COMPUTER SCIENCE MASTERS PROGRAM

XML Sensor Compression-Decompression for Wireless Sensor Networks Applications

Class:	CSS600A (3nd Phase)
Name:	Steve Dame
Student ID:	314161
Date:	Dec 14, 2011

Table Of Contents

1. Introduction

This is the third special project related to my eventual master's project in wireless sensor networks. The over-arching objective of this wireless sensor networks system is to be able to aggregate data from tens of thousands to millions of sensors into a very large database. Thus each building block on the path to this large robust system needs to be developed. The scope of this portion of the project is to continue to explore compression and decompression of XML sensor data and to develop an XML codec (compressor – decompressor). A company (AgComm, Inc.) was formed to potentially market a successful wireless sensor technology development and in the some of the following text the "AgComm XML" format will be referenced in various sections.

2. Background

In phase one of our CSS600 special projects class, we explored XML compression and wrote some code to compress XML data into a much smaller payload than the verbose XML. Although the initial compression space was explored, there was still more research and development needed before a good design for an XML codec. In this previous effort WBXML was researched and found to be too opaque to work with, not very mainstream technology any longer (developed previously at Nokia), and abandoned. The current industry trend seems to be towards the W3C accepted standard for XML compression known as Efficient XML Interchange (EXI) .

In phase two, we developed some firmware on the PSoC3 microcontroller to sense temperature and produce XML packets (uncompressed) which were then connected to the Valhalla Wireless Viking remotes (Remote_One and Remote_Two) whereby these XML packets were just forwarded as payload to the Base wireless gateway unit. These packets were blended in with the Viking output format and each record output from the Base station unit was either in the native Viking format or the XML AgComm format. In this phase, we were not able to handle the non-XML packets so we filtered them for purposes of setting up a client / server connection to be able to serve up these packets to the webserver (cssvm01.uwb.edu) or any other server on the network. We used dRB (Distributed Ruby) to successfully design this client server distribution of the wireless sensor data.

The non-XML format of the Viking base station gateway sensor output is a proprietary format that splits the output into a radio identification record (with location) and a sensor record which is on a different output line from the device (connected via UDP to the "Hercules" lab server -- hercules.uwb.edu). A sample of this format is as follows.

```
Base, 0x227c3c, lat: 46.000000, lng: -119.000000, elv: 0.00
2011-12-14 21:14:30,60,0x227c6a,67.99,12.18,-82,
2011-12-14 21:14:41,60,0x227c3c,69.00,4.92,-105,
2011-12-14 21:15:23,60,0x227bdb,70.30,12.17,-78,
Remote_One, 0x227c6a, lat: 46.000000, lng: -119.000000, elv: 0.00
2011-12-14 21:15:29,60,0x227c6a,68.05,12.19,-92,
2011-12-14 21:15:41,60,0x227c3c,69.03,4.92,-102,
Base, 0x227c3c, lat: 46.000000, lng: -119.000000, elv: 0.00
2011-12-14 21:16:26,60,0x227bdb,70.33,12.18,-74,
2011-12-14 21:16:28,60,0x227c6a,68.02,12.18,-84,
2011-12-14 21:16:41,60,0x227c3c,69.03,4.93,-106,
Base, 0x227c3c, lat: 46.000000, lng: -119.000000, elv: 0.00
Remote_One, 0x227c6a, lat: 46.000000, lng: -119.000000, elv: 0.00
2011-12-14 21:17:27,60,0x227c6a,67.99,12.18,-80,
2011-12-14 21:17:29,60,0x227bdb,70.27,12.18,-106,
Base, 0x227c3c, lat: 46.000000, lng: -119.000000, elv: 0.00
2011-12-14 21:17:41,60,0x227c3c,69.06,4.93,-99,
Base, 0x227c3c, lat: 46.000000, lng: -119.000000, elv: 0.00
Remote_One, 0x227c6a, lat: 46.000000, lng: -119.000000, elv: 0.00
2011-12-14 21:18:26,60,0x227c6a,67.93,12.18,-88,
Remote_Two, 0x227bdb, lat: 46.000000, lng: -119.000000, elv: 0.00
2011-12-14 21:18:31,60,0x227bdb,70.21,12.18,-75,
```

Sample 2.1 - Viking Radio Proprietary Format

The AgComm XML data format creates a schema for all sensors to be self describing using structured formats that database parsers can more easily understand. The following is an example of a couple or sensor packets that contain sensor_id, time and temperature data elements. Each of these packets is well formed and has additional attributes to more fully describe the parametric nature of the data, its units, type and other relevant attribute data about any particular element. For better readability the XML is run through a "Tidy" program to add line breaks and nested indention, but none of this whitespace exists in the in actual XML records (except for a <CR> or <CRLF> at the end of each line (i.e. newline).


```xml
<?xml version="1.0" encoding="UTF-8"?>
<sensor_data>
        <sensor_id capability="T" version="0.12">agcm-fsk0</sensor_id>
        <time>1307298731</time>
        <temperature unit="Fahrenheit" type="ftsk_therm">79.2</temperature>
</sensor_data>

<?xml version="1.0" encoding="UTF-8"?>
<sensor_data>
        <sensor_id capability="T" version="0.12">agcm-c600</sensor_id>
        <time>1307297366</time>
        <temperature unit="Fahrenheit" type="omega">74.1</temperature>
</sensor_data>
```


