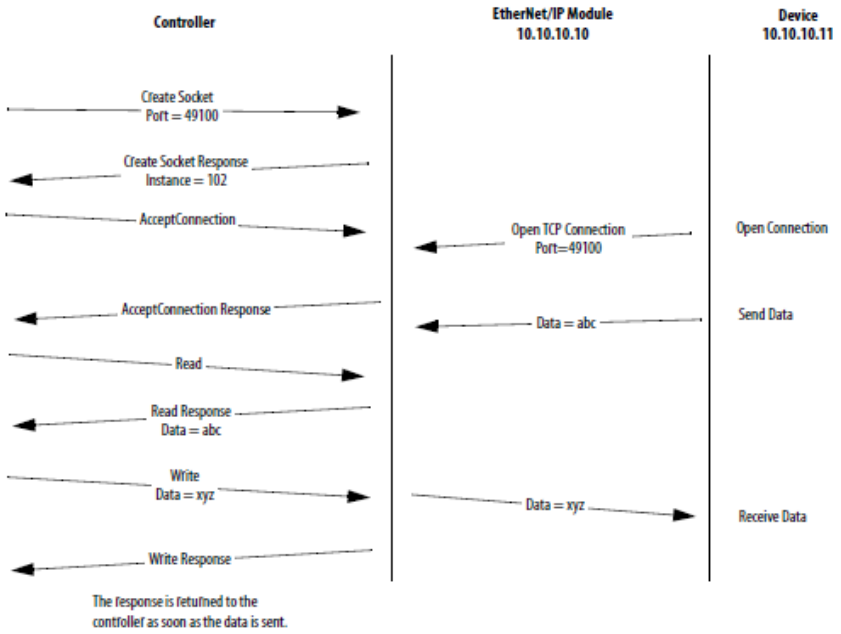


Figure 2: TCP/IP Controller Server to Client Device

Implementing Sockets: ROS to Logix

The client script has to be compliant with ROS and sockets communication requirements for Linux systems. C++ is the programming language used to write the ROS node. To send a trajectory to the controller, a sine wave (x = A*sin(w*t)) is generated, outputting float values to be sent to the controller to move the motor. Within the main function, the sine wave is created and is generated in a while loop. In addition, the client socket code is written in here as well. Below is peusdocode on how this works:

Instantiate variables
Set server port number = 10002
Create standard socket (sockfd)
    // I want a socket that uses the internet domain and is a stream socket, and the
    // computer will chose the most appropriate protocol, which will be TCP
    Error message if sockfd is invalid
Set server's IP Address
    Error if IP Address does not exist
Set server address in memory, then specify server
    Using Internet Protocol as the address type
    Set IP Address
    Set Port Number
    Test if Client can establish connection to this server
Enter While loop (while ros is okay)
    Run sine wave, publish values as floats in ROS

Set String ASCII in memory, set initial values to zero
Print float values into the string ASCII
Send the String ASCII to the server

*Credit to LinuxHowtos.org for explaining socket communication and providing a template for creating client and server interface that was used for this project

Implementing Sockets: Logix to Motor

To program a ControlLogix controller, the Rockwell controller used for this project, their proprietary software, Studio 5000, has to be used. It is in Studio 5000 that the Logix code was made. Three sets of routines were created: one for TCP/IP, parsing through the data from the client, and moving the motor. The routines for TCP/IP and motion is written in the style of Ladder Logic, a style of PLC (Programmable Logic Controller) programming. Ladder logic programs are represented by graphical diagrams resembling circuit diagrams of relay logic. The parser code is written in the style of Structured Text, which is similar to written code like C, VBA, and Python.

For the TCP/IP program, the flow of it is similar to Figure 2 presented earlier in the document. First, all socket messages that are present, which are old messages, are deleted when the program first turns on. A socket is created when the old messages are deleted and when a user enables it by toggling “Create_Socket_Req.” When a socket is created, a user can enable the controller to accept incoming connection from the client by toggling the bit “Accept_Request.” When this action completes, the data is read as a String, stored in a “Read_Response” tag, and triggers the parser code to execute.

