Interfacing VRPN with Luminary Microcontroller over Bluetooth

Philip Weiss and Dorsa Sadigh
July 27, 2010

1. Introduction

Prerequisites: This document assumes you have read the EE149 Tracking Tools instructions and have calibrated the cameras and enabled VRPN streaming.

Naturalpoint's OptiTrack streams real-time tracker position data over the VRPN (Virtual Reality Peripheral Network) protocol, among others. We have chosen to use the VRPN protocol. In our system, there are 3 programs that are involved. First there is the C program which reads the incoming VRPN data from Tracking Tools over a network socket, puts it in a data packet, and sends it over a socket to the Python Bluetooth program. The Python program accepts incoming connections from the BlueSMiRF Bluetooth devices and sequentially broadcasts the packet with the VRPN data to all properly configured BlueSMiRFs in range. Lastly, the embedded program, which runs on the Luminary board, receives the data packet and decodes it back into position data.

2. General Usage

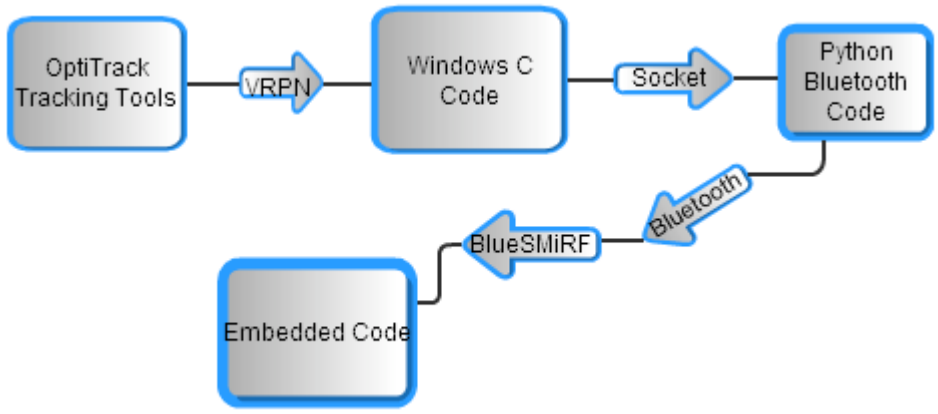

Figure 1 - Block Diagram of Basic System

1. Setup of Rigid Bodies and Tracking Tools

Before running any of the programs, make sure that:

- You have started Tracking Tools and loaded a camera calibration
- You have defined at least one rigid body tracker whose name is a number in the range from 0 to NUM_TRACKERS-1 (defined in main.cpp of the Windows C++ code)
- This rigid body is in view of the cameras and being actively tracked
- VRPN streaming is enabled on port 3883

2. BlueSMiRF Setup

Before starting the programs, you first must verify that the BlueSMiRF's settings are correct. To do this, use USB-TTL RS232 adapter to interface the BlueSMiRF to a PC. Using a terminal emulator like HyperTerminal, connect to the COM port assigned to the adapter at 115200 bps (if the BlueSMiRF isn't configured right you may need to connect at 57600 bps or another bitrate).