Sample 2.2 - AgComm Radio XML Format

If both the Viking radio and AgComm radios are operating, then therefore, there will be both types of records arriving at different times at the server UDP port. Before we can deal with compressed and uncompressed data, we need to agree on a design that will allow for the standardization of all data records to arrive at the server to either be XML or compressed XML.

Therefore it was best that we first design a "Viking_to_XML" filter which can convert the Viking packets into AgComm formatted XML packets. This is somewhat challenging, because the filter has to remember state information from previously input lines in order to convert everything on its output to a standardized XML format.

Figure 2.1 shows the flow of sensor data through the various hardware components of the system. The system is a multi-tier wireless sensor network, with a long range back-haul radio system and Linux based server on the backend to receive the incoming sensor data packets. The low power *AgBee* radios provide a diverse quantity of low cost and low power sensors which are aggregated together through an *AgBee Concentrator* embedded radio and system on a chip interface. This concentrator is connected to a 900MHz back-haul remote radio via a hardwired serial port running at 115Kbaud.

On the left, the sensing is performed in the *AgBee* radio/sensor devices, converted to digital compressed XML ("XMLC") packets and sent through the 2.4GHz network, bridged through the *AgBee-Concentrator* to connect through the 900Mhz backhaul network from Valhalla Wireless. The data packets are then routed through a server which the 900MHz base station is connected to via Ethernet. UDP packets are sent to this Linux server (hercules.uwb.edu) and then they are passed to the CSS Virtual Machine (cssvm01.uwb.edu). Since the focus of this study was to concentrate on the XML Codec, only the components relevant to that part of the system are shown below. The Valhalla Wireless Sensor Protocol ("VWSP") data is interleaved with the XMLC and both are received at the host server for splitting and decompression on Hercules. The output then can be seen as straight XML such as in Sample 2.2 above.

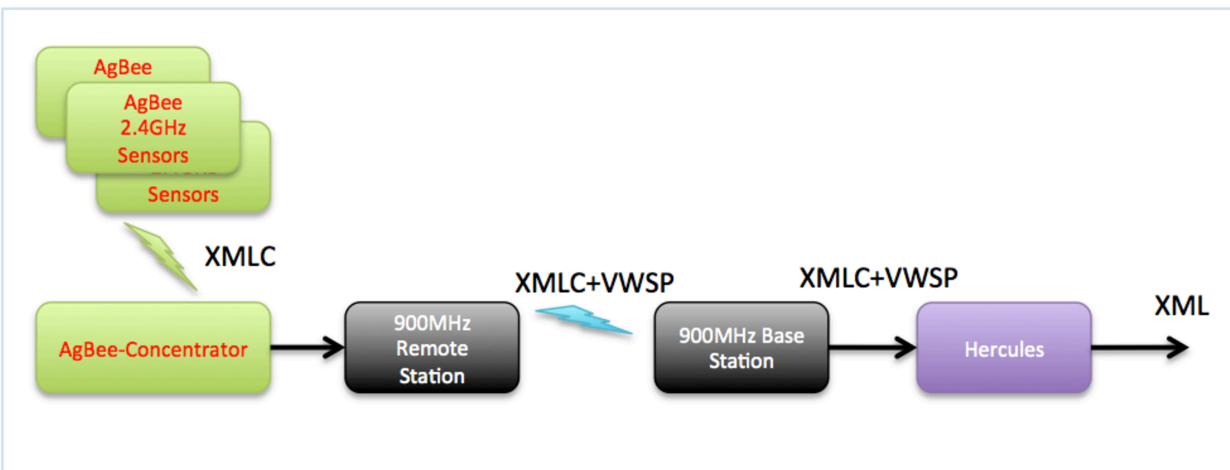

Figure 2.1 - XML Sensor Data Packet Flow – Hardware Components

3. Components of the Software System

The software components consist of a Java UDP listener program (provided by Valhalla Wireless), a program to convert the proprietary output of the Viking radio into XML (Viking_to_XML.rb), XML_Encoder.rb, XML_Decoder.rb and a file for storing the XML Codec training data that can be output from the xml_codec.rb class library. XML_Encoder.rb, XML_Decoder.rb are just wrapper functions that instantiate and call the methods within the xml_codec.rb class library.

