Applied RFID Technology

Josh Gallegos

Revision 12/11/2008

Professor Zalewski

CNT4104

1. Introduction to RFID Technology

1.1 General Overview

RFID functionality is an underutilized, powerful technology with limitless possibilities: It is a device which can sniff out electric tags in the immediate area, which have unique information associated with them. If ten tags are in the area, then they can all be noticed and logged. This has countless applications in commercial and military environments, allowing inventory, timekeeping and accountability for a number of objects. It can be put in cell phones to track employees through buildings, can be placed on animals to locate subjects immediately by unique identifiers , also eliminating archaic branding methods[1], or can make barcode scanning of hundreds of items go from an hour long project to a second. Also, unlike barcodes, it can be done at a distance and from angles, circumventing frustrating operational parameters required for barcode scanners[4].

For programming applications, RFID simply outputs a small bit of information when its sensor activates a hit. A common way to do this is a gateway interface. As a tag (See Figure 1 below) passes through a conveyor belt, for example, a tag inside it can identify it and logs the instance with a timestamp.

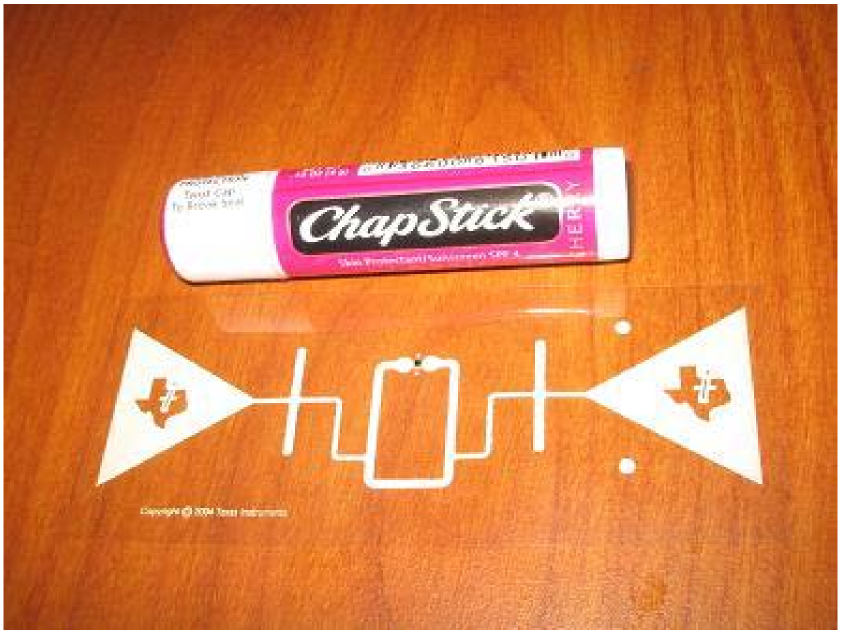


Figure 1: An RFID tag. The ChapStick is for size comparison.

This tells manufacturers that the tag has passed through a certain area, giving its location, time and a confirmation that it was sent out. If, for example, a packaging company wanted to confirm that a certain box was sent out, they could review their logs and see that indeed, it passed a conveyor belt or doorway to a truck at 10PM, or whatever the log records, as pictured in Figure 2 below. With very little investment, and with much better efficiency than manually walking around and noting the contents of the room or using a bar code scanner, the room has been inventoried effectively. This is the power of RFID.