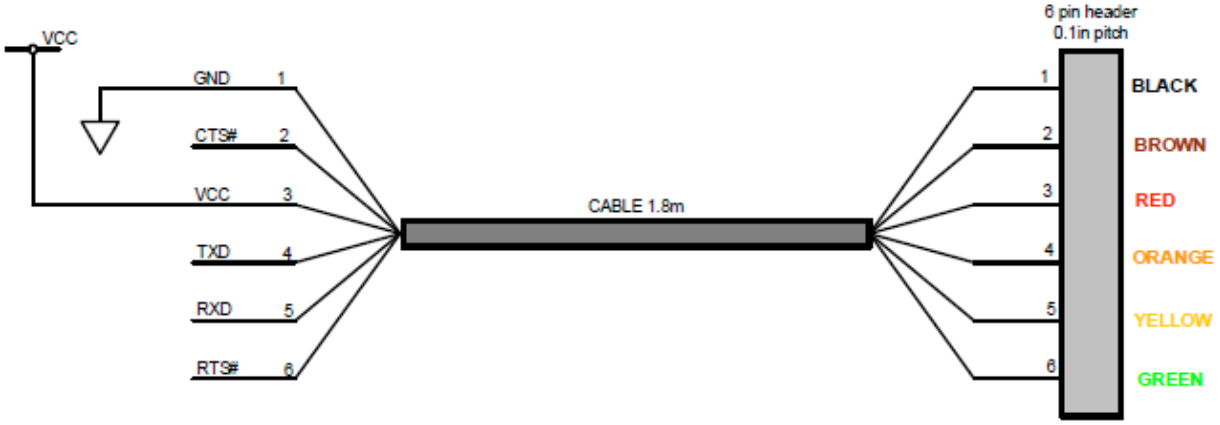

Figure 2 – Connection diagram for USB-TTL level RS232 adapter.[1]

To enter programming mode, type "$$$" until you see "CMD", or until the red light on the BlueSMiRF begins to flash rapidly. To see the settings, type "d" and press enter. There are various commands to change these settings. The important ones are:

Command	Purpose	Followed by 'Enter'?
$$$	Enters command mode	No
D	Displays basic settings	Yes
SU,11	Sets baud rate to 115.2 kbps	Yes
SM,3	Set BlueSMiRF to autoconnect mode	Yes
SR,001583be9cad	Set Bluetooth address to connect to as 001583be9cad	Yes
---	Exits command mode	Yes

Once these settings are correct, connect the BlueSMiRF back to the Luminary board.

The configuration above tells the BlueSMiRF to try to connect to the stored Bluetooth address. The Bluetooth address is the address of the Bluetooth dongle in the desktop PC running the Python Bluetooth script. This means it will be connected to the Python script as soon as the script starts, if the Bluetooth devices are in range.

3. Windows Programs

1. Compile and Launch C++ code. To compile, you must have the VRPN and quat headers present and libraries linked. Once launched, the program will connect to the VRPN server (Tracking Tools) and open a listening socket on port 10625.
2. Start Python script. Make sure a version of Python before 3.0, and the PyBluez library are installed. This will connect to the listening socket and then continuously accept BlueSMiRFs connecting and VRPN data packets, which will be forwarded to the BlueSMiRFs.

4. Embedded Code

Compile and flash the μVision 4 project. Press the reset button on the Luminary. Streaming data for properly named trackers will be displayed.

3. Implementation Details

1. Windows C++ Code

The purpose of this code is to connect to the Tracking Tools VRPN server, handle incoming tracker data, and send a data packet over a socket to a client (the Python Bluetooth broadcast code).

1. Constants and global variables

Name	Value (or Initial Value)	Description
CONSTANTS		
M_PI	3.14159265358979323846	Value of π
UART_TX_BUFFER_SIZE	0x40 = 64	Size of the transmit xqueue's buffer
UART_VALUES_PER_PACKET	8 (subject to change)	Number of int32 values per packet
PORT	10625	Port number for the network socket to listen on
NUM_TRACKERS	2	Number of OptiTrack trackers to send data for
GLOBAL VARIABLES		
static xqueue_t uartTx0	undefined	The main xqueue to hold packets to transmit
static byte uartTx0Buffer	undefined	The array containing the entire packet to send
SOCKET sock, client	undefined	The network sockets for our listening server and connected client

2. Packet format

0xFF	0xFF	Length	Name	X	Y	Z	Qx	Qy	Qz	Qw	checksum
Header			Data (Each is a 32-bit int)								Sum of every byte

Description of Packet Elements:

Name/Value	Description/Purpose	Number of Bytes
Header (0xFF 0xFF)	Used to detect the beginning of the packet	2
Length (equal to UART_VALUES_PER_PACKET)	Number of data elements in the packet	1
Name	The id of the tracker	4
X	X position of the tracker	4
Y	Y position of the tracker	4
Z	Z position of the tracker	4
Qx	X component of the tracker's quaternion rotation	4
Qy	Y component of the tracker's quaternion rotation	4
Qz	Z component of the tracker's quaternion rotation	4
Qw	W component of the tracker's quaternion rotation	4
Checksum	Sum of each byte in the packet, used to determine if packet has been corrupted	1