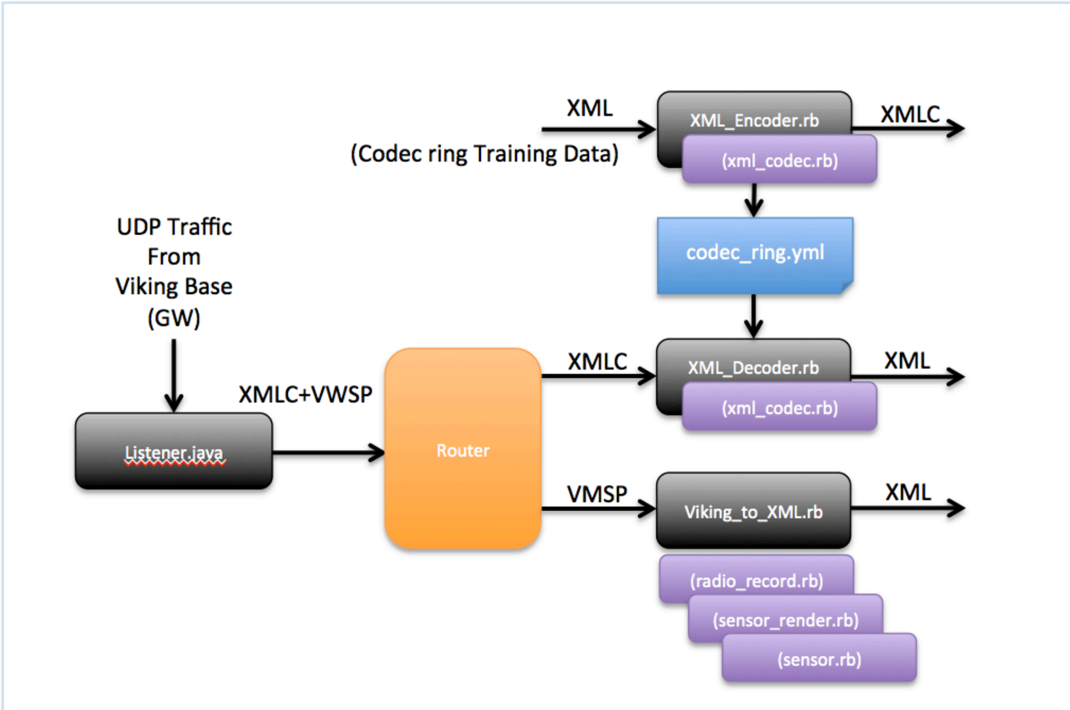

Figure 3.1 - XML Sensor Data Packet Flow – Software Components

Once the incoming data is detected at the output of the Java listener program as being compressed (or uncompressed) XML or Viking data, it can be routed to the appropriate filter or decoder code to extract a common XML syntax that can more easily be understood by numerous database parsing services.

Within the xml_codec.rb class, there are methods to convert and incoming data object from compressed XML to XML ("to_XML") or from XML to compressed XML ("XMLC") as well as methods to create and store a "codec_ring" (encoder/decoder ring) from the underlying schema of elements, attributes and attribute names. For example, the following Viking proprietary data…

```
Base, 0x227c3c, lat: 46.000000, lng: -119.000000, elv: 0.00
2011-12-14 21:17:41,60,0x227c3c,70.51,4.92,-102,
```

Listing 3.1 - Valhalla Wireless Sensor Protocol is converted into XML by piping the listener program through the Viking_to_XML.rb program with the following command line operation

```
$ java listener | ./Viking_to_XML.rb
```

resulting in the following XML.


```xml
<?xml version="1.0" encoding="UTF-8"?>
<sensor_data>
	<sensor_id alias="Base" capability="T:V:R" version="0.1">0x227c3c</sensor_id>
	<time>1323610856</time>
	<temperature unit="Fahrenheit">70.51</temperature>
	<battery unit="Volts">4.92</battery>
	<rssi unit="dBm">-102</rssi>
</sensor_data>
```


Listing 3.2 - AgComm XML format for Wireless Sensors

This XML can be used as training data for the xml_codec.rb program to create a codec_ring.yml decoders index file. The example above creates the following codec_ring.yml data which is a “YAML” file format for very rapid serialization of Ruby hash data structures.


```yaml
---
193: sensor_data
194: sensor_id
195: capability
196: T:V:R
197: version
198: "0.1"
199: alias
200: Base
201: time
202: temperature
203: unit
204: Fahrenheit
205: battery
206: Volts
207: rssi
208: dBm
209: Remote_One
210: Remote_Two
:"0.1": 198
:"T:V:R": 196
:Base: 200
:Fahrenheit: 204
:Remote_One: 209
:Remote_Two: 210
:Volts: 206
:alias: 199
:battery: 205
:capability: 195
:dBm: 208
:rssi: 207
:sensor_data: 193
:sensor_id: 194
:temperature: 202
:time: 201
:unit: 203
:version: 197
```