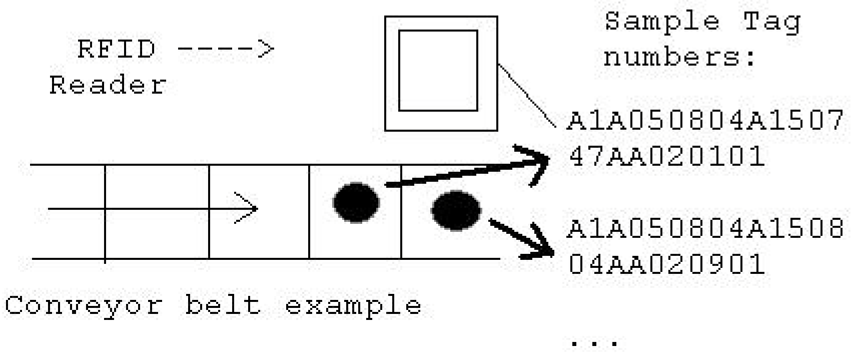
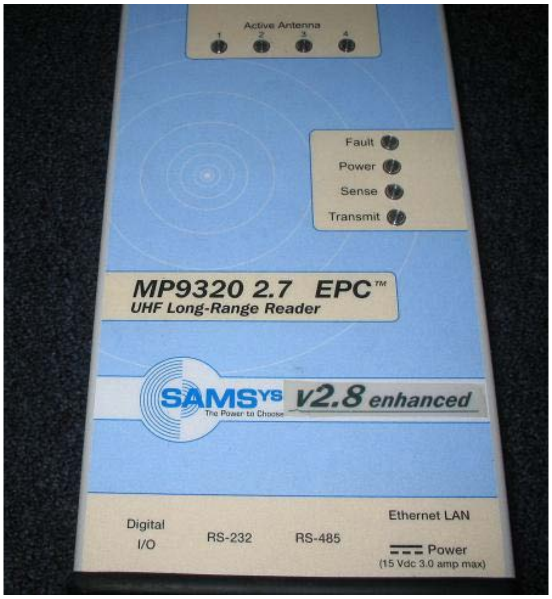


Figure 2: Example of an RFID application.

1.2 Device Physical Description

The device is illustrated in Figures 3-7. The base is a box with a variety of lights (Figure 3) and connection plugs (Figures 4, 5). The back has four plugs (Figure 6) allowing four different sensors, three of which must be plugged with resistors for proper operation (Although more are supported, for this simple example application, one does just fine). The open one is connected to a plate sensor, which is square shaped and does not appear as one might imagine, as indicated in Figure 7.



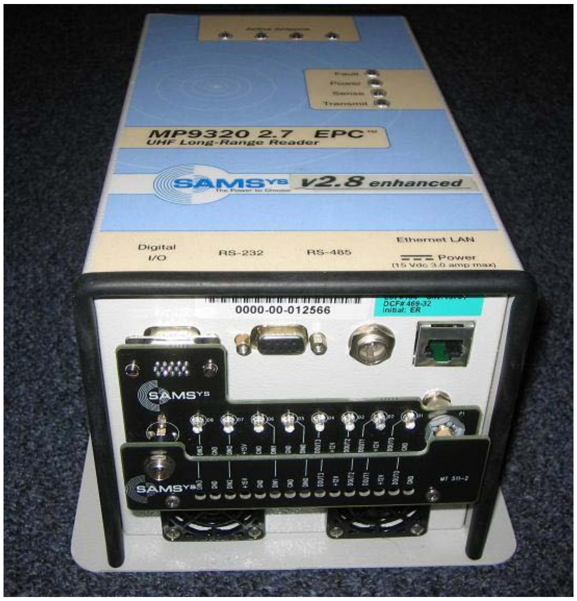
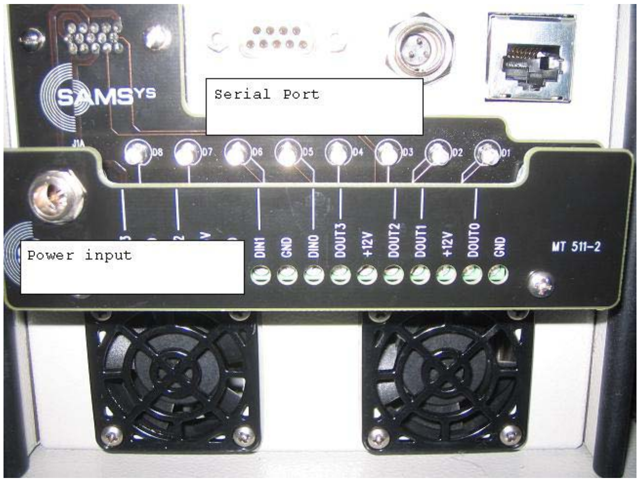


Figure 3 (left): Top view of RFID box, includes lights for operational status.