3. Important Functions (init, repeated functions)

Name	Purpose	Brief Outline	Return Values
int main()	main	• Start VRPN connection • Initialize **vrpn_Tracker_Remote**'s • Associate Trackers with the **handle_pos** function • Receive connection on network socket • Poll for new tracker data	• 0 on termination
int openSocket()	Listen for and accept incoming client socket's connection	• Initialize socket and bind to port **PORT** • Block while waiting for a connection • Start the connection with the client	• 0 if the function fails • 1 once a connection has been

			established
void closeSockets()	Close network sockets and cleanup	• Close **client** and **sock** sockets • Cleanup Windows sockets	none
void sendPacket(…)	Send a data packet to the client	• Assemble packet data • Put data into a packet • Put packet into an xqueue • Send data over the socket	none
void VRPN_CALLBACK handle_pos(…)	This function is called when new data for a tracker is received	• Convert quaternion values to Euler angles • Send a packet with the tracker name and data	none

4. Guide to VRPN and Socket Functions/Data Structures/Classes Used

VRPN[2, 3]:

Classes:

vrpn_Connection: A class describing a connection to a VRPN server. See "vrpn\vrpn_Connection.h" for member functions and variables.

vrpn_Tracker_Remote: A class describing one rigid body being tracked by the server. See "vrpn\vrpn_Tracker.h" for member functions and variables.

Functions:

vrpn_get_connection_by_name(connectionName): Takes parameter **connectionName** as a string containing "host:port" of the server. Returns a **vrpn_Connection** class containing the connection information. See "vrpn\vrpn_Connection.h" for more information.

new vrpn_Tracker_Remote(namestr, connection): Constructor for **vrpn_Tracker_Remote**. **namestr** is a string containing the name of the tracker. **connection** is the **vrpn_Connection** class with a connection information to the VRPN server. See "vrpn\vrpn_Tracker_Remote.h" for more information.

register_change_handler(name, handle_pos): Member function of **vrpn_Tracker_Remote**. Registers the callback function **handle_pos** to be called when new data comes in for the tracker registered in the constructor. See below for information on the callback function. **name** is a pointer to a variable which will be passed into the callback function. **name** can be NULL.

void VRPN_CALLBACK handle_pos (void * userdata, const vrpn_TRACKERCB t): An example callback function. Takes in a **void** pointer **userdata** with the data given in the **register_change_handler** function to identify which tracker triggered the callback function (or

any other use). **t** is a data structure containing information on the tracker's position. See "vrpn\vrpn_Tracker_Remote.h" for more information on the **vrpn_TRACKERCB** struct.

Sockets[4]:

Commented in the code, also refer to Beej's Guide[4], especially sections 5 and 9.

2. Python Bluetooth Code

1. Global variables

Name	Initial Value	Description
serverIP	"localhost"	The name or IP address of the computer running the Windows C++ code
serverPort	10625	The port to connect to on the server
clients	[ ]	The list of **btClient** objects
latestPacket	'None'	The value of the most recent packet received

2. btClient class

Each instance of the btClient class represents an active connection with a Bluetooth device (BlueSMiRF).