Listing 3.3 - Codec “ring” index file

For computing a compressed output from the `Viking_to_XML.rb` output the XML data is piped to the XML_Encoder.rb program by executing the following command line:

```
$ java listener | ./Viking_to_XML.rb | ./XML_Encoder.rb
```

This can be extended to a full end to end validation test by passing the XMLC data into the decoder process as follows:

```
$ java listener | ./Viking_to_XML.rb | ./XML_Encoder.rb | ./XML_Decoder.rb
```

The `XML_Encoder.rb` is just a wrapper script that performs a cursory record check, creates an encoder object, and then calls the "`to_XMLC`" function. Line1 is required to let the system know this file is a ruby executable. This script depends on the path to the xml_codec.rb which is established on the line 2. Line 4 continues to read in all lines from the STDIN. Line 5 checks to see if this record at least starts with an XML grammar (this could be expanded in the future to perform basic XML validation checks). Now that the line is known to be an XML format, an XML Codec object can be created from this standard input on line 6. Line 7 calls the "to_XMLC" method which is then output to the STDOUT.

```
1  #!/usr/bin/env ruby
2  require File.join(File.dirname(__FILE__), 'lib/xml_codec.rb' )
3
4  $stdin.each do |line|
5    if line =~ /\<\?xml/
6      xc = XMLCodec.new(line)
7      xmlc = xc.to_XMLC
8      puts xmlc
9    else
10     puts "NO ENCODABLE DOCUMENT FOUND"
11   end
12 end
```

Listing 3.4 - XML_Encoder.rb Wrapper Script

The `XML_Decoder.rb` is another wrapper script that performs a cursory record check to ensure that the input is NOT an XML document, creates a decoder object, and then calls the "to_XML" function which is the complementary function to the encoder. Line 7 calls the "to_XML" method that outputs XML to the STDOUT.

```
1  #!/usr/bin/env ruby
2  require File.join(File.dirname(__FILE__), 'lib/xml_codec.rb' )
3
4  $stdin.each do |line|
5    if line =~ /\<\?xml/
6      puts "NO DECODABLE DOCUMENT FOUND"
7    else
8      xd = XMLCodec.new(line)
9      xml = xd.to_XML
10     puts xml
11   end
12 end
```

Listing 3.5 - XML_Decoder.rb Wrapper Script

4. Conclusions

Additional tools have been created to continue smoothing the processing of self-describing wireless sensor data packets. Future work will include expanding the training capability for creating the codec_ring.yml file by analyzing several successive lines of input XML data which causes the system to store each unique element, attribute and attribute name into the codec ring forward and reverse index ("hash"). Although all of the above tools run well under MacOSX, there are still some small bugs that need to be worked out on the Hercules and CSSVM01 platforms that seem to be language version specific or other library include deficiencies. The full set of code is provided in Appendix A (Viking_to_XML) and Appendix B (xml_codec tools).

REFERENCES

[1] http://www.w3.org/TR/exi/

APPENDIX A – Viking_to_XML Code

```ruby
#!/usr/bin/env ruby
require File.join(File.dirname(__FILE__), 'lib/sensor_render' )
require File.join(File.dirname(__FILE__), 'lib/radio_record' )
#===============================================================================
#     Program: VikingX_to_XML.rb
# Description: Parses VikingX radio packets and translates into AgComm XML
#      Inputs: STDIN
#     Outputs: STDOUT
#      Author: Steve Dame (sdame@uw.edu)
#     Version: 0.1
#===============================================================================

# define an empty array of radio record hash
rr = {}

NAME = 0
TIME = 0
ID   = 1
ID2  = 2

# -----------------------------------------------------------------------------
# for each line of radio data, instantiate a new radio object if none previous
#   or parse data for an existing radio object.  The logic is that we need to
#   check to see if the first comma separated record is a time string.  If so,
#   then it is data for a radio that we either have in our in memory radio hash
#   record or hasn't yet been defined (in which case we need to discard rec)'
# -----------------------------------------------------------------------------
$stdin.each do |rec|

  begin

    # expecting a comma separated record unless it is XML or invalid format
    tmp = rec.to_s.split(',')
    item = []   # start with empty array of record items, "split and strip"
    tmp.each {|x| item << x.strip}
```

```
    # --------------------------------IF---------------------------------------
    # check for only one item in the comma separated parameters --may indicate
    #   an XML pass-through payload
    # -------------------------------------------------------------------------
    if (item.size == 1)
      # check for xml payload (TODO we should also check for well formed xml here)
      puts item[0] if item[0] =~ /\<\?xml/

    # ---------------------------------ELSIF-----------------------------------
    # check to see if first item is a time/date data record
    # This is an indicator that it is a non-XML radio sensor record
    # that will either be for an already instantiated radio object
    # or not an object yet (in which case the data will be tossed)
    # VALID SAMPLE FORMAT (prior to splitting above):
    #    "2011-10-27 12:16:22,60,0x227c3c,69.61,4.93,-58,"
    # -------------------------------------------------------------------------
    elsif item[TIME] =~ /\d{4}\-\d{2}\-\d{2} \d{2}:\d{2}:\d{2}/
      # extract the ID and convert to a symbol to use as key
      key = item[ID2].to_sym
      # just attempt to use the key, and rescue if no object exists yet
      # If RadioRecord does exist, then parse the data and print the XML
      begin
        rr[key].parse_data(rec)
#         rr[key].dump_sensors
        rr[key].to_xml
      rescue
        $stderr.puts "no Radio Record exists yet with key:[#{key}]"
      end
    # ---------------------------------ELSE------------------------------------
    # Create a new RadioRecord for any valid Radio Record format such as:
    # VALID SAMPLE FORMAT (prior to splitting above):
    #   "Remote_One, 0x227c6a, lat: 46.000000, lng: -119.000000, elv: 0.00"
    # -------------------------------------------------------------------------
    else
      # check to see that tmp[0] is just a valid name string
      if item[NAME] =~ /\w/
        # puts "->rec:" + item[NAME]
```

```
        # extract and validate (as 1-6 digit hex key) the ID
        #   and convert to a symbol to use as key
        # note: if RadioRecord exists already for this key, then a new one
        #   will be gracefully created at the same key position in hash rr
        if item[ID] =~ /0x[0-9a-f]{1,6}/
          key = item[ID].to_sym
          rr[key] = RadioRecord.new(rec)
        end
      end
    end
  # if rescue needed then must have been an invalid record format
  rescue
    $stderr.puts "non-parseable/ignored sensor record:[" + rec.strip + "]"
  end
end
```