The parser code takes in the String from the “Read_Response” tag and parses through the string and does logic if there is a decimal (46 is the decimal value of the Char “.”). All tags in the controller can be viewed from the “Controller Tags” table or “Parameters and Local Tags” table. Below is a snapshot of what this data looks like in the controller in the “Controller Tags” table:

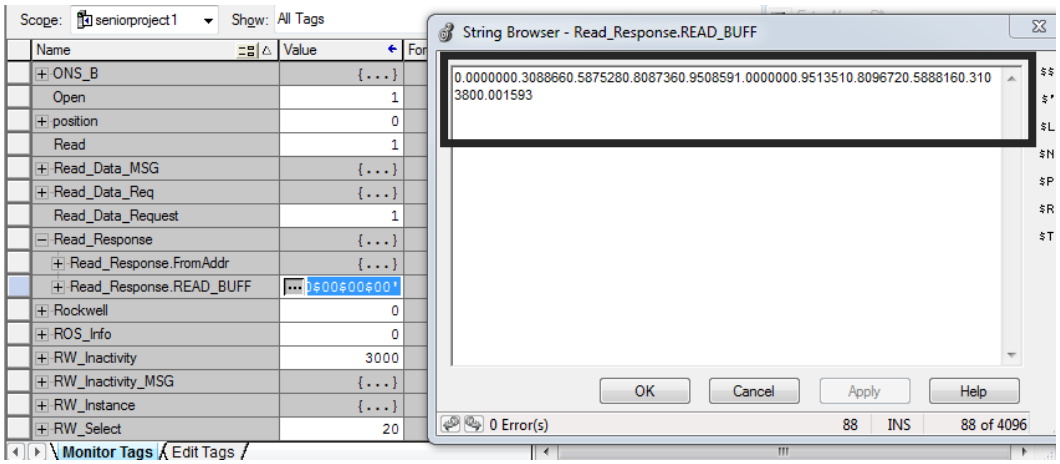


Figure 3: Read Data from Client in Controller

Within the Read_Response tag has two different entities. The “Read_Response.READ_BUFF” houses the string data, and the characters in this string are stored in an ASCII array, shown in the snapshot below:

Read_Response.READ_BUF...	{...}	{...}	ASCII	SINT[4096]
Read_Response.READ_B...	'0'		ASCII	SINT
Read_Response.READ_B...	'.'		ASCII	SINT
Read_Response.READ_B...	'0'		ASCII	SINT
Read_Response.READ_B...	'0'		ASCII	SINT
Read_Response.READ_B...	'0'		ASCII	SINT
Read_Response.READ_B...	'0'		ASCII	SINT
Read_Response.READ_B...	'0'		ASCII	SINT
Read_Response.READ_B...	'0'		ASCII	SINT
Read_Response.READ_B...	'0'		ASCII	SINT
Read_Response.READ_B...	'.'		ASCII	SINT
Read_Response.READ_B...	'3'		ASCII	SINT
Read_Response.READ_B...	'0'		ASCII	SINT
Read_Response.READ_B...	'8'		ASCII	SINT
Read_Response.READ_B...	'8'		ASCII	SINT
Read_Response.READ_B...	'6'		ASCII	SINT
Read_Response.READ_B...	'6'		ASCII	SINT
Read_Response.READ_B...	'0'		ASCII	SINT
Read_Response.READ_B...	'.'		ASCII	SINT
Read_Response.READ_B...	'5'		ASCII	SINT
Read_Response.READ_B...	'8'		ASCII	SINT

Figure 4: String of Data represented by ASCII Characters

These characters are stored in the style of ASCII. ASCII code is the numerical representation of a character, and the numerical type used is “SINT,” which is an atomic data type that stores an 8-bit signed integer value (-128 to +127). In universal language, the characters are represented by decimal values (see an ASCII conversion chart in Appendices).

For all the float values in the string array, there is always going to be a decimal point and 6 number characters following the decimal. Using this information is the basis of the logic for the parser. Overall, when it finds a decimal, there are three pointers that help keep track of how long the float value is. One keeps track of where the decimal is with respect to the string, another keeps track of the decimal with respect to the READ_BUFF.DATA[] array in Read_Response. It uses the array to loop through the string to find the decimal. The third pointer is the pointer for the string array that will store the parsed float values. The string values of the floats will then be stored as floats, converted from string, in a Cam array “Cam_1” as a SLAVE value.