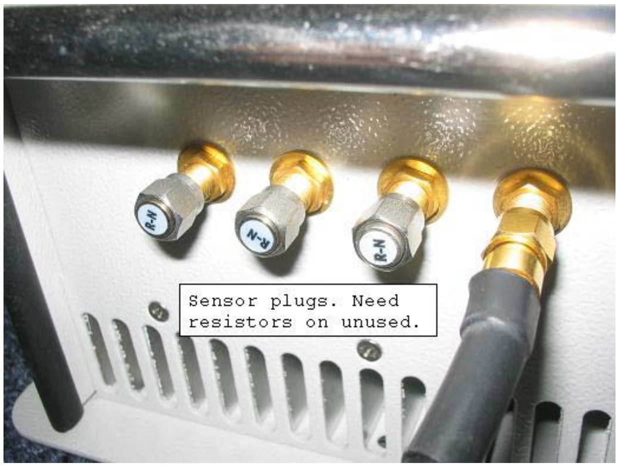
Figure 4 (right): The front of an RFID box



Serial Port

Power input

Figure 5: Close-up of the front of the RFID box. Two plugs are utilized as indicated.



Sensor plugs. Need resistors on unused.

Figure 6: Back of the RFID box, includes ports for sensors.

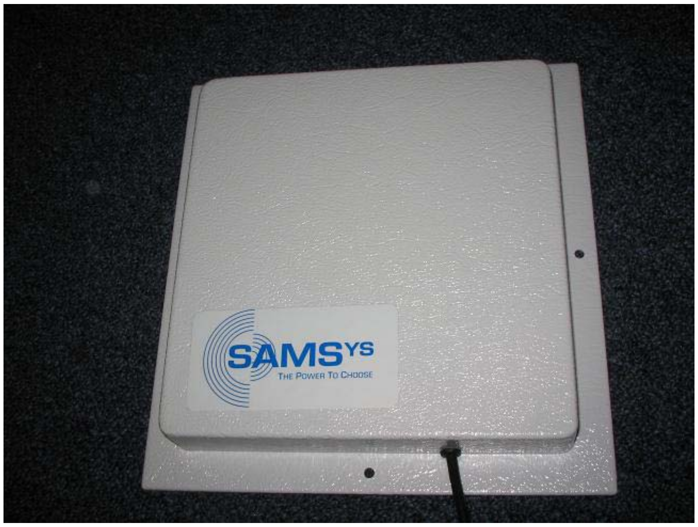


Figure 7: RFID Sensor/Reader.

The box is then connected to a computer via a serial cable, as visible in Figure 5 and diagramed in Figure 8. In this case I have used a serial-to-USB cable since I only have a laptop, which lack serial connectivity. (For any future students doing this project with similar circumstances, see references item

3 for a link to the serial-to-USB driver I located). A power supply also must be plugged in. The wiring setup is shown schematically in Figure 8.