A.1 - Viking_to_XML.rb top level script

```
require File.join(File.dirname(__FILE__), 'sensor_render' )
require 'date'

class RadioRecord
  # ------------------------------------------------------------------------
  # define CONSTANT (enum-like) indices for the Viking radio data record
  # ------------------------------------------------------------------------
  NAME = 0
  ID   = 1
  LAT  = 2
  LNG  = 3
  ELV  = 4
  TIME = 0
  NDX  = 1
  ID2  = 2
  TEMP = 3
  BATT = 4
  RSSI = 5
```

```
 POS_VALUE = 1
 POS_NAME  = 0

# ---------------------------------------------------------------------------
# @method initialize()
# @param rec
# @return symbol
# ---------------------------------------------------------------------------
def initialize(rec)
  # parse and clean each item within the Viking radio data record
  # to create a hash_map of the data
  # Each comma separated data record is in the form:
  # RadioName, ID, Loc_Lat_string, Loc_Lng_String, Loc_Elev_String
  # Example:
  #  Radio_One, 0x227c3c, lat: 46.000000, lng: -119.000000, elv: 0.00
  tmp = rec.to_s.split(',')
  item = []   # start with empty array of record items
  tmp.each {|x| item << x.strip}

  # initialize internal object record hashes
  @loc = {}
  @sensor = {}
  @info = {}

  @info[:name] = item[NAME]
  @info[:id]   = item[ID].to_sym

  @loc[:lat] = item[LAT].split(/: /)[POS_VALUE]
  @loc[:lng] = item[LNG].split(/: /)[POS_VALUE]
  @loc[:elv] = item[ELV].split(/: /)[POS_VALUE]
end

# ---------------------------------------------------------------------------
# @param name [String]
# ---------------------------------------------------------------------------
def name_exists(name)
  return( @name =~ name )
end
```

```
#
def id_exists(id)
  return( @id =~ id )
end

def get_id
  return(@info[:id])
end

def get_name
  return(@info[:name])
end

# ---------------------------------------------------------------------------
# @param data string
def parse_data (data)

# ---------------------------------------------------------------------------
# Sample VikingX Data
# ---------------------------------------------------------------------------
# DATE/TIME            NX,   ID    , TEMP, Volt, RSSI
# 2011-10-27 12:16:13,60,0x227bdb,68.58,12.17,-34,
# 2011-10-27 12:16:22,60,0x227c3c,69.61,4.93,-58,
# Remote_One, 0x227c6a, lat: 46.000000, lng: -119.000000, elv: 0.00
# 2011-10-27 12:17:09,60,0x227c6a,68.53,12.19,-28,
# Base, 0x227c3c, lat: 46.000000, lng: -119.000000, elv: 0.00
# Remote_Two, 0x227bdb, lat: 46.000000, lng: -119.000000, elv: 0.00
# 2011-10-27 12:17:13,60,0x227bdb,68.55,12.17,-30,
# 2011-10-27 12:17:22,60,0x227c3c,69.64,4.92,-54,
# Base, 0x227c3c, lat: 46.000000, lng: -119.000000, elv: 0.00
# Remote_One, 0x227c6a, lat: 46.000000, lng: -119.000000, elv: 0.00
# 2011-10-27 12:18:08,60,0x227c6a,68.50,12.19,-50,
# Remote_Two, 0x227bdb, lat: 46.000000, lng: -119.000000, elv: 0.00
# 2011-10-27 12:18:12,60,0x227bdb,68.32,12.17,-29,
# 2011-10-27 12:18:22,60,0x227c3c,69.64,4.93,-53,
# ---------------------------------------------------------------------------