A Cam Profile is a representation of non-linear motion (i.e., a motion profile) which includes a start point, end point, and all points and segments in between. A cam profile is represented by an array of cam elements. A Cam element contains a Cam left and a Cam right position, and the segment in-between. A Cam array stores these Cam positions. The Cam positions are represented by a Master, Slave, and Segment value. Master is usually time, and this is preset in the parser code because the ROS code did not send out time stamp values and the values were used as position of the motor, which is stored as a Slave value of the Cam position. The segment can be either linear or cubic, represented by 0 or 1 respectively. These

were preset as well, alternated between linear and cubic to generate a smooth Cam profile. Below is a plot of what a Cam profile will look like inside the Cam Profile Editor:

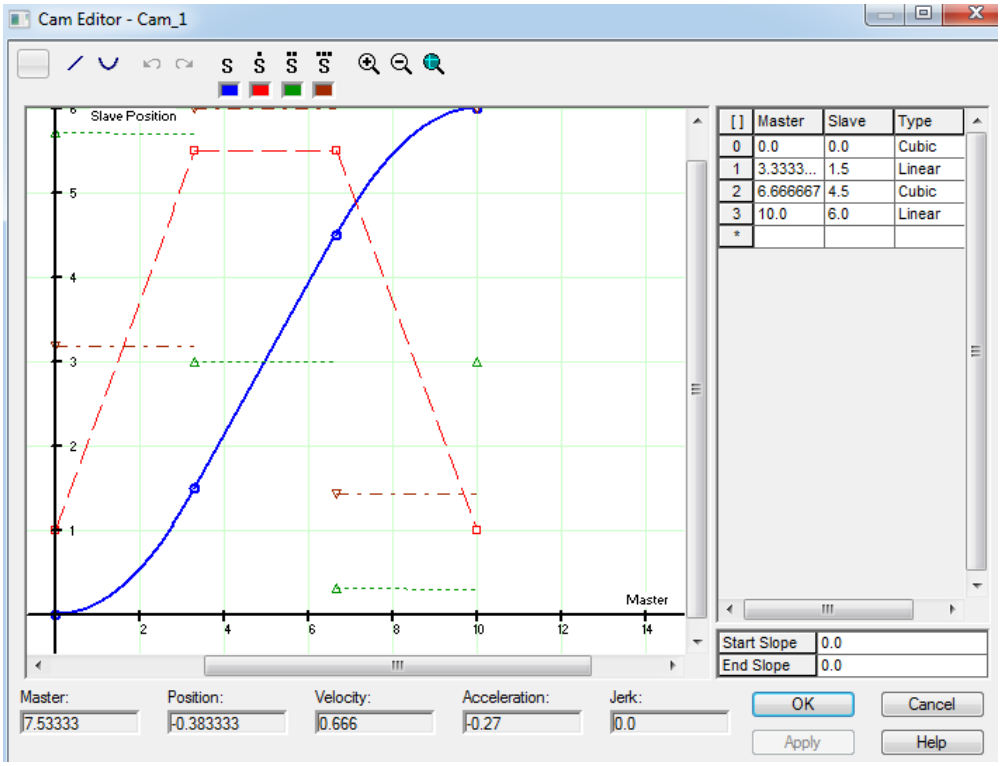


Figure 5: Cam Profile Example

Once the Cam array is populated, in the Main Routine, which houses logic for moving the motor, enable the motor, and if the motor is not moving, toggle the "use_cam" bit to initiate the MCCP (Motion Calculate Cam Profile) Axis instruction. This Axis instruction uses "Cam_1" to create a Cam Profile and store it in "cam_profile," a Cam_Profile array. To move the motor with this Cam_Profile, enable it to move, and the MATC (Motion Axis Time Cam) will use "cam_profile" to send the move to the drive, and the drive will execute the motor for the allotted time set in the parser routine.

Testing and Results

The program described in "Methodology" section was executed, and the motor moved as expected. The YouTube link (https://www.youtube.com/watch?v=sa7QtafQrTc) is a recording of the system executing, showing the motor moving. First, in Studio 5000, the controller creates a socket connection and accepts any incoming connections. In the Linux laptop, the program is initialized and float values are being populated every 1 sec. This was chosen to be slow so it is easier to see what values are being populated, making it clear if what

is being populated is the same thing the controller is reading. The accept incoming connection tag for the socket message "Conn_Accept_Msg" has to be re-toggled for the controller to capture the data from the ROS script. The data is then read for 1 second, starting at the first float value that was generated. Once this message is done, the parser script runs, saving the data from the ROS script as separate float values as SLAVE positons in the Cam_1 array. Going into the Main_Routine, the MCCP is initialize, populating a cam profile for the motor. The motor was already enabled at the beginning of the process (before the video started), and the MATC is enabled, moving the motor using the generated cam profile.