Member Variables	
Name	**Value**
btSock	The socket object for the corresponding Bluetooth connection
btAddr	The Bluetooth address of this client
Static Methods	
Name	**Purpose**
clientsList()	Return a list of the Bluetooth sockets among all **btClient** instances
remClient(clSock)	Remove this Bluetooth socket from **clients**

3. Important Functions

Name	Purpose	Brief Outline	Return Values
Main Script	Most of the action is here	• Open a TCP socket and connect to the server serverIP:serverPort • Open a Bluetooth socket and advertise virtual serial port service • Wait for an event	N/A

		on the TCP or Bluetooth socket • If Bluetooth event, handle new client or client disconnect • If TCP event, handle incoming data or server disconnect	
btBroadcast(message)	Send the **message** to all connected Bluetooth devices	• Loop through the list of **btSocket**s, call send method on associated Bluetooth socket to send **message**	none
shutdown(code)	Cleanup and exit with **code**	• Call countdown function to wait 5 seconds before exiting • Loop trough list f Bluetooth sockets and close them • Close Bluetooth server and TCP client sockets	none
btClient.clientsList()	See above	• Loop through **clients** list and return the associated **btSock** Bluetooth socket	A list of all the sockets associated with all **btClient**s
btClient.remClient(clSock)	See above	• Loop through **clients** list until the **btSocket** with the matching **btSock** is found	none

4. Guide to PyBluez and Socket Functions Used

For full documentation, see the PyBluez API Documentation[5] and Python Sockets Standard Library[6].

PyBluez:

BluetoothSocket Class

The **BluetoothSocket** class is used to describe a Bluetooth socket on the local machine.

BluetoothSocket Class Important Member Functions

BluetoothSocket(proto): The constructor for **BluetoothSocket**. **proto** is the Bluetooth protocol to use (only RFCOMM is available in the Windows version of PyBluez). Defined in the PyBluez API Documentation[5] as __init__.

accept(): Blocks until an incoming connection has been received on this socket. Returns a tuple (**BluetoothSocket**, **addrport**) where the **BluetoothSocket** is the socket associated with the connection to the new client and **addrport** is the RFCOMM channel number (not used in this application). **bind** and then **listen** must be called first.

bind(addrport): Associates the **BluetoothSocket** with a specific RFCOMM channel to listen or transmit on. **addrport** is a tuple (**host, channel)** where **host** is the Bluetooth adapter and **channel** is the RFCOMM channel. **host = "”** for default adapter**.**

close(): Closes any connection associated with this **BluetoothSocket**.

listen(backlog): Listens for incoming Bluetooth connections. **backlog** is the number of connections it will allow to wait before rejecting them.

recv(buffersize): Receives up to **buffersize** bytes from the **BluetoothSocket**.

send(data): Sends a string **data** over the socket. Returns the number of bytes sent.

Other PyBluez Functions

advertise_service(sock, name, service_id, service_classes, profiles): Advertises a Bluetooth service as being available to nearby Bluetooth devices. **sock** is a bound, listening **BluetoothSocket**, **name** is a string containing the service name, **service_id** is a string containing the hexadecimal UUID, **service_classes** is an array of service types, and **profiles** is an array of Bluetooth profiles supported. See the PyBluez Documentation[5] and "An Introduction to Bluetooth Programming"[7] for more information.

lookup_name(address, timeout): Blocks while looking up the name associated with the Bluetooth address contained as a string in **address**. Gives up after **timeout** seconds. Not used in our code because it stalls the program. It's possible that using threads could eliminate the blocking issue.

stop_advertising(sock): Stops advertising services registered with this BluetoothSocket. Usually this is called before closing the socket

Sockets:

Refer to Python Sockets Standard Library[6]. This Python Sockets Tutorial[8] is also a good introductory resource.

The select function[9] is used in the Python code. The documentation[10] for it gives the complete information. Select is used when you want to handle input on multiple I/O objects (sockets in this case). It takes a list of sockets that you're waiting for, and returns a list of which sockets are ready for reading or writing (or give an exception). At that point you can call the right function (i.e. **recv(…)** or **accept()**) on the socket.