```

```
    tmp = data.to_s.split(',')
    item = []   # start with empty array of record items
    tmp.each {|x| item << x.strip}

    @info[:time] = tmp[TIME]
    @dt = DateTime.parse(@info[:time])
    @info[:unix_time] = @dt.to_time.to_i

    # create a sensor object for the temperature and store it in the sensor hash
    s = Sensor.new("temperature","Fahrenheit",tmp[TEMP])
    @sensor[:temp] = s

    # create a sensor object for the battery voltage and store n -the sensor hash
    v = Sensor.new("battery","Volts",tmp[BATT])
    @sensor[:battery] = v

    # create a sensor object for the radio RSSI and store it in the sensor hash
    r = Sensor.new("rssi","dBm",tmp[RSSI])
    @sensor[:rssi] = r

    @my_XML = SensorRender.new(@info[:unix_time])

    root = @my_XML.addRootElement("sensor_data")
    @my_XML.addSubElement_Value( root, "sensor_id", @info[:id].to_s)
    @my_XML.addAttribute("alias",  @info[:name])
    @my_XML.addAttribute("capability", "T:V:R" )
    @my_XML.addAttribute("version", "0.1" )
    @my_XML.addSubElement_Value( root, "time", @info[:unix_time].to_s )

#     @sensor.each {|k,v| p v.name}

    @sensor.each do |k,v|
        @my_XML.addSubElement_Value(root,v.name, v.value)
        @my_XML.addAttribute("unit",v.unit)
    end
  end
```

```
# dump the XML output
def to_xml
    puts @my_XML.getXML
end

def dump_sensors
  @sensor.each {|s| p s}
end
end
```

A.2 - radio_record.rb helper class library

```ruby
#!/usr/bin/ruby -w
require File.join(File.dirname(__FILE__), 'sensor' )
require "rexml/document"
include REXML

#==============================================================================
#        Class: SensorRender
# Description: Build XML packets from sensor data
#       Inputs:
#      Returns:
#==============================================================================
class SensorRender

  # @param time [Object]
  def initialize(time)
    root_element = %{sensor_data}
    @xml_doc = REXML::Document.new
    @xml_doc << XMLDecl.new( "1.0", "UTF-8")

    # if time is not passed in then just initialize to current time
    if time == nil
      @current_time = Time.now.to_i
    else
      @current_time = time
    end
    @cur_element = @xml_doc
    @root_element = nil
  end

  #============================================================================
  #       Method: writeSensorXML_ToFile
  # Description: Write XML tree formated sensor data to a filename
  #       Inputs: filename - file to store data
  #      Returns:
  #============================================================================
  def writeSensorXML_ToFile(filename)
```

```ruby
#    @xml_doc.write ($stdout, 0)
    puts @xml_doc
  end

    #==============================================================================
    #       Method: getXML
    # Description:
    #       Inputs:
    #      Returns:
    #==============================================================================
    def getXML
      return @xml_doc
    end


  #==============================================================================
  #       Method: addRootElement
  # Description:
  #       Inputs:
  #      Returns:
  #==============================================================================
  def addRootElement(root_name)
    @cur_element = @cur_element.add_element(root_name)
    return @cur_element
  end


    #==============================================================================
    #       Method: addElement
    # Description:
    #       Inputs:
    #      Returns:
    #==============================================================================
    def addElement(element)
      @cur_element = @cur_element.add_element(element)
      return @cur_element
    end

```

```ruby
#=============================================================================
#      Method: addSubElement
# Description:
#      Inputs:
#     Returns:
#=============================================================================
def addSubElement(parent, element)
  @cur_element = parent.add_element(element)
  return @cur_element
end

#=============================================================================
#      Method: addSubElement_Value
# Description:
#      Inputs:
#     Returns:
#=============================================================================
def addSubElement_Value(parent, element, value)
  @cur_element = parent.add_element(element)
  @cur_element.add_text(value)
  return @cur_element
end

#=============================================================================
#      Method: addElement_Value
# Description:
#      Inputs:
#     Returns:
#=============================================================================
def addElement_Value(element, value)
  @cur_element = @cur_element.add_element(element)
  @cur_element.add_text(value)
  return @cur_element
end

#=============================================================================
#      Method: addValue
# Description:
```

```ruby
  #       Inputs:
  #      Returns:
  #==========================================================================
  def addValue(value)
    @cur_element.add_text(value)
    return @cur_element
  end

  #==========================================================================
  #       Method: addAttribute
  # Description:
  #       Inputs:
  #      Returns:
  #==========================================================================
  def addAttribute(key, value)
    @cur_element.add_attribute(key,value)
    return @cur_element
  end
end
```