Goal Assessment

The following goals that are marked with a check were met:

- ✓ Establish socket communication between ROS and the controller
- ✓ Send float values to the controller
- Use the data from ROS to move motor

However, for the last goal (Use the data from ROS to move motor), the motor is not move ENTIRELY from ROS because the timing of the motor was preset by values within the controller. Therefore, the goal can be considered to not been complete. It is most likely, with more time, that this will definitely be completed. The ROS script would have to publish values for time in some way, and the controller will have to parse through the data for both position and time, and store the time values in the MASTER property of a Cam position.

Administrative

The management plan for completing this project was changed multiple times because the goals of the project were altered. The two most notable changes were:

- Move two motors -> move one motor
- Use the SAM and API as framework -> use Sockets

The first change, move one motor instead of two, happened because the project was not progressing fast enough. Conflicts that affected the time for the project are the following:

- Equipment arrived in the middle of February
- API software is proprietary, so legal software agreements were made and were not finalized till early March. Then, the API software was obtained on April 1st
- After learning more about the API software from using it, a judgement call was made that this approach will not allow the project to be completed by end of April. The switch to sockets was made, thinking this approach is much faster and has a smaller learning curve and understanding the API approach.
- There was a learning curve for sockets

- Limited Logix experience made progress slower for programming the controller

The above list mentions the change from using the API approach to sockets. There seemed to be more steps with using this approach, such as understanding how the API works with Simumechanics, installing various files, libraries and kernels into SAM, running the Simumechanics simulation and converting the code so it works with a newer version of Matlab (going from version 2011 to version 2014). Testing this before even implementing ROS made the API approach less likely to be completed. Sockets has a smaller learning curve. There are more resources on this, for both the controller side and the Linux side. The sockets approach was decided in the beginning of April, giving a month to implement this method. If the sockets approach was started earlier, for example in mid-February, a lot more could have been accomplished.

Project Implications

Working alongside a company slows a project down. The equipment and software needed were not acquired at the start of the project. In the future, if more equipment is needed from Rockwell Automation, expect to receive it later than anticipated. However, working with a company means working with employees who can help with the project. This assistance can help move a project move along faster if a problem occurs.

People and students who continue this project will most likely not have experience with programming in Logix. One way a person can learn is to use the Help contents information. Another is to attend training classes at Rockwell Automation. The training classes provide instruction manuals, so one can obtain them and learn on their own, asking assistance when needed.

Current Status/Future Work

Both the SAM and sockets methods were proposed by Rockwell engineers. After learning more about both, the sockets method was deemed more feasible and probably better than using the SAM and API. There is a bigger learning curve for the API and SAM solution than using sockets. First, one would have to learn how it works by setting up the API system that was created for Simumechanics. The necessary infrastructure libraries will have to be installed properly. More work may have to be put into doing this if the version of Matlab used is newer than the one the API works with. Necessary packages needs to be downloaded into the SAM module into separate directories. Once these steps are passed, then Simumechanics can run and successfully compile code into SAM, and the controller will execute. However, the socket interface, via messaging, is not well suited for real-time control as communication with this method is not scheduled or deterministic. This may not be as big of an issue since sockets is used for ROS-Industrial to send joint trajectories to controllers (http://wiki.ros.org/Industrial/Tutorials/), (http://wiki.ros.org/simple_message). If the ROS-Industrial does us TCP and sockets to send joint trajectories to controllers, then it is worth looking in to using this approach and using the ROS-Industrial stack for future work. However,

since sockets is a concern for the Rockwell controllers if implementing real-time control, it is best that another method is looked at for controlling the motors using ROS in collaboration with Rockwell engineers. Or, the SAM and API method is looked into more deeply, since it has been used for Simumechanics to control simulated robots.