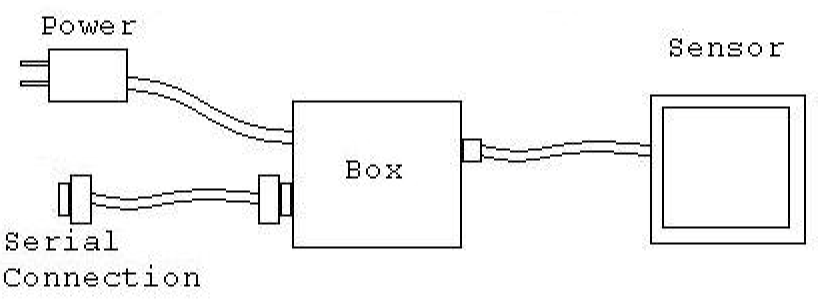
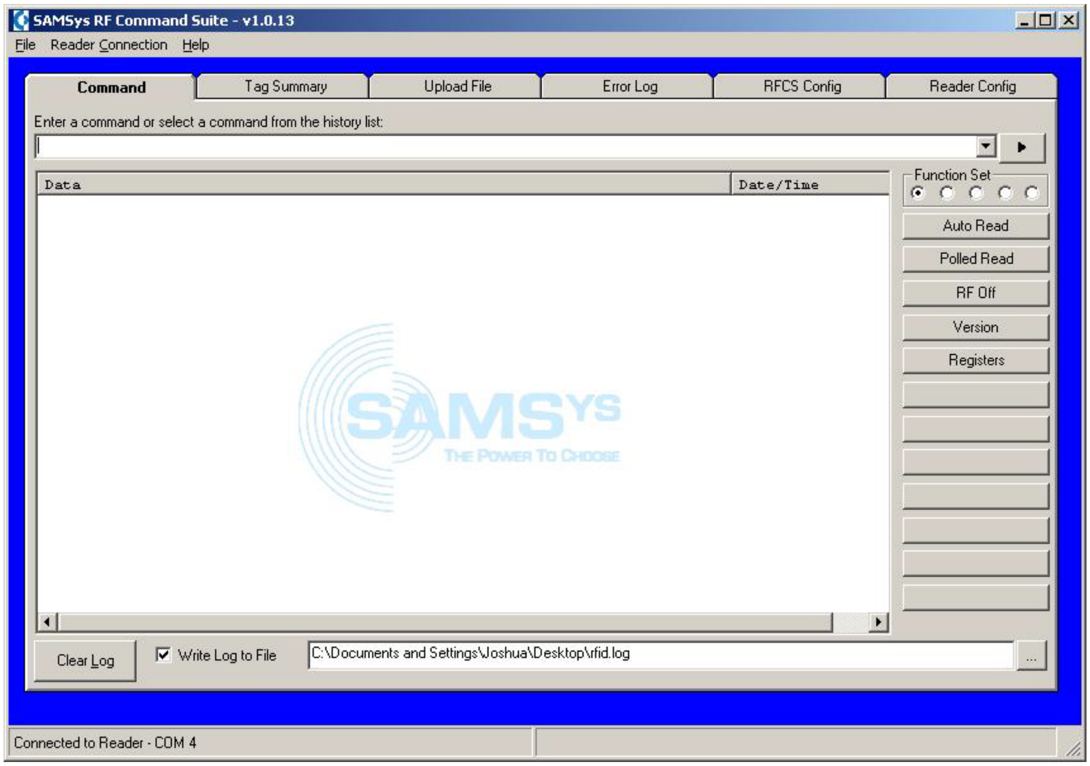


Figure 8: Simple wiring schematic. It needs power, a serial cable and a sensor.

Most RFID tags contain at least two components. One is just an integrated circuit for saving and processing information. The second component is an antenna for signal transmissions [2].

1.3 Device Operation

After having it all connected, software provided by the manufacturer called the "RF Command Suite" allows box-to-computer communication. If the box is turned on and running correctly, without the "Fault" light lit in red (My only problem in this stage was that I did not have a resistor plugged into the front in port 2, which absolutely must be done for proper operation), then once the software is run from the folder "\MP9320\RF Command Suite\Setup_RFCommandSuite_1_0_13g.msi", it should be completely ready to go. A small indication is on the bottom of the screen indicating that a connection was successful, as shown in Figure 9.