A.3 - sensor_render.rb helper class library

```ruby
class Sensor
  def initialize(name, unit, value)
    @name = name
    @unit = unit
    @value = value
  end

  def name
    return(@name)
  end

  def unit
    return(@unit)
  end

  def value
    return(@value)
  end
end
```

A.4 - sensor.rb helper class library

APPENDIX B – XML_CODEC Code

```
#!/usr/bin/ruby -w
require 'yaml'
require "rexml/document"
include REXML

#==============================================================================
#       Class: XMLCodec
# Description: XML Encoder / Decoder Class library
#      Inputs:
#     Returns:
#==============================================================================
class XMLCodec

  EOF = 0xff                  # end of file

  def initialize  (rec)
    # ------------------------- BASE64 Symbol table -------------------------
    # create the encoder symbols from the base64 table
    @base64_char = []
    i = 0..25
    i.each {|c| @base64_char << (c+0x41).chr}
    i = 0..25
    i.each {|c| @base64_char << (c+0x61).chr}
    i = 0..9
    i.each {|c| @base64_char << (c+0x30).chr}
    @base64_char << '+'
    @base64_char << '/'
    # -----------------------------------------------------------------------

    # -------------------------- Decoder Ring -------------------------------
    # load default decoder ring
    @ring = begin
      YAML.load(File.open("codec_ring.yml"))
    rescue ArgumentError => e
```

```
      puts "Could not parse YAML: #{e.message}"
    end
    # -------------------------------------------------------------------------

    # --------------------------- XML or XMLC  --------------------------------
    # initialize differently depending on whether input is XML or XMLC
    if rec =~ /\<\?xml/
      @xml_doc = Document.new(rec)
    else
      @xmlc_doc = rec
      @xml_doc = Document.new
      if rec[0] == 0x01
        @xml_doc << XMLDecl.new( "1.0", "UTF-8")
        @root = @xml_doc
        root_token = rec[1]
        begin
          root_name = @ring[root_token]
          @root = @xml_doc.add_element(root_name)
        rescue
          puts "FATAL: Bad Root Token lookup in codec ring"
        end
      else
        puts "FATAL: Unrecognized XML format or version."
      end
    end
    # -------------------------------------------------------------------------
  end

  #============================================================================
  #      Method: to_XML
  # Description: decode XMLC document to XML document output
  #      Inputs: @xmlc_doc
  #     Returns: @xml_doc
  #============================================================================
  def to_XML

    xmlc_len = @xmlc_doc.length
    state = :state_idle
```

```
#    2.upto(xmlc_len-1) { |i| printf("0x%0.2X - %c\n",@xmlc_doc[i], @xmlc_doc[i])}

    value = ""
    element_token = ""
    attr_token = ""
    attr_value = ""
    attr_name = ""

    # rip through all of the characters one by one
    2.upto(xmlc_len-1)  do |i|
      cur_char = @xmlc_doc[i]
      case state
        # ------------------------------- IDLE --------------------------------
        when :state_idle:
          # from idle we can only add an element and there can only be tokens
          if ((cur_char & 0x80) == 0x80)
            state = :state_build_element
            element_token = cur_char
            @cur_element = @root.add_element(@ring[element_token])
          end
        # ------------------------- BUILD ELEMENT STATE -----------------------
        when :state_build_element:
          if ((cur_char & 0x80) == 0x80)
            # no attributes, just output the element value (i.e. "text")
            if(element_token == cur_char)
              value = ""
              state = :state_build_element_value
            else
              attr_token = cur_char
              attr_name = @ring[attr_token]
              state = :state_build_attribute_value
            end
          end

        # ----------------------- BUILD ATTRIBUTE NAME -----------------------
        when :state_build_attribute_name:
          if ((cur_char & 0x80) == 0x80)
```

```
          # ----------------------------------------------------------------
          # if element_token, then time to complete this element by adding
          #   the attribute that was in process and the element text
          # ----------------------------------------------------------------
          if(element_token == cur_char)
            value = ""
            state = :state_build_element_value
          # ----------------------------------------------------------------
          # otherwise this must be the next attribute, which means we need to
          #   add the current completed attribute, and stay in the
          #   build_attribute state and reset to the next value
          # ----------------------------------------------------------------
          else
            attr_token = cur_char
            attr_name = @ring[attr_token]
            state = :state_build_attribute_value     # get value and complete
          end
        end
      # ----------------------- BUILD ATTRIBUTE VALUE -----------------------
      when :state_build_attribute_value:
        if ((cur_char & 0x80) == 0x80)
            attr_token = cur_char
            attr_value = @ring[attr_token]
            @cur_element.add_attribute(attr_name, attr_value)
            value = ""
            state = :state_build_attribute_name     # check for next attribute
        else
          puts "state_build_attribute_value: decoder error!"
        end
      # ----------------------- BUILD ELEMENT VALUE -------------------------
      when :state_build_element_value:
        if ((cur_char & 0x80) == 0x80)
            @cur_element.add_text(value)

            # Is this the next element or end of document
            if(cur_char == EOF)
              return @xml_doc
            else
