Michigan State University
Computer Science
Collaborative Design

Union Pacific Railroads Wireless Network Research Project

Tyler Tom – Project Manager//Systems Developer
Bill Vassas – Webmaster//Systems Developer
Chien Se – Systems Developer
David Wilson – Systems Developer

Index

2…………………………Index
3…………………………Statement of Problem
4…………………………Team Member Introduction
5…………………………Proposed Solution
7.......................................Protocol Overview
8.......................................Protocol In Depth
8………………………………Data Transmission
9………………………………Wireless Device Addressing
10.....................................Implementation Overview
11.....................................Implementation In Depth
11……………………….........Configuration Files
14……………………………..Train Device to Wireless Interface Device
15……………………………..Wireless Interface Device to Train Device
16……………………………..Wireless Interface Device to Wireless Interface Device
18……………………………..Hook into WID Internals (Tx)
19……………………………..Hook out of WID Internals (Rx)
20……………………….........Node Management
21……………………… RF Modems
23.....................................Hardware Overview
24………………………..Conclusion and Results
24……………………………..Protocol Results
25……………………………..RF Modem choice
26……………………………..RF Transmission Issues
27………………………..Program Compilation and Running
28………………………..Appendix A: High Level Transmit and Receive Design
29………………………..Appendix B: Transmit Thread Data Flow Design
30………………………..Appendix C: Receive Thread Data Flow Design
31………………………..Appendix D: Town Crier Wrapper Protocol API, Packet Format

Statement of Problem

The Union Pacific Wireless Network Project is designed to provide Union Pacific Rail Roads a viable wireless alternative to the current wired data network that exists on board the locomotives of their fleet. As networks need to be changed, individual systems upgraded and new systems added the locomotives must be pulled out of service to have the maintenance performed. This down time costs money in manpower and service time lost by the locomotive. By creating a wireless alternative to the current wired network we aim to provide a solution capable of reducing the down time of the locomotive, the number of man hours required to perform systems upgrades and additions while maintaining, if not improving the security and reliability of the network.

Team Introduction

Tyler Tom (tomtyler@msu.eudu)	– Project Manager//Systems Development
Bill Vassas (vassaswi@msu.edu)	– Webmaster//Systems Development
Chien Se (sechien@msu.edu)	– Systems Development
David Wilson (wilso417@msu.edu)	– Systems Development

Solution Proposal

Creating a wireless solution that the current wired network can be moved to involves many challenges. First off is finding acceptable hardware, since the network is on-board a locomotive that already has many wireless RF devices communicating on 160 MHz, 220MHz, 450MHz, 896MHz, 924MHz, 2.4GHz, 5.6GHz, and 6.5GHz as well as a high noise floor from various other electro magnetic fields emanating from various device's requires us to look at many different technologies, this is discussed later on in the papers discussion on wireless technologies and vendors.

Second, the network topology as it is laid out in its wired format requires that multiple connections be made from each device to multiple other devices. This leaves us with two possible solutions. One, we create wireless links for every existing wired connection or two, we create a single wireless interface for each device that said device will be known as on the wired network. We propose that the latter of the two solutions is the more proper way to form a wireless network.

Pursuing option number two for the topology of the wireless network, we are proposing the creation of a wireless interface device that has multiple wired data connections to it and will be invisible to the device connected to it. These connections will allow the interface to be plugged directly to the devices as they are now without having to alter their current software configuration. The wireless interface will be able to address each wired connection to it directly via an internal address table; much like a network address table (NAT) found in common consumer grade routers and switches. Each device will be aware of other devices NAT via a protocol that is referred to as the Town Crier Protocol.

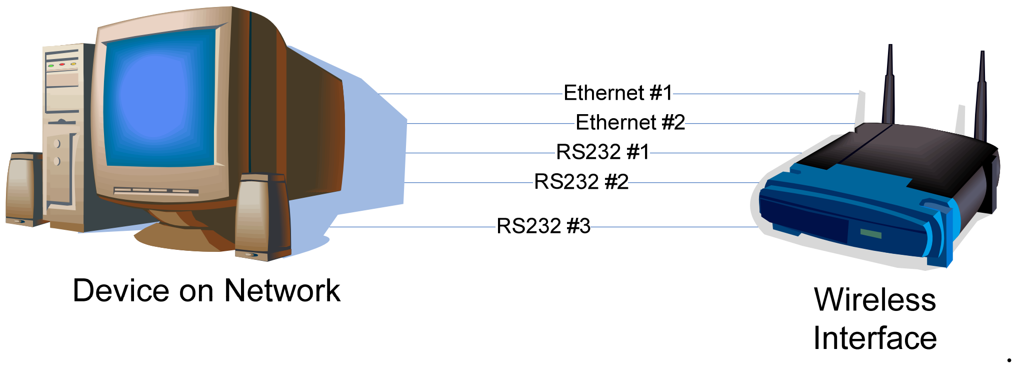


Figure 1. Device on network attached to its wireless interface

Solution Proposal

As we previously stated each device will be aware of other devices NAT in a global NAT table; this information will be stored on the wireless interface in a routing table (table 1), also as we've previously stated how this data is synced is yet TBD. In order to carry out the NAT scheme we must be able to uniquely identify not only the radio to which the device is connected, but the connection on the device that we are trying to communicate to. This will be done by chunking the data portion of the transmission frames into a bitcode portion and a data portion (figure 2). While this will take away from the available bandwidth, it should take no more than a few bytes (at the moment we are proposing that a 4 bit bitcode be used to allow us to be able to uniquely identify 16 connections from any given device).

MAC Wireless Interface 1	BITCODE – connection 1 BITCODE – connection 2 BITCODE – connection 3
MAC Wireless Interface 2	BITCODE – connection 1 BITCODE – connection 2 BITCODE – connection 3 BITCODE – connection 4
MAC Wireless Interface 3	BITCODE – connection 1 BITCODE – connection 2

Table 1. Example global NAT table

Due to personnel constraints we will be constructing the prototype interfaces out of PC's (either laptops or desktops) and implementing the wireless interface software in Linux. By using Linux as our development platform we hope that the software may be ported to special hardware to enable the fabrication of small special purpose pieces of hardware that will be the final wireless network interface product.

Protocol Overview

The protocol developed replaces multiple wire connections to multiple machines with a single wireless interface device (hence forth referred to as a WID) that utilizes peer to peer data transfers. The WID maintains knowledge about all other WID's on the network and what connections those WID's have.

In order to properly forward data from one device specific connection to another device specific connection the WID's utilize a rule table that is setup local to the machine that tells it where data from a specific connection should be sent to. The destination WID sends this data out on the connection specified by the source WID. This whole process should seem invisible to the source device.

Protocol in Depth

Data Transmission

There are two challenges to replacing multiple data links and mediums with one data link and medium, one is developing a protocol capable of packaging multiple types of data and transmitting them over a single medium, in our case RF.

The protocol takes in data, of any type up to 1500 bytes at a time via a struct called data1500. This struct contains information about where the data came from (i.e. from what connection), where it is going to, and how much data is to actually hold in the struct as well as the actual data (up to 1500 bytes).

Once the struct has been built it must be formatted for sending. Since most RF modems can stream data size should not normally be a problem, however we do not want any one radio to be able to monopolize another radio thus essentially denying a connection to others trying to talk, so instead we package the data into 100 byte chunks.