3. Embedded Code

1. Constants and global variables

Name	Value (or Initial Value)	Description
CONSTANTS		
NUM_TRACKERS	2	Number of objects to track
NAME_OFFSET	0	Number of first tracker to track – will track rigid bodies with names in the set [NAME_OFFSET, NAME_OFFSET + NUM_TRACKERS)
DISP_RATE	0x10000	Frequency in μs at which the display is updated
UART_RCV_BUFFER_SIZE	0x40 = 64	Size of the receive xqueue's buffer
UART_VALUES_PER_PACKET	8 (subject to change)	Number of int32 values per packet
GLOBAL VARIABLES		
Position trackers	none	Array containing rigid body tracker position values
int trackers_index	0	Index of next unused element in the **trackers**
static xqueue_t uartRx0	none	The main xqueue to hold received packets
static byte uartRx0Buffer		The array containing the entire packet received

2. Position Struct

Member Variables:

Variable	Purpose
int32 name	The integer representing the name of the tracker this struct corresponds to.
int32 x	X position of the tracker

int32 y	Y position of the tracker
int32 z	Z position of the tracker
int32 qx	X component of the tracker's quaternion rotation
int32 qy	Y component of the tracker's quaternion rotation
int32 qz	Z component of the tracker's quaternion rotation
int32 qw	W component of the tracker's quaternion rotation
int32 yaw	Yaw angle of the tracker
int32 pitch	Pitch angle of the tracker
int32 roll	Roll angle of the tracker

Associated Functions:

Name	Purpose	Brief Outline	Return Values
int Position_add(int32 name)	Add a new position struct to the **trackers** array.	• Check if our array still has space. • Add tracker with name **name**.	• 0 on success • -1 on fail
int Position_getIndex(int32 name)	Get the index of **Position** with name **name** in the **trackers** array.	• Iterate through **trackers** array	• -1 if the function fails • Index of the **Position** we're looking for
void Position_update(…)	Update the values of a **Position** in the **trackers** array.	• Call **Position_getIndex** to find array index. • Fill in **Position** struct with new values	none

3. Important Functions

Name	Purpose	Brief Outline	Return Values
int main()	Main function	• Initialize variables and call **init()**. • Infinite loop for displaying data • Cycle between which tracker data to show every 5 seconds.	none
void init()	Initializes Luminary board, UART, and display.	• See function	none
void UART0Handler()	Interrupt service routine (ISR) for incoming data on	• Read all bytes stored in the receirve buffer. • Call **packetExtractFromStream**	none

	UART port 0 (BlueSMiRF)	(defined in "149_packet.h" • Check if this is a packet we're interested in. • Fill in a **Position** struct with the packet data • If this is a new tracker, add it to the **trackers** array**,** in either case update the element with the new values.	

4. Guide to Serial Communication, Packet, and Xqueue Functions/Structs

Luminary Serial Communication:

Functions associated with UART can be found on pg. 243 of the Stellaris Peripheral Driver Library User's Guide[11].

Packet Functions:

Packet functions are outlined in "149_packet.h".

Xqueue:

Xqueue data structure and functions are outlined in "149_xqueue.h"

4. References

1. Future Technology Devices TTL-232R Datasheet, http://www.ftdichip.com/Documents/DataSheets/Modules/DS_TTL-232R_CABLES_V201.pdf
2. VRPN Tracker Remote Class Documentation, http://www.cs.unc.edu/Research/vrpn/vrpn_Tracker_Remote.html
3. VRPN Connection Class Documentation, http://www.cs.unc.edu/Research/vrpn/Connection.html
4. Beej's Guide to Network Programming, http://beej.us/guide/bgnet/output/html/multipage/index.html
5. PyBluez 0.7 API Documentation, http://pybluez.googlecode.com/svn/www/docs-0.7/index.html
6. Python Sockets Standard Library Documentation, http://docs.python.org/library/socket.html
7. An Introduction to Bluetooth Programming, http://people.csail.mit.edu/albert/bluez-intro/c212.html
8. Sockets in Python, http://www.devshed.com/index2.php?option=content&do_pdf=1&id=593
9. select.select, http://docs.python.org/library/select.html#select.select
10. select — Waiting for I/O completion, http://docs.python.org/library/select.html
11. Stellaris Peripheral Driver Library User's Guide, http://chess.eecs.berkeley.edu/eecs149/sp10/docs/SW-DRL-UG-3618.pdf