Appendices

Important Information from the "EtherNet/IP Socket Interface" Application Technique Manual

These socket services are available.

Socket Service	Socket Instance	Page
Socket Create	Server or client	26
OpenConnection	Client	28
AcceptConnection	• If you issue an AcceptConnection service, the instance is a listen type. • If the AcceptConnection service returns an instance as a result of an incoming connection request, the socket instance is a server type.	30
ReadSocket	Server or client	32
WriteSocket	Server or client	34
DeleteSocket	Server or client	37
DeleteAllSockets	Server or client	38
ClearLog	Server or client	39
JoinMulticastAddress	Server or client	40
DropMulticastAddress	Server or client	41

Table 3 - Maximum Packet Sizes

Service	Unconnected Size	Standard Connection Size	Large Connection Size
ReadSocket	484 bytes	484 bytes	3984 bytes
WriteSocket	462 bytes	472 bytes	3972 bytes

If an application sends variable size messages, a common strategy is to first send a fixed-size header containing the size of the message followed by the message. The receiving device can first issue a ReadSocket service of the fixed size header to determine the remaining size, and then issue a subsequent ReadSocket service to receive the remaining data.

Unlike I/O connected via an EtherNet/IP module, communication via messaging to socket instances can continue when a module is inhibited. If you want to stop socket communication when a module is inhibited, your application code must detect the status of the module and take the appropriate action.

As noted previously, the socket interface enables a Logix5000 controller to communicate via an EtherNet/IP module with Ethernet devices, such as bar code scanners, RFID readers, or other standard Ethernet devices, that do not support the EtherNet/IP application protocol. The socket interface, via messaging, is not well suited for real-time control as communication with this method is not scheduled or deterministic.

ASCII Table Conversion

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

ROS Node Code

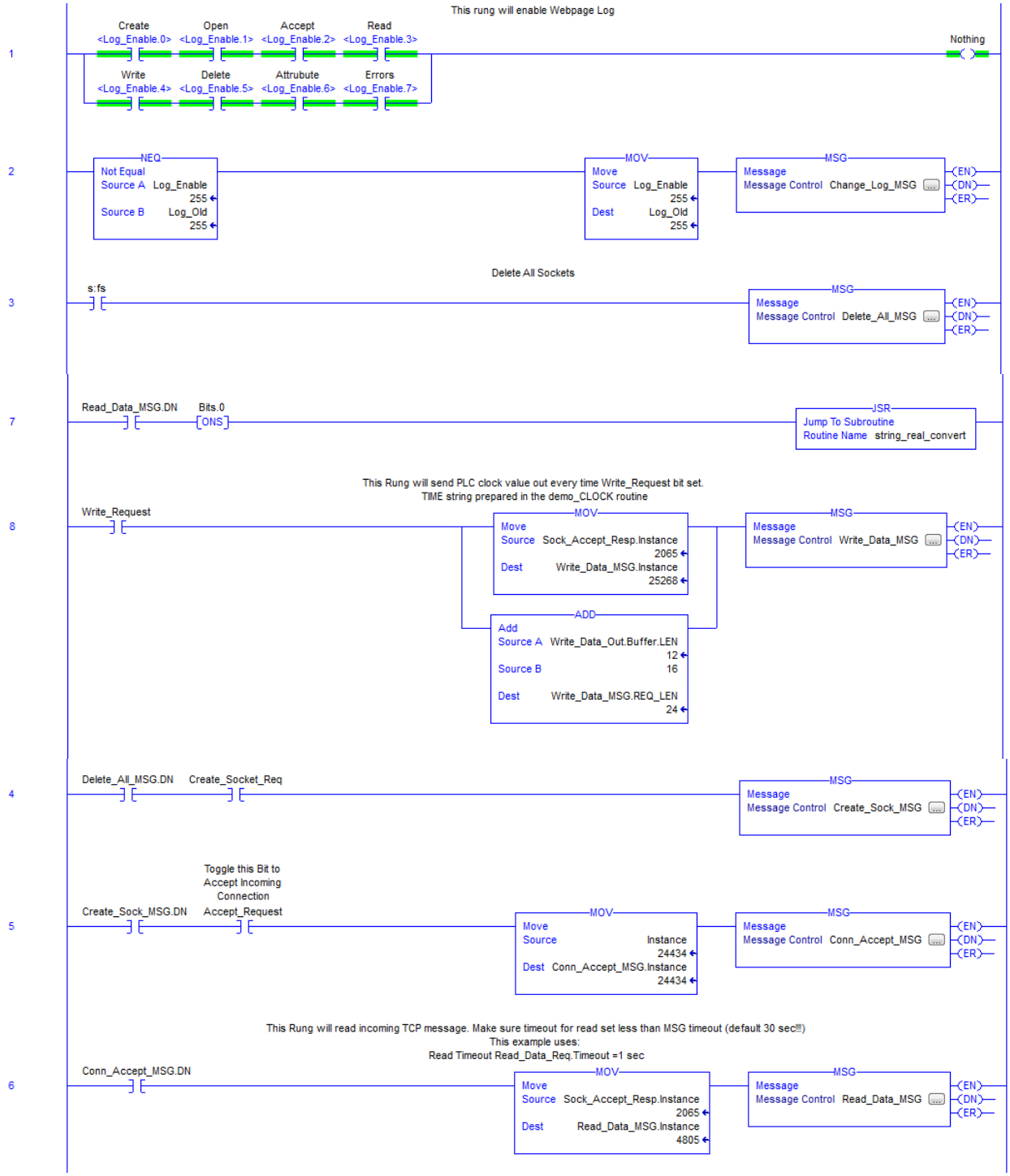
```
#define _USE_MATH_DEFINES

#include <ros/ros.h>
#include <std_msgs/Float64.h>
#include <std_msgs/Float32.h>
#include <math.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define MAX_FLOAT_LEN 32
void error(const char *msg)
{
    perror(msg);
    exit(0);
}

// make sine wave a method if desired
/*float motorMove(long double omega) {
    double a = 1.0;
    double t = 0.0;
    double dt = 1.0;
    double x_des = a*sin(omega * t);
    t+=dt;
    return x_des;
}*/

int main(int argc, char **argv) {

    // name of this node will be "vel_scheduler"
    ros::init(argc, argv, "vel_scheduler");

    // get a ros nodehandle; standard
    ros::NodeHandle nh;