The first 4 bytes of the chunk are used for administrative purposes, such as the connection that it is to be sent out on (the far destination connection, referred hence forth as the DA_DBC), which chunk it is if it is part of a fragmented message, the final chunk ID, the message ID and the original message length. These five things allow the protocol to reassemble the original sent message from multiple chunks of data and send it out on the proper data connection.

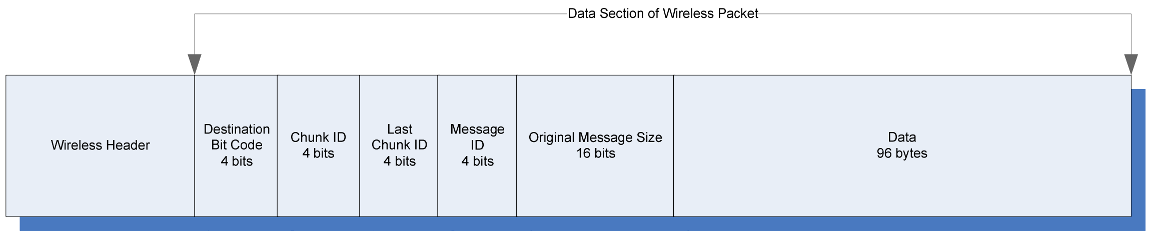

Data Section of Wireless Packet
Wireless Header
Destination Bit Code 4 bits
Chunk ID 4 bits
Last Chunk ID 4 bits
Message ID 4 bits
Original Message Size 16 bits
Data 96 bytes

Figure 2. Frame Layout

Destination Bit Code	Used to direct where the data will be sent out on the device that received the packet
Chunk ID	Allows linear identification of fragmented packet.
Last Chunk ID	Allows us to know how many fragments to look for in a fragmented message, if this is set to 0 and Chunk ID is set to 0 we know there has been no fragmenting.
Message ID	Gives us a unique identifier for a message in case multiple sources are transmitting at the same time and frames are received in a non contiguous manner
Original Message Size	Tells us how many bytes that we need to build into the packet that we are sending out

Protocol in Depth
Wireless Device Addressing

The other challenge, that of the data links brings about the issue of how to address the various hardwire connections that were replaced by the virtual links we are creating.

That is done in two parts. The first is the Destination Lookup Table (henceforth referred to as the DLT), this is essentially a NAT that contains information about every wireless device on the network, including what connections it has running into it. That information must be setup locally on each Wireless Interface Device (henceforth referred to as a WID).

In order for all WID's to have the same knowledge, a master node is used known as the Town Crier. The Town Crier maintains and distributes the Global DLT. This special version of a DLT is almost identical to the DLT that a WID would have, except the Town Crier also keeps state information about each WID on the network. Each WID transmits its local information to the Town Crier in a heart beat message sent out once every 5 minutes; the Town Crier then adds an entry to the Global DLT if no entry for that WID exists and marks the time that it was received and then broadcasts the DLT out to each device on the network. If there is already an entry for that WID the Town Crier compares the data it received to the data it has on file and if they are different it updates the data on file and then broadcasts the Global DLT out to everyone notifying them of the change, if there has been no change between what is on file and what was received it simply updates the time it was last heard from. If a WID is not heard from in 7 minutes the Town Crier sends a heart beat request to the WID in question, if it does not receive a response it removes the WID from the Global DLT and broadcasts the update.

As well as the DLT, which lets a device know what devices are out there, a WID must know what local connection gets forwarded to what far connection. This is accomplished via the rule table, which like the local information, must be setup on each WID as it is most likely specific to that WID. The rule table does not simply consist of DA_DBC's, instead it consists of character tags representing the DBC from which the data came (SA_DBC), the destination address of the far wireless device (DA) and the destination bitcode of the far device (DA_DBC). The character tags are then used to lookup the SA_DBC, DA, and DA_DBC from the DLT so that rules can be built that can be used to fill in these crucial pieces of information required to properly send out any data other than a broadcast.

Implementation Overview

Implementing the protocol proved to be no less of a feat than designing it. Union Pacific originally requested that we develop this project to work on a Rabbit Micro Control board, this however was not possible due to limitations of the hardware further complicated by personnel constraints, however in an effort to make translating the code onto a microcontroller easier all code was written in C and on Linux. C was chosen as it allowed for the fastest running code without getting down to a low level language like assembly or VHDL, Linux was chosen as the development platform as there are many devices running embedded versions of Linux, thus we hope that the code can perhaps be smoothly transitioned in the future.

On top of code and platform considerations the protocol called for our implementation to be invisible to the end point devices. This meant that all connections to the devices from the WID needed to be capable of capturing data not meant for it and also be able to emulate the original sender when sending data to the end point device.

Once the data is captured, a thread hooking into the WID internals is called that allows the WID to process the data received from the end point device. Processing includes chunking the data into 96 byte segments to allow it to be sent out in the protocol formatted frame. Once the data is chunked the frame is finished by a framing unit that looks up the DA and DA_DBC of where it is to be sent and then placed into an out queue.

A transmit thread then sends the data out to the proper recipient. Once on the other side the data is placed into a received queue and then sorted out to the proper connection queue where it is reassembled if it has been fragmented and then sent out on the connection its initial form.

The WID's utilize three non-code files to help with their tasks; these files are DLT.conf, RULES.conf and LOCAL.conf. DLT.conf is built by the Town Crier and then transmitted to all WID's on the network. It contains information about every device on the network and is used by RULES.conf in order to build the rule sets used to direct data transmission. RULES.conf contains information regarding where data should be sent from what connection, this is essentially where the link that we are replacing is specified. LOCAL.conf is used to name the WID device, it also contains the RF modem device address which it probes the radio for as well as all information about connections to the WID from the end point device, this information is transmitted to the Town Crier via a heartbeat thread once every 5 minutes with the first transmission happening immediately after the WID is finished setting itself up. The character tags for the device name and connection names in LOCAL.conf are important to the protocol, since a device will most likely know well before the DLT is received what connections send data to what devices and more importantly, what connections on those devices data needs to be sent to, rules are set in character tags, these tags are then looked up in the DLT to form the rule object which is used to direct data from a connection to a specific connection on a specific device..

Implementation in Depth
Configuration Files

There are three configuration files that are used by the WID's in order to carry out the protocol. These files are DLT.conf; LOCAL.conf, and RULES.conf, example files are located in the CONF_FILES directory of the product code.

DLT.CONF

The DLT.conf file is generated by the Town Crier and then transmitted to every WID that it believes to be on the network at that time. The file contains information about every device on the network and each connection that those devices have, this information is compiled via the information sent to the Town Crier via the heartbeat threads of all WID's. It is in a specific format that is as follows, whitespace does not matter so long as there is some in between each field.

```
#DA   tag   DBC   tag
0x0000      HELLOWORLD  0x0000      CONNA
0x0000      HELLOWORLD  0x0001      CONNB
0x0000      HELLOWORLD  0x0002      CONNC
0x0000      HELLOWORLD  0x0003      CONND
```

Figure 3. DLT format

The first field is the device address of the RF modem connected to that particular device WID, and is made up of four hex characters. The next second field is the character tag describing that particular device, this is used in RULES.conf. The third field is the destination bit code (henceforth referred to as the DBC) of each connection specified on the device, the protocol allows for up to 15 (0x000F is reserved for the Town Crier) connections to be specified, thus only the least significant byte of the four specified in the DLT should be set to identify the connection. The fourth and final field is the connection tag; this again is used to label the connection and is also used in RULES.conf. Thus it is important to know at least what a device is labeled and what the connections on the device are labeled.