Enter a command or select a command from the history list:

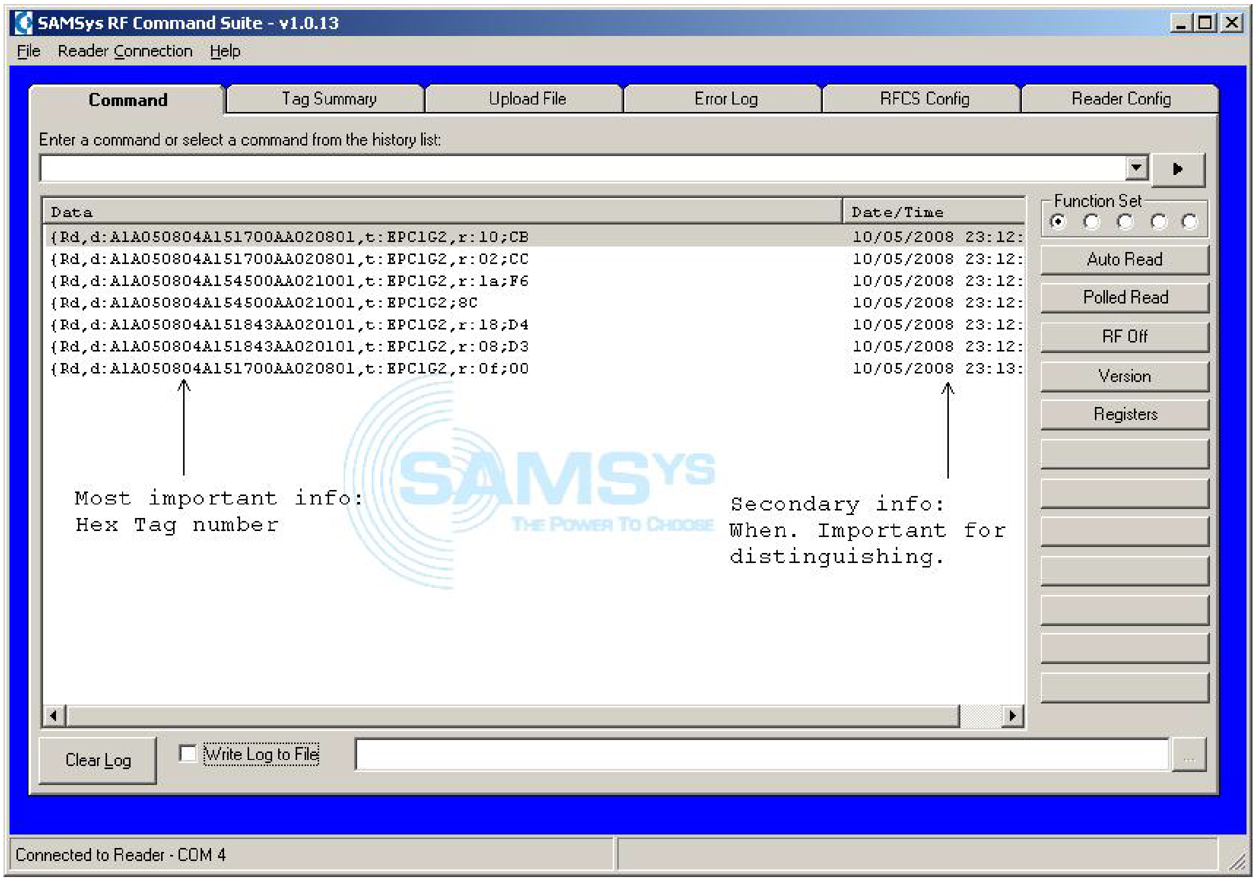
☑ Write Log to File C:\Documents and Settings\Joshua\Desktop\rfid.log

Connected to Reader - COM 4

Figure 9: The program without any sensor output.