```

```
40      // create a publisher object that can talk to ROS and issue twist messages on named topic;
41      // note: this is customized for stdr robot;
42      ros::Publisher pub  = nh.advertise<std_msgs::Float32>("joint_angle_command", 1);
43
44      // x = a*sin(omega*t)
45      long double omega = 3.14;
46      double a = 1.0;
47      double t = 0.0;
48
49      // incrementing t
50      double dt = 0.1;
51
52      //allow callbacks to populate fresh data
53      ros::spinOnce();
54
55
56      int sockfd, portno, n;
57
58      // name of server is it's IP Address
59      char *name = "192.168.20.5";
60      struct sockaddr_in serv_addr;  // structure to contain an IP Address
61      struct hostent *server;
62
63      // creating string ASCII
64      char floatArray[MAX_FLOAT_LEN];
65
66      // Create socket for client
67      ROS_INFO("about to create socket");
68
69      portno = 10002;  // port number of server
70
71      // error displayed if this fails
72      sockfd = socket(AF_INET, SOCK_STREAM, 0);  // standard, define socket file descriptor,
73                                                 //always use 0 for last arugment
74      if (sockfd < 0)  // fails if socket call returns -1
75          error("ERROR opening socket");
76      server = gethostbyname(name);
```

```
77      if (server == NULL) {
78          fprintf(stderr,"ERROR, no such host\n");
79          exit(0);
80      }
81
82      // setting fields in serv_addr (address of server)
83      memset((char *) &serv_addr,0, sizeof(serv_addr)); // sets bytes in memory to zero,
84                                                        // initializes serv_addr to zeros
85      serv_addr.sin_family = AF_INET; // always set to AF_INET
86
87      // set IP address to serv_addr, use bcopy because server->h_addr is a character string
88      bcopy((char *)server->h_addr,  // h_addr = the first address in the array of network addresses
89           (char *)&serv_addr.sin_addr.s_addr,
90           server->h_length);
91      serv_addr.sin_port = htons(portno); // set port number of server, converted to network byte order
92
93      // client establish connection to server (socket descriptor, address, size of the address)
94      if (connect(sockfd,(struct sockaddr *) &serv_addr,sizeof(serv_addr)) < 0)
95          error("ERROR connecting");
96
97      // do work here in infinite loop (desired for this example),
98      // but terminate if detect ROS has faulted (or ctl-C)
99      while (ros::ok())
100     {
101         // publishing x = a*sin(omega * t)
102         std_msgs::Float32 ros_float;
103         double x_des = a*sin(omega * t);
104         t+=dt;
105         ros_float.data = x_des;
106         pub.publish(ros_float);
107
108         // read and write
109         ROS_INFO("about to read and write");
110
111         // setting bytes in memory for string ASCII to 0
112         memset(floatArray, 0, sizeof(floatArray));
113         snprintf(floatArray,sizeof(floatArray),"%f",ros_float.data);
113           snprintf(floatArray,sizeof(floatArray),"%f",ros_float.data);
114           ROS_INFO("floatArray %s", floatArray);
115
116           // send string ASCII to server, n = 1 if successful
117           n = send(sockfd,floatArray, strlen(floatArray), 0);
118           if (n < 0)
119               error("ERROR writing to socket");
120
121           ROS_INFO("move sent to controller");
122
123           // here, enable client to receive data from server
124           /*n = read(sockfd,floatArray,255);
125           if (n < 0)
126               error("ERROR reading from socket");
127           printf("%s\n",floatArray);*/
128
129           ros::Duration(0.1).sleep();
130
131     }
132     // done with reading and writing to controller
133     ROS_INFO("completed move distance");
134
135     // Close client socket
136     close(sockfd);
137
138     return 0;
139 }
140
141
```