As we can see the DLT excerpt in figure 3 specifies a device "HELLOWORLD" with DA 0x0000 and four connections 0x0000 – 0x0003 labeled CONNA – CONND.

Implementation in Depth

LOCAL.CONF

The LOCAL.conf file is setup by the user on the WID. This file contains information regarding the local machine such as the device name (also referred to as the device tag), the device address (which is actually probed from the radio and thus does not need to be set in the file, but is written back out to the file for future reference) and finally connection information.

```
DEVICE_NAME: ARC
DEVICE_ADDRESS: 0x68dd
CONNECTIONS:
0x0000      CONNA eth0  0
0x0001      CONNB eth1  0
0x0002      CONNC ttyS0 9600
0x0003      CONND ttyS1 38400
0x000f      CONNF ttyS2 115200
```

Figure4. LOCAL format

The DEVICE_NAME: DEVICE_ADDRESS: and CONNECTIONS: pre-tags must remain in place. Connection information must also start on the line below the CONNECTION pre-tag and consists of four fields. The first field is the connections DBC; this is used for addressing data to a specific destination connection from the WID and also for properly routing received data to that connection. The second field is the DBC tag and must be referenced in RULES.conf to specify where data from that connection is forwarded to. The third field specifies what connector that connection actually is and is referenced in a Linux format as that is what the WID's are currently built on. The fourth and final field is the baud rate of the connection, this data is not needed for Ethernet connections and thus a 0 is put in its place all baud rates 0 – 115200 with the exception of 76800 are valid baud rates, this should be set to the speed of the device connected to it however. Like DLT.conf spacing does not matter so long as it exists between fields.

We see an example LOCAL.conf in figure 4. This example describes a device referred to as ARC with a device address of 0x68dd and 5 connections, A-D and F. Connections A and B are Ethernet connections as we can see from field 3 and are located on eth0 and eth1 respectively, since they are Ethernet connections baud rate is unimportant to them and thus is set to 0. Connections C, D and F are serial connections on serial ports ttyS0 - ttyS2, they have baud rates of 9600, 38400 and 115200 respectively.

Implementation in Depth

RULES.conf

The RULES.conf file also must be set on the WID. It consists of rules describing how data from which DBC on the local WID should be sent to which far WID and which connection on that WID it should be sent out. Since it is most likely that this information will be known well in advance of the first DLT ever being received it behooves us to set these rules up in advance and fill in information about them once we receive a new DLT.

Rules are set in a specific format, DBC-TAG -> DA-TAG_DBC-TAG. The tags are then used to lookup information in the DBC. All tags must be the same as tags in the LOCAL.conf files that they are originally posted in, thus it is necessary to know these tags in advance of writing the RULES.conf file.

```
CONNA -> Foo_CONN1
CONNA -> GPS_CONN0
CONNB -> Foo_CONN1
CONNC -> ARC_CONN3
```

Figure5. RULE format

An example RULES.conf is provided in figure 5. The example shows us four rules specifying the data forwarding rules for 3 connections, CONNA, CONNB and CONNC. We see that CONNA and CONNB both send data to device Foo's connection CONN1. Also we see that CONNA sends data to device GPS's connection CONN0. CONNA is an example of how one would specify point to multi point connections. Unfortunately specifying a point to multi point connection means that all data will be sent from the source connection to the multiple endpoint connections; at this time it is not possible to direct specific data packet types to specific connections.

Implementation in Depth
Train Device to Wireless Interface Device

Data is captured from the train device via two types of transmission mediums, Ethernet and Serial. The data is captured via two threads running functions designed specifically to grab any data sent out on the line, thus turning the WID into an un-intended listener and invisible to the end point device connected to it.

- **DynamicReadThreadCreate (dynamicReadThreadCreate)**

The purpose of this function is to dynamically create the listen threads so each interface device can communicate with its train device's connections, being either Ethernet or Serial connections. The function parses the localInfo struct myInfo, which knows what connections are present on the interface's train device. The listen threads are passed a connectionArgs struct containing the DBC for the physical train connection and the character string name of the connection for setting up sockets bound to specific hardware on the interface.

- **Listen Serial Thread (comReceive)**

A serial thread is spawned for each serial connection on the Interface device. This thread runs continuously checking the serial port for new data coming from a train device. The serial port is accessed through the POSIX library. The serial port is set to the appropriate baud rate, which is defined in the LOCAL.conf configuration file. The serial port is then configured to local mode, a character size of 8 data bits, no parity checking, raw input and output mode, and a force to wait on at least one character. The serial port was configured in this way so the Interface Device would be as transparent as possible and not change any of the data coming from a train device. After the serial port is setup, the thread continuously loops-checking for new data. When data is received it is put into a buffer of 1500 bytes. When there is no more room in the buffer for new data, the buffer is sent out and cleared. Also, if the serial thread has not received data for 50,000 microseconds and there is data in the buffer, the buffer is sent out and cleared. The buffer gets sent to the CTX thread.

- **Listen Ethernet Thread (ethReceive)**

Like the Listen Serial Thread, the Listen Ethernet Thread opens a socket for communication from the interface device to the train device. The main function is void *startserver(void* arg), which is called from file dynamicReadThreadCreate.c. The argument to this function is the struct connectionArgs defined in protocol.h, which contains the DBC for the thread, and the connection "type", which is a string literal or name of the specific Ethernet hardware. For example, "eth0" would be passed as the type. A RAW socket using the AF family, set to promiscuous mode, and set for all protocols for Ethernet using ETH_P_ALL, defined in linux/if_ether.h. Other Linux structs used are socketaddr_ll to store transmitted information and ifreq struct to store hardware information.

Implementation in Depth

Wireless Interface Device to Train Device

Data is sent from the WID to the Train Device via a single function that switches on the type of connection that the data is addressed to.

- **Write Data (writeData)**

This function receives the data1500 struct and determines what type of train connection this data is bound for. It looks at the da_dbc field in the data1500 struct and enters either the write data Ethernet function or the write data serial function. This function is linear, no more threading is involved when called.

- **Write Data Serial Function (comSend)**

The write data serial function is used to write data to a connection on the Interface device, which is then sent to a train device. This function is called each time a frame of data in the central queue has a destination bit code for a serial connection. The serial port is accessed through the POSIX library. The serial port is set to the appropriate baud rate, which is defined in the LOCAL.conf configuration file. The serial port is then configured to local mode, a character size of 8 data bits, no parity checking, raw input and output mode, and a force to wait on at least one character. The serial port was configured in this way so the Interface Device would be as transparent as possible and not change any of the data being sent to a train device. After the serial port is setup, the function writes a frame of data to the serial port.

The main functionality is within a loop where the function receives Ethernet packets from any location, stores the packet inside a data1500 struct, then creates a thread for the function ctx() giving data1500 as the argument. The function loops back, waiting for more packets from the train device.