Although I did not have any problem with it, it is notable that problems may exist with the port number used for the program. The ports, of course, must be matching between the serial port and what the program is looking for. This is done easily by accessing the hardware manager and going to ports. Here, you can find your serial port and modify the number. This is applicable for those using the USB-to-serial conversion wire, and it still appears under the “ports” listing. In the RF Command Suite, access menu “Reader Connection>>Serial Port Settings”. Simply match the port numbers and the box should operate correctly.

Next, we must confirm the operation of the RFID box. With it running correctly, this is simply done by waving one of the provided RFID tags in front of the sensor. It should beep softly as it registers the tag being there. An item appears on the program, indicating a time and a tag identifier. This can be seen in Figure 10, where I waved just three tags in front of the reader. Notice how it registered each one twice (the last one item was to prove that the first one would still register as the same tag). This is because it registers quite frequently as an item passes. If, for example, a conveyor belt was moving slowly, it is imaginable that an item may register 5 or 6 times. This is a slight problem, but can easily be countered with logical coding: duplicate results will exist. A savvy programmer could simply take one item and eliminate duplicates that exist within, say, 5 seconds. This gets us past the duplicate item problem quite easily.



Enter a command or select a command from the history list:

```
Data                                              Date/Time
{Rd,d:A1A050804A151700AA020801,t:EPC1G2,r:10;CB    10/05/2008 23:12:
{Rd,d:A1A050804A151700AA020801,t:EPC1G2,r:02;CC    10/05/2008 23:12:
{Rd,d:A1A050804A154500AA021001,t:EPC1G2,r:1a;F6    10/05/2008 23:12:
{Rd,d:A1A050804A154500AA021001,t:EPC1G2;8C         10/05/2008 23:12:
{Rd,d:A1A050804A151843AA020101,t:EPC1G2,r:18;D4    10/05/2008 23:12:
{Rd,d:A1A050804A151843AA020101,t:EPC1G2,r:08;D3    10/05/2008 23:12:
{Rd,d:A1A050804A151700AA020801,t:EPC1G2,r:0f;00    10/05/2008 23:13:
```

Most important info:
Hex Tag number

Secondary info:
When. Important for
distinguishing.

Connected to Reader - COM 4

Figure 10: Program with some sensor output displayed on screen.

Two pieces of useful data can be derived from this. The main item is the hexadecimal representation of the tag number. This provides us with unique data on each tag. In the conveyor belt example, a package with one of these tags on it can be easily differentiated from the other ones. Many tag numbers exist too, allowing virtually as many combinations as one could possibly ever need. Quick arithmetic supports this conclusion: Each hexadecimal digit contains 4 bits of data, or 2^4 possibilities. There are 24 hex digits available. This allows 2^{96} bit combinations, or 7.923 x 10^{28} possibilities! This number is quite a bit larger than would ever be necessary to maintain a proper unique identifier for a package.

The second piece of information provides us with a “when” to the question of the tag’s whereabouts. Simply knowing that it passed through a certain location is not always good enough. The timestamp provides us with a very basic tracking ability, letting us know exactly when it arrived, departed or passed through a gateway, or however the purpose may be. What does this give us?