Logix Code

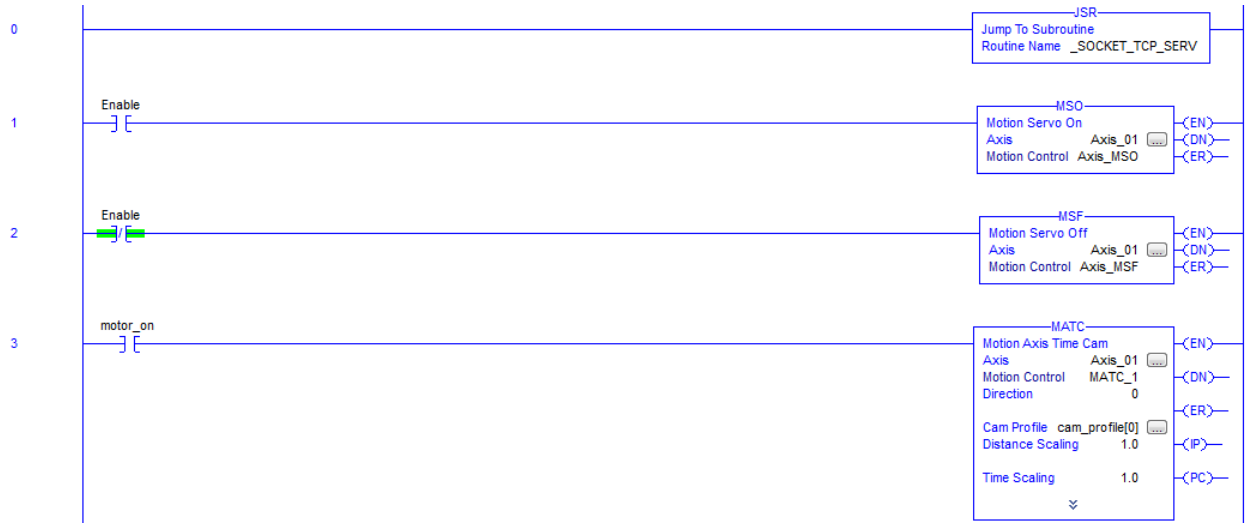
TCP/IP Sockets Code

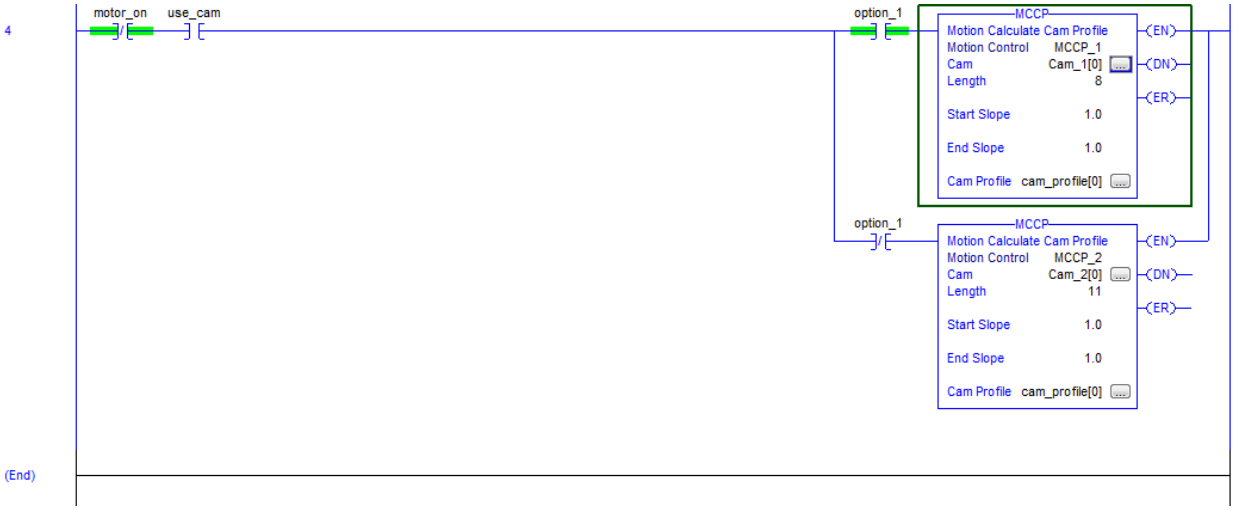


Parser Code

```
// i is pointer for READ_BUFF.DATA[] (index 0), j pointer for Read_Response String (Read_Response.READ_BUFF)(index 1),
// k is pointer for SourceA[] and Cam_1[]
i := 0; j := 1; k := 0;

// Parameters for MID, parser instruction in LOGIX
QTY := 0; STRT := j;

// INCREMENT is how many numbers are before the decimal for a value (12.5647, marks the 12)
// BEGINING is where we started in the string, wrt READ_BUFF.DATA[]
INCREMENT := 0; BEGINNING := 0;

Cam_1[0].Master := 0.0; Cam_1[1].Master := 3.3333333; Cam_1[2].Master := 6.666667; Cam_1[3].Master := 10.0;
Cam_1[4].Master := 13.333333; Cam_1[5].Master := 16.66667; Cam_1[6].Master := 20.0; Cam_1[7].Master := 23.33333333;

// parse through read data
WHILE k < 8 DO
    // 46 is "."
    IF Read_Response.READ_BUFF.DATA[i] = 46 THEN

        // ermehgerd found a decimal

        j := i + 1;  // mark decimal in string, Read_Response.READ_BUFF
        INCREMENT := i - BEGINNING;   // mark numbers in beginning of decimal point of number
        STRT := j - INCREMENT;  // start at beginning of number