- **Write Data Ethernet Function (ethSend)**

The write data Ethernet function is used to write data to a connection on the Interface device, which is then sent to a train device. This function is called each time a frame of data in the central queue has a destination bit code for a serial connection. A RAW socket is opened using the AF family, set to promiscuous mode, and set for all protocols for Ethernet using ETH_P_ALL, defined in linux/if_ether.h. Other Linux structs used are socketaddr_ll to store transmitted information and ifreq struct to store hardware information. The function then sends the data passed into it, then closes the socket.

Implementation in Depth

Wireless Interface Device to Wireless Interface Device

The communications between WID's is where a significant portion of the protocol takes place. It is in this phase that the original data is packed into the Town Crier protocol and transferred over the air to the far WID. Once there it is reassembled into its original format and set out over the connector specified in the Town Crier packet.

- **Data Queues**

Data is moved about the WID via Circular FIFO queues, aka cfifo's. These queues allow us to put protocol frame structs into them and are then assured that the frames will be removed in the order they were received. 18 cfifo's are utilized in the WID code, 1 for each possible connection to the WID(0 - 14) 1 for the Town Crier (15), for the Rx Thread and for the Tx Thread (16 and 17 respectively).

- **Circular FIFO Queue (cfifo)**

For data queues in our application we used a circular first-in-first-out queue. This data structure was used instead of a dynamic queue to minimize memory usage. There is a main outgoing circular queue, which a central thread pulls data from to send to the wireless radio. This is done so data is not lost while the thread is in the process of sending data to the wireless radio. Each connection on the Interface Device has its own circular queue. There is a thread for each connection, which pulls data from its own circular queue and sends to the connection. This is done so data is not lost while a thread is in the process of sending data to the connection. The queue is done in a first-in-first-out method so the first device to receive a message will receive it first, and the first device to send a message will send it first.

The circular queue has a set allocation, which can be changed at the FIFOSIZE definition in the protocol header file. An array is used to keep the queued data. The array is defined in a struct with other variables used for management. Head is used to point to the head of queue. Size is used to keep the total size of the queue. Fifo_num is used to tell which global semaphore the queue is to act on. The semaphore is used so reading and writing from the queue is done exclusively.

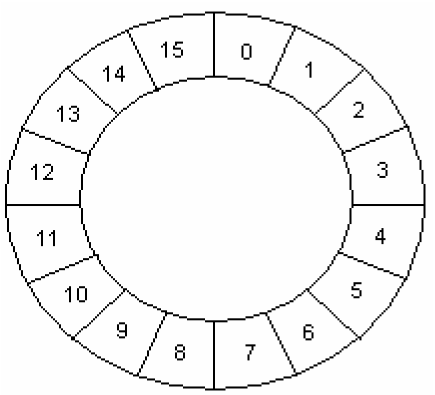


Figure6 An empty CFIFO

Implementation in Depth

Wireless Interface Device to Wireless Interface Device

Figure 6 shows the circular buffer. The numbers are the indexes into the array. The fifo structure is acted on by functions to keep the circularity of the buffer. A setup_fifo function is used to initialize the fifo by setting the head and size to zero and associating a global semaphore with the fifo. A read_remove function is used to pull the first data frame off the fifo. This function will return a one on success and a zero if there is no data in the buffer. The function will also increment the head to the next position in the buffer. If the head is at the end of the array the read function will increment the head to the beginning of the array. The check_read function will read data from the head of the fifo but will not increment the head, so the data will not be "pulled" from the fifo. The check_size function will return the size of the buffer. The write_fifo function will write data to the fifo, putting it at the end of the queue. Write_fifo will return a one on success and a zero if the queue is full. The write_fifo will also increment the size. The last function is lookup_frame which will return a specific frame of data based on a number given. If lookup_frame is given a 5 it will return the 5th frame of data in the queue. Lookup_frame returns a zero if the data cannot be found.

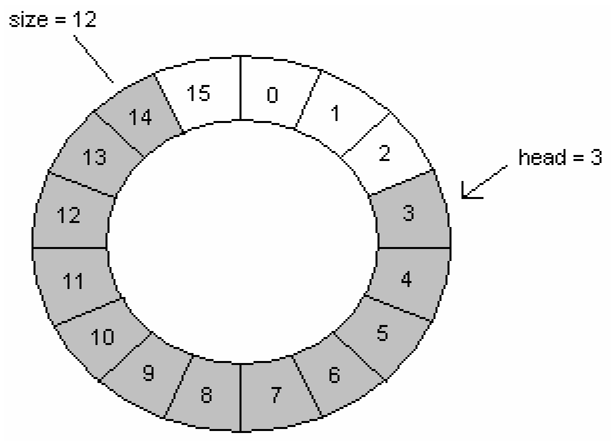


Figure7 Data in CFIFO

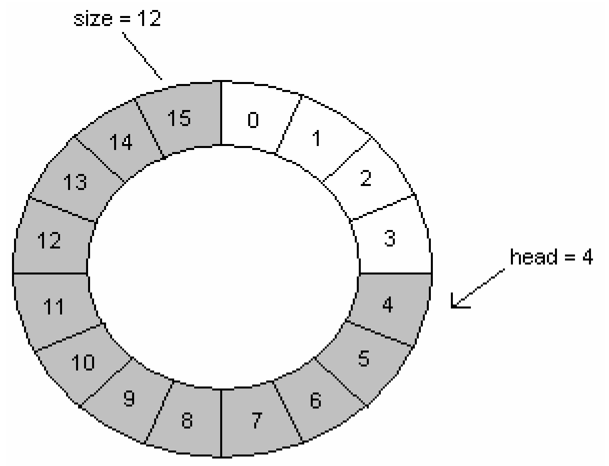


Figure8 CFIFO after reading and writing data

Implementation in Depth

Hook into WID Internals (Tx)

In order to allow the code that interfaces with the endpoint devices to hook into the WID and return to processing data from the interface (i.e. listening for data) a hook function that runs as a thread was created.

- **CTX (central transmit mediator)**

The CTX thread function takes as a parameter a data1500 pointer which contains the data received by the listening threads and acts as a mediator to insure it gets processed, thus allowing the listen threads to listen without worrying about the data getting processed. It then sends the data to the chunking function which returns a buffer of data chunks. The thread then sends each chunk to the framing function which places the data into the Tx FIFO. At this point the data is done being processed and the memory of the original data1500 pointer is freed.

- **Chunker**

The chunker takes the 1500 bytes of the data1500 struct and chunks them by placing it 96 bytes at a time into chunk structs consisting of 100 bytes starting at the fifth byte as the first four bytes are reserved for protocol usage. The chunker then sets a message ID generated by rand() % 16 guaranteeing us that the resultant message ID is no larger than 4 bits and will fit into the packet format. As well as the message ID the chunker sets the Chunk ID and Final ChunkID.

- **Framer**

The framer takes in a single data chunk at a time, looks for the rules associated with the source DBC and generates frames with the proper DA_DBC set in the packet. The generated frames are then placed into the CONN[17] queue which is a CFIFO queue specifically reserved for outgoing data (i.e TxQueue).