```

```
                element_token = cur_char
                @cur_element = @root.add_element(@ring[element_token])
                value = ""
                state = :state_build_element
              end
          else
            value << cur_char
          end
        # ------------------------------ ERROR --------------------------------
        else  puts "illegal decoder state"
      end
    end

    return @xml_doc
  end

  #=============================================================================
  #       Method: to_XMLC
  # Description: convert xml_doc to compressed XML output
  #       Inputs:
  #      Returns:
  #=============================================================================
  def to_XMLC

    @xmlc_doc = ""              # clear the XMLC output string
    @xmlc_doc << 0x01           # first char is always the XML revision
    root = @xml_doc.root        # xml doc root always gets the first symbol

    begin
      enc_char = @ring[root.name.to_sym]
      @xmlc_doc << enc_char       # second char is always the root element token
      # ---------------------------------------------------------------------
      # parse through each element and each attribute of each element
      #  to create the compressed tokenized string output
      root.each do |e|
        elem_token = @ring[e.name.to_sym]
        @xmlc_doc << elem_token          # encode start with element token
        e.attributes.each do |attr_name,attr_value|
```

```ruby
            attr_token = @ring[attr_name.to_sym]
            @xmlc_doc << attr_token
            attr_token = @ring[attr_value.to_sym]
            @xmlc_doc << attr_token
        end
        @xmlc_doc << elem_token         # must delimit the values by the element
        @xmlc_doc << e.text
      end
      @xmlc_doc << EOF
      # -------------------------------------------------------------------------
    rescue ArgumentError => e
      puts "Problem with codec_ring #{e.message}"
    end
    return @xmlc_doc.to_s
  end

  #===========================================================================
  #       Method: create_ring
  # Description: create decoder ring from XML record  (erases previous ring)
  #       Inputs: @xml_doc - xml document to base new codec ring upon
  #      Outputs: @ring - refreshed new codec ring
  #===========================================================================
  def create_ring
    @ring = {}                      # clear the old codec_ring
    xml_keys = @base64_char         # get the symbol key lookup table

    root = @xml_doc.root            # xml doc root always gets the first symbol
    token = xml_keys.shift[0]+0x80
    @ring[token] = root.name
    @ring[root.name.to_sym] = token
    # -------------------------------------------------------------------------
    # parse through each element and each attribute of each element
    #  to create the "forward index" --> decoder
    #  and at the same time create the "reverse index"  i.e. --> encoder
    # -------------------------------------------------------------------------
    root.each do |e|
      token = xml_keys.shift[0]+0x80          # create the next token
      @ring[token] = e.name                   # store the element name
```

```
      @ring[e.name.to_sym] = token          # index its token
      e.attributes.each do |attr_name,attr_value|
        # ---------------------------------------------------------------------
        # only hash new attribute token if not already stored
        if(nil == @ring[attr_name.to_sym])
          token = xml_keys.shift[0]+0x80    # create the next token
          @ring[token] = attr_name          # store the name of the attr
          @ring[attr_name.to_sym] = token   # index its token
        end
        # ---------------------------------------------------------------------
        # only hash new attribute value if not already stored
        if(nil == @ring[attr_value.to_sym])
          token = xml_keys.shift[0]+0x80    # create the next token
          @ring[token] = attr_value         # store the name of the attr
          @ring[attr_value.to_sym] = token  # index its token
        end
        # ---------------------------------------------------------------------
      end
    end
    return @ring
  end

  #=============================================================================
  #       Method: append_ring
  # Description: append decoder ring from XML record  (adds to previous ring)
  #       Inputs:
  #      Returns:
  #=============================================================================
  def append_ring
    # note: add a future capability to update the codec ring
  end

  #=============================================================================
  #       Method: load_ring
  # Description: load decoder ring from YAML file
  #       Inputs:
  #      Returns:
  #=============================================================================
```

```
def load_ring (file)

@ring = begin
    YAML.load(File.open(file))
  rescue ArgumentError => e
    puts "Could not parse YAML: #{e.message}"
  end
end

#============================================================================
#      Method: save_ring
# Description: save decoder ring to YAML file
#      Inputs:
#     Returns:
#============================================================================
def save_ring (file)
  begin
    File.open(file, "w") {|f| f.write(@ring.to_yaml) }
  rescue ArgumentError => e
    puts "Could not parse YAML: #{e.message}"
  end
end

#============================================================================
#      Method: get_ring
# Description: return the encoder / decoder ring
#      Inputs:
#     Returns: @ring - codec ring
#============================================================================
def get_ring
  return @ring
end
end
```

B.1 - xml_codec.rb