        QTY := 7 + INCREMENT;  // amount of characters to copy over, at least 8
        MID(Read_Response.READ_BUFF, QTY, STRT, SourceA[k]);
        STOR(SourceA[k], Cam_1[k].Slave);  // convert from string to real
        i := i + 8;  // go to next possible place for a decimal, when there is only one number in front of it
        BEGINNING := i - 1;  // now begining is at possible number left of a potential decimal
        k := k + 1;  // go to next element in SourceA and Cam_1
    ELSE
        // still haven't found decimal, so increment once
        i := i + 1;
    END_IF;

END_WHILE;
```

Motion Code (Main Routine)





- Controller seniorproject1
  - Controller Tags
  - Controller Fault Handler
  - Power-Up Handler
- Tasks
  - MainTask
    - MainProgram
      - Parameters and Local Tags
      - MainRoutine
      - string_real_convert
      - _SOCKET_TCP_SERV
    - Unscheduled Programs / Phases
- Motion Groups
  - Demo_group
    - Axis_01
  - Ungrouped Axes
- Add-On Instructions
- Data Types
  - User-Defined
  - Strings
  - Add-On-Defined
  - Predefined
  - Module-Defined
- Trends
- Logical Model
- I/O Configuration
  - 1756 Backplane, 1756-A4