Implementation in Depth
Hook out of WID Internals (Rx)

In order to get received data out of the WID and back out to the end point device the data must be first processed and then sent out using a hook out of the internals that was previously mentioned, the write data function.

- **Sorter Thread**

The sorter thread checks the receive queue (CONN[16] is reserved for incoming data) for data and if data is there, moves the data from the receive queue, to the proper connection queue where it will be processed by the CRX thread that is responsible for that connection

- **CRX Create**

CRX Create is called at the beginning of the WID's operation. It dynamically creates a number of CRX threads that will be responsible for processing data on the specific connection queues that are in use. This is done based off of information from LOCAL.conf.

- **CRX Thread**

The CRX Thread is started via the CRX Create function. It takes a parameter allowing it to identify which connector it is working on. The thread runs forever and continually checks the associated cfifo for data. If there is a data frame in the cfifo then CRX grabs it and checks to see if it has already started building a message that has the same message ID as the frame it just received, if it has it builds the data onto the message that is in construction, if it is not part of an already existing message the thread will construct a new buildup struct and keep the message until it has been completed. A CRX thread can build up to 16 individual messages at a time. When the message has finished its reassembly (we know this by the number of bytes in the original message, and the number of bytes of our reassembled message) it is sent out to the train device over the connector specified by the queue via the already mentioned write data function. The buildup struct is then wiped and a new message can be built in it.

Implementation in Depth
Node Management
The Town Crier

The Town Crier is a network management node that becomes necessary if you want the network to intelligently be aware of other devices on the network and changes that take place to the network. All nodes on the network transmit their local information to the town crier via the Heart Beat Thread that is to be discussed later. The Town Crier receives this data and checks the Global DLT that it maintains, if the device information is already in the DLT and the information is exactly the same as the previous set it received, the Town Crier updates the last heard from time. If the information is new or different from what is on file in the DLT the Town Crier updates the information and sends out the updated DLT to all nodes on the network so that they can update their information as well.

The Town Crier also continually checks the last heard times of the devices the in DLT against the current time to see what devices may have timed out. If a device has not sent a heartbeat message in 7 minutes the Town Crier sends a heartbeat request to the device, if it does not respond with a heart beat message within 1 minute it is considered down and removed from the DLT. The DLT is then broadcasted to all nodes on the network so that they can make appropriate updates as well.

The Town Crier is not necessary for WID usage, however you must configure a DLT.conf file by hand if you want to not use the Town Crier.

WID Heart Beat Thread

In order to maintain state information about the network the WID's have a thread constantly running, much like the many other threads that constantly run. This thread has one specific purpose in that on a set interval it sends its myInfo data (the localInfo struct built up when LOCAL.conf is parsed) to the Town Crier. This is known as the heartbeat message and is sent once every 5 minutes.

WID Heart Beat Request Response

In order to respond properly to a heart beat request the WID has a function which when it sees a heart beat request, replies o the Town Crier with its local info. The WID heart beat request response is tied into the normal receive data flow.

RF Modems

Up until now all code we've discussed is radio independent. This means that so long as functions that do what the following functions do are written to interact with another transport medium that any transport medium could theoretically be used.

There are two files that contain information specific to the radio, these files are radio.h and radio.c. With the .h file specifying functions used in the .c file.

Configure Modem Connection (In & Out)

There are two versions of the Configure Modem Connection function, one configures a connection for reading data, the other for writing data, and both must be called on the same device connector in order to open up full duplex mode on that connector. These functions are use to set the two file descriptors used to communicate with the radios, RADIO_FD_IN and RADIO_FD_OUT. The Connections are opened in a raw mode so that all data transmitted and received is taken as is and not formatted depending on the bytes sent. All connections are opened at a baud rate of 115.2Kbps.

Modem Command

Modem Command is a function used to send AT commands to the modems. These commands are specific to the modems we chose but other modems would surely have a similar feature, thus the Modem Command function takes a string of characters to send, the file descriptor on which to send them, and a buffer to capture the response. This is the function that we use to probe the radios in order to retrieve their DA. All AT commands have responses to them.

Get SADA

Get SADA utilizes modem command and sends out the proper AT commands to retrieve the DA of the radio attached to the WID. These commands are
+++ (enter command mode)
ATMY (probe for my DA)
ATCN (leave command mode)

RF Modems

Send Data

Send data is a generic write function that utilizes the write command, it takes in a buffer to send, the file descriptor over which to do it and the number of bytes to send out.

Send Data Thread

The send data thread is began at startup and runs forever like many other threads. It checks the CONN 17 buffer which is the buffer reserved for frames that are to be transmitted. If there is data in the queue (using a function of cfifo to check the queue for the number of frames currently in it) the send data thread will read a frame from the queue and frame it into the modems own packet that will be sent out. This packet is 109 bytes long, and consists of an 8 byte header and a 1 byte checksum following our data. Once the packet has been framed up for the modem it sends the frame out over the air.

Receive Data Thread

The receive data thread is also began at startup and also runs forever. This thread reads in 109 bytes at a time from the modem and then places the 100 bytes of our frame into the receive queue (CONN 16 is reserved for this purpose). The thread also verifies that the packet received is indeed a receive status packet by checking the message after 4 bytes have been received, if it is not the read is reset.

NOTE:
For more information about the radios used please read the beta documentation from MaxStream included with this manual.

Hardware Overview

Various pieces of hardware have been necessary for the completion of this project. We have had to construct a test network consisting of 3 nodes, each node however consists of a train device and a wireless interface device, thus we required 6 computers, plus one more to function as the Town Crier. All machines are various Pentium 4s with varying amounts of ram and all are running the Linux distribution, Fedora Core 4.

On top of the PC's that we've had to setup to allow emulation of a network in the lab, this project would not have been possible without the RF modems. We looked at many different technologies, 802.11, Bluetooth, UWB, and a plethora of RF modems in the 900 MHz and 2.4 GHz range before eventually settling on an RF modem manufactured by MaxStream. We chose to use MaxStreams 9xTend 900 MHz RF modem as it had a feature set unbeatable by other manufacturers.

The 9xTend is capable of an over the air throughput of 115.2Kbps and had a developers kit that placed the modems onto an interface board that allowed 4 computer scientists to develop for it without having to have intimate knowledge about how to send data to and from a device through anything but C and a little serial programming.

On top of the 9xTends over the air throughput it also had a built in AES encryption module eliminating the need for software encryption which would have been computationally expensive and rather slow. While a radio could receive the encrypted data without knowing the key it was encrypted with, the data would just be tossed as it would appear as gibberish, thus only radios that have the same encryption key would be able to communicate.

As well as the encryption key acting as a way to keep certain devices from communicating with one another the radios will not talk to radios that do not have the same network ID set, thus you can again limit which radios talk to which radios.

On top of all these other features MaxStream agreed to give us a beta firmware for the radios that allowed us to use an API to format radio frames by hand and send them out to the radio. This allowed us to format frames to send to individual radios without having to change who the radio was set to talk to in its firmware, which would involve computationally costly AT commands.

The beta firmware however has some things in it that are disabled as their functionality has not yet been implemented. The most worrisome of these features is the encryption. In the beta firmware the device encryption is not able to be used, we have been assured by MaxStream that in the final release this issue will be resolved and hardware encryption will once again be enabled.

Project Results and Conclusion

Protocol Results

The problem presented to us by Union Pacific was both interesting and difficult, create a protocol that can be used to transport two types of transportation mediums across a wireless medium and also develop a schema for the network. Our solution to this problem was equally interesting.

Our first step was to tackle the problem of data transportation. Thus we developed the wrapping protocol which is able to wrap both serial and Ethernet traffic and move it across a wireless link, and in fact can most likely move any type of data across the wireless link. The software implementation of this protocol has 3 parts, end point device to WID, WID to WID, and finally WID to end point device. In the end point WID interaction there are hooks that can be used to move any data across the WID link. A hook into the WID where data is to be passed in can be used by any programmer to use the WID link to move their data to the far WID, and a backwards hook into the WID that is used to retrieve the data from the far WID. These hooks are what allow anything to be sent across the WID link. We developed two modules that utilize these hooks that transport both serial and Ethernet data. The serial module receives and sends data in a raw mode so as to not interfere with the data in anyway. The Ethernet module does much the same as the serial module but is a bit more complicated in that it acts as an unintended listener and receives all packets on the line tied to it despite whom they may be addressed for, and on the backside it spoofs the packet out exactly how it was received.

Since there was a possibility that the radios we chose to use for this project would not be deployable by Union Pacific the code interfacing with the radio was kept separate than the rest of the WID code. If a different RF modem, or any other wireless or wired transport medium were decided upon a programmer would only have to rewrite the radio.c and radio.h files and write functions named the same as those in radio.c and radio.h that have the same functionality as the originals to interface with the new transport medium.

The next problem was how to manage the WID nodes properly in a network. This presented us with a few different options where we eventually decided to utilize a management node protocol that we developed called Town Crier. Town Crier utilizes a single node that does not interface with any end point devices. This node maintains information about all WID's on the network via a heartbeat message containing information about the device that each WID transmits to a set RF device address 0x7FFF, while this is RF modem specific, it could be easily changed if a different wireless medium were decided upon. If a heartbeat message contains new information the global list is updated and sent out to everyone on that list. If it's old information the Town Crier updates the timestamp on the device. If the difference between the timestamp and the current time is 7 minutes or greater the Town Crier will time the device out and update the list.

Project Results and Conclusion

RF Modems

For the wireless component to this project we examined various technologies from the IEEE approved 802.11 and Bluetooth to UWB and many RF Modems in between. We needed to find a wireless technology and device that had certain features.

- Individually addressable
- Transmission Control (i.e. ACK//NAK)
- Capable actual RF throughput

802.11 technologies were ruled out by Union Pacific as they wanted to explore other solutions. Bluetooth was ruled out due to its rather low transmit power, this perhaps could have been augmented with an amplifier but without an electrical or RF engineer on the team we decided against this. UWB ruled itself out by the fact that we could only find one company that manufactured a UWB radio modem. And of the RF modems one stood out above the rest, MaxStream's 900 MHz 9xTend RF modem. This radio was uniquely addressable and had RF level transmission control all at an actual data throughput rate of 115.2 KBps, as an added bonus it had hardware level encryption using AES.

The radios came in a development kit that had them attached to a microcontroller which allowed us to interface with them over a serial line. Thus we were now limited to the maximum output speed of the radios, allegedly 115.2 Kbps.

In order to make our implementation of the Town Crier protocol as efficient as possible we opted to use a beta firmware (2015) from MaxStream that has not yet been publicly released. This firmware allows us to set the radios into an API mode that allows us to create entire RF frames and send them to the radio for immediate transmission versus having to send commands to the modem to have it switch settings such as destination address. However this firmware, being a beta firmware, has broken some of the features that the 9xTend normally has though we have only verified one, encryption. We know that the beta firmware has the AES encryption disabled, MaxStream says this will be enabled at full release. While a software implementation of AES could be used to encrypt data before it hits the radio that would not be the proper way to create an embedded device as you would want separate hardware to run the encryption, thus the radio implementation would definitely be better than a software implementation.

Project Results and Conclusion

RF Transmission Issues

Though the data sheet for the 9xTend claims to have a 115.2 Kbps throughput, in practice we have found that we are only able to attain a fraction of that speed, approximately 2.5 Kbps. While we are not sure why this is we believe it to be a setting on the modem that we have not yet identified. Thus the link is only capable of handling data at that speed. Any faster than that and data is lost. This is due to the RF Modem itself. The 9xTends have two buffers, a DI and a DO that are used for data being transmitted and received. These buffers are 2.1K a piece, once the 2.1K is full though any data that is sent to the radio is dropped. Thus the absolute data limit that the protocol can handle is dictated by the speed of the wireless link.

$$\sum \frac{TotalDataOutfromDevice}{Time} \leq SpeedofWirelessLink$$

If this equation is not observed then data will be lost due to the way the RF modems operate. A solution to this problem could be to buffer the data in software to insure that all data is transmitted, however if the data coming into the WID is coming in at a rate faster than the WID can transmit the data the buffer will overflow and data would be lost unless there are sufficient breaks in data transmission so that the buffer could be sent. This solution would rely upon a bad assumption that there will be breaks in the data flow of sufficient length to allow the buffer to clear out. The optimal solution is to replace the link with a technology that has a larger data throughput such as UWB or any of the 802.11 technologies.

Despite the transmission speed issues the 9xTends are capable of transmitting at 1 watt and thus have excellent range. Interference testing through multiple walls (most likely cement w/rebar) revealed that we were able to maintain a link to the radios, additional interference tests were run with the radios on separate floors (though near windows), 1 floor differences did not present a challenge, however 2 floors of difference did and while the radios were still able to transfer data, more data was lost than was acceptable. Distance testing within the Michigan State University Engineering Building we were able to attain a distance of 500 feet and still maintain a reliable link.

Program Compilation and Running

In order to compile the WID run ‘make WID’ from the main code directory.

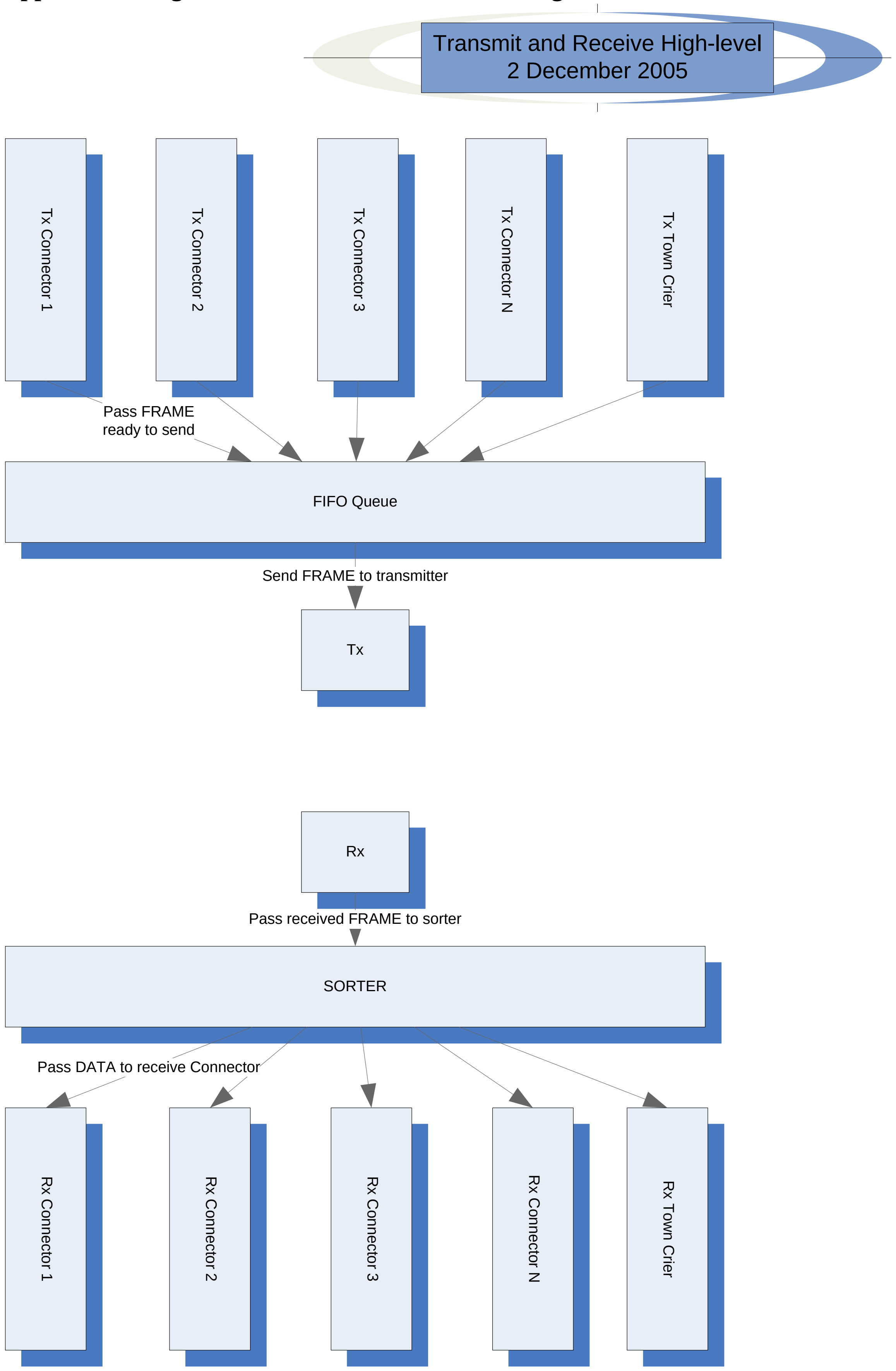
Before running the WID make sure to setup the LOCAL.conf and RULES.conf as specified earlier in this document.

Then to start the wireless interface device type ‘./WID’.

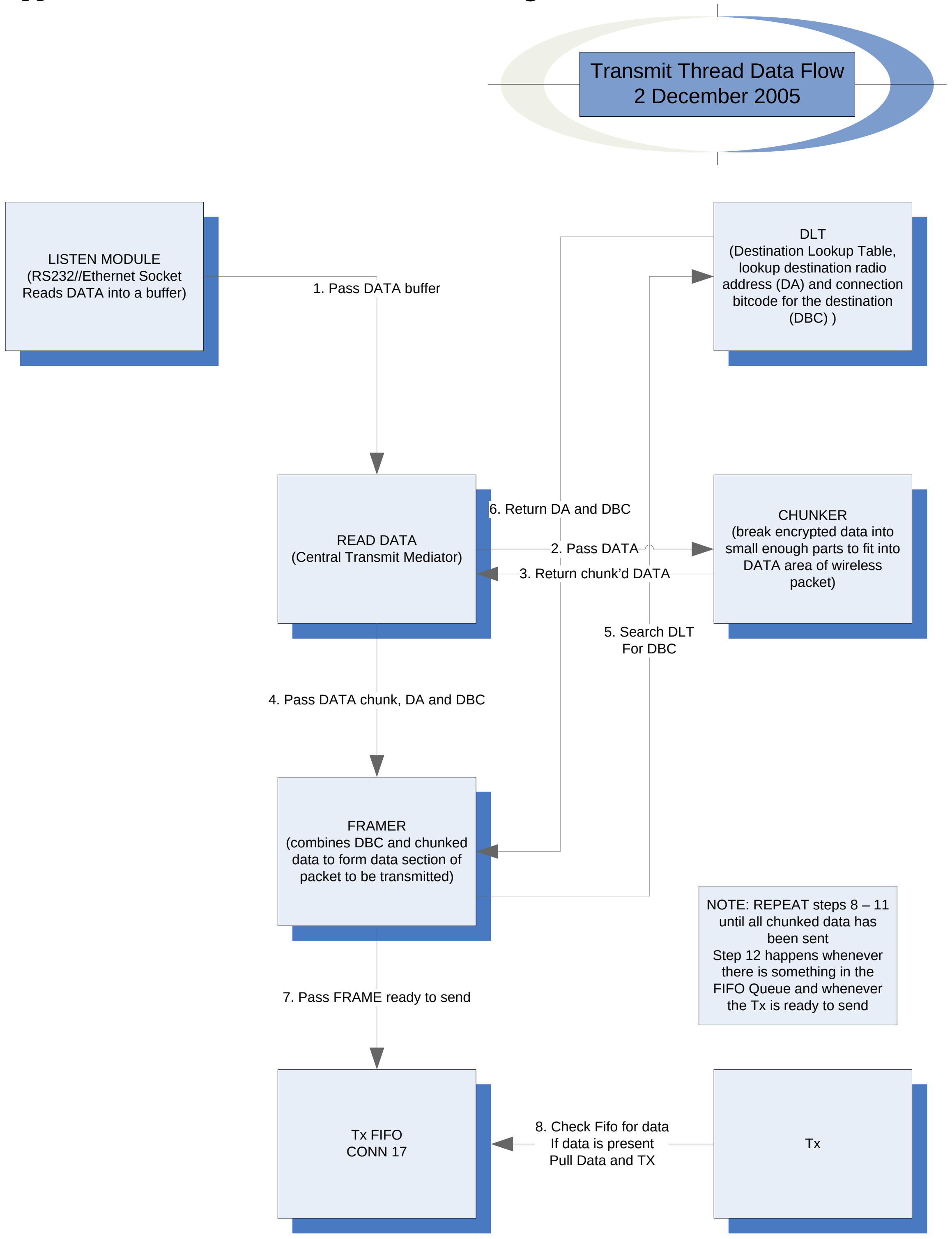
In order to compile the Town Crier run ‘make TC’ from the main code directory.

Then to start the wireless interface device type ‘./TC’.

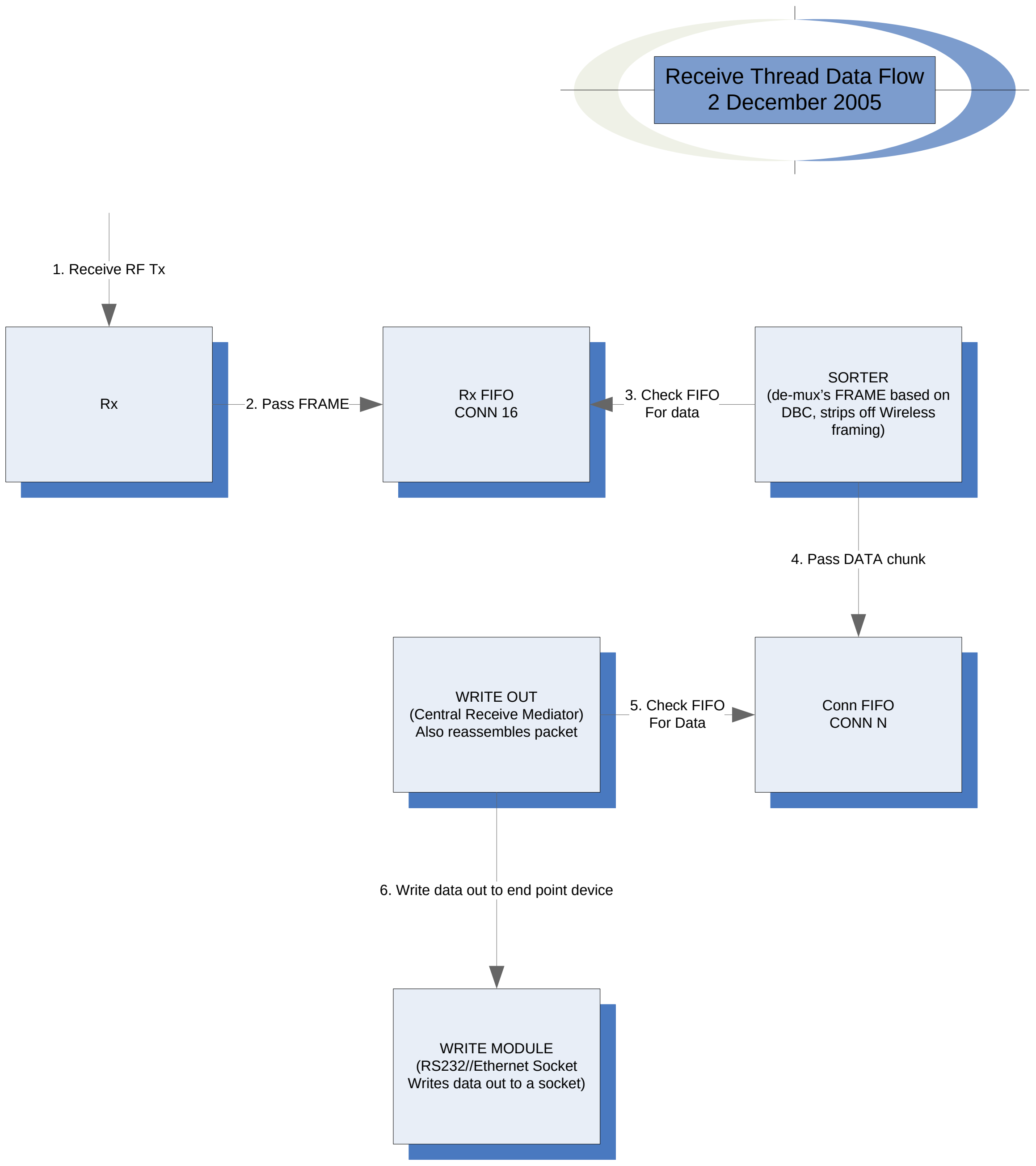
Appendix A: High Level Transmit and Receive Design



Appendix B: Transmit Thread Data Flow Design


Transmit Thread Data Flow
2 December 2005
LISTEN MODULE (RS232//Ethernet Socket Reads DATA into a buffer)
1. Pass DATA buffer
DLT (Destination Lookup Table, lookup destination radio address (DA) and connection bitcode for the destination (DBC))
READ DATA (Central Transmit Mediator)
6. Return DA and DBC
2. Pass DATA
3. Return chunk'd DATA
CHUNKER (break encrypted data into small enough parts to fit into DATA area of wireless packet)
5. Search DLT For DBC
4. Pass DATA chunk, DA and DBC
FRAMER (combines DBC and chunked data to form data section of packet to be transmitted)
NOTE: REPEAT steps 8 – 11 until all chunked data has been sent
Step 12 happens whenever there is something in the FIFO Queue and whenever the Tx is ready to send
7. Pass FRAME ready to send
Tx FIFO CONN 17
8. Check Fifo for data If data is present Pull Data and TX
Tx

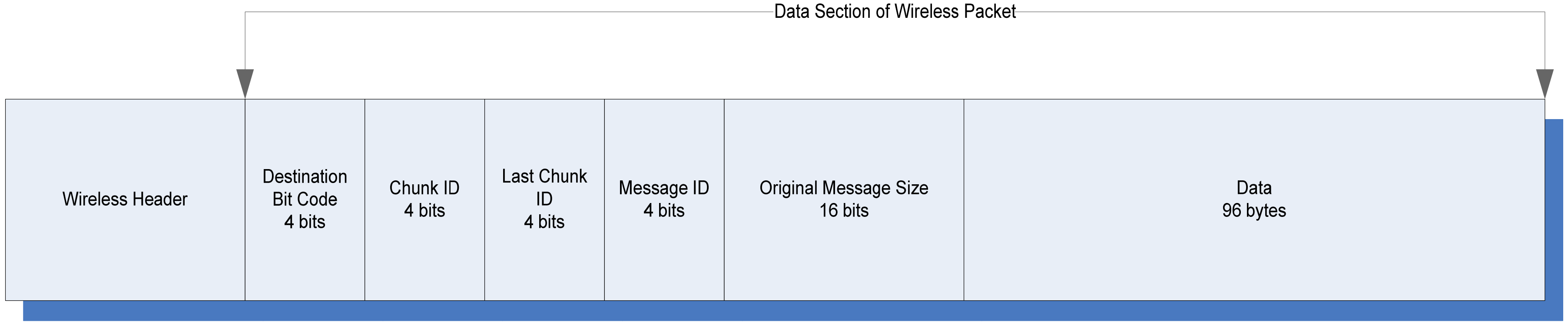
Appendix C: Receive Thread Data Flow Design



Appendix D: Town Crier Wrapper Protocol API, Packet Format

Bitcode API
2 December 2005

WRAPPER PROTOCOL API
The wrapper protocol will utilize the data section of the wireless packet. It will take exactly 32 bits. The first 4 bits of the data packet will be used for the Destination Bit Code (DBC). The 4 bit long DBC will allow us 15 connections +1 reserved for the Town Crier threads. The next 8 bits are used for fragmenting larger packets. The first 4 bits of the 8 are used to identify which data chunk is in the data portion of the packet. The last 4 bits of the 8 are used to identify the last data chunk expected. When the Chunk ID and the Last Chunk ID match we know we have received the last data chunk. If Chunk ID and Last Chunk ID are 00000000 then the data chunk is not a fragment and contains the entire message. The next 4 bits are for the message ID, since multiple fragments from different sources may be received by a WID we need to be able to uniquely Identify a message, this value is received from the rand function mod 16. The next 16 bits are used to store the original message size which allows us to rebuild the original message.


Data Section of Wireless Packet
Wireless Header
Destination Bit Code 4 bits
Chunk ID 4 bits
Last Chunk ID 4 bits
Message ID 4 bits
Original Message Size 16 bits
Data 96 bytes