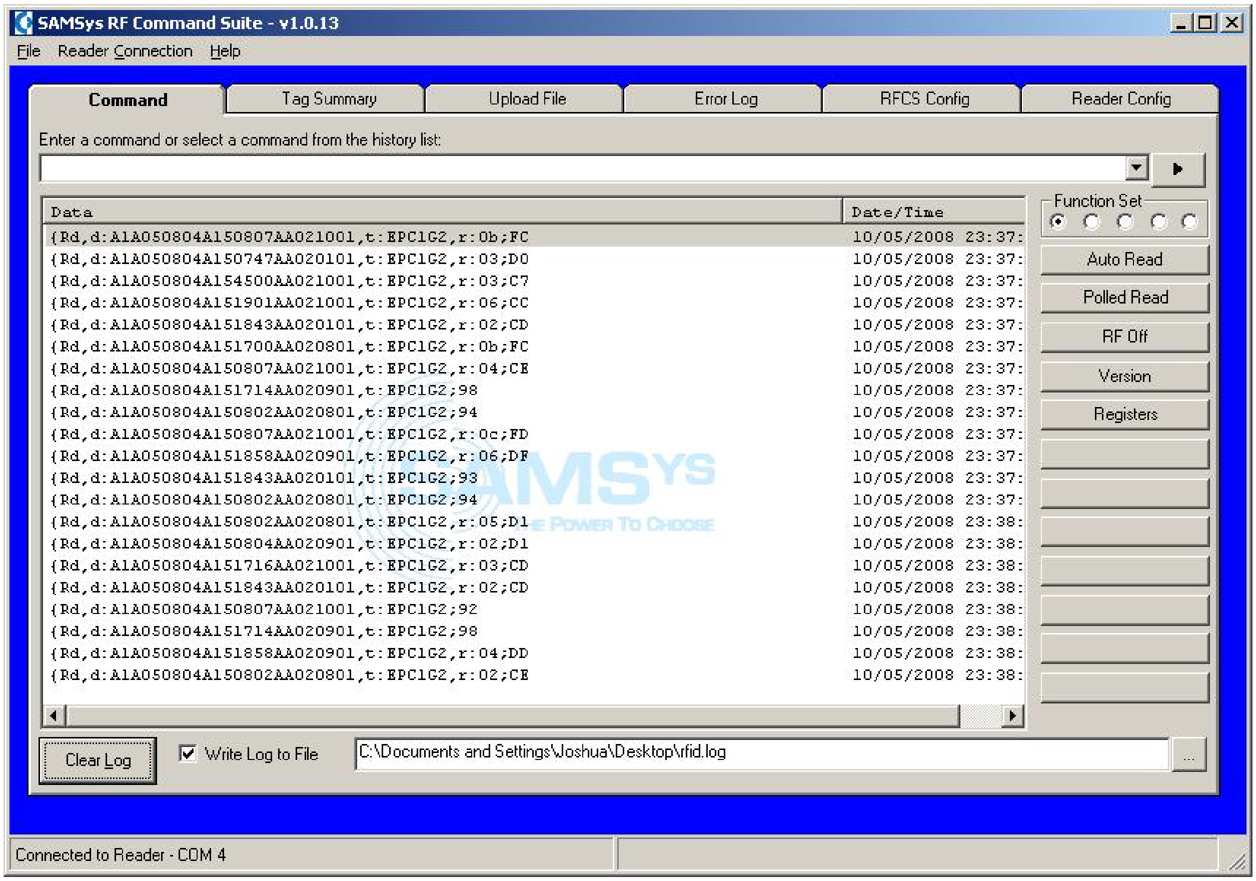
- Proof of entry. This provides evidence that a sought-after item has entered a building. Knowing this can save a company from having to look for something if it never entered in the first place!
- Proof of departure. A packaging company, for example, can show that a certain item left the building at a certain time.
- Productivity. When did it enter a room? When did it exit? An item took X minutes to complete a certain production stage. This valuable information can provide feedback to a company.

We now have three problems to work out:

1. The data must be retrieved at an unknown point of time.
2. The data must always be ready to access.
3. The result must be output from the program into useful data.

These problems are all very easily solved by the RF Command Suite. By simply leaving it on all the time on the server side, it constantly collects data, scanning continuously for tags. Therefore, as long as the server is properly working, then it can be accessed at any unknown point in time.

The RF Command Suite also provides a simple output log in plain text. This is done by simply clicking on the “Write Log to File” checkbox, and indicating an output file. The following, Figure 11, demonstrates the output of 11 tags being waved in a handful in front of the RFID box, for testing purposes. Note that the log button has been checked, and a random file indicated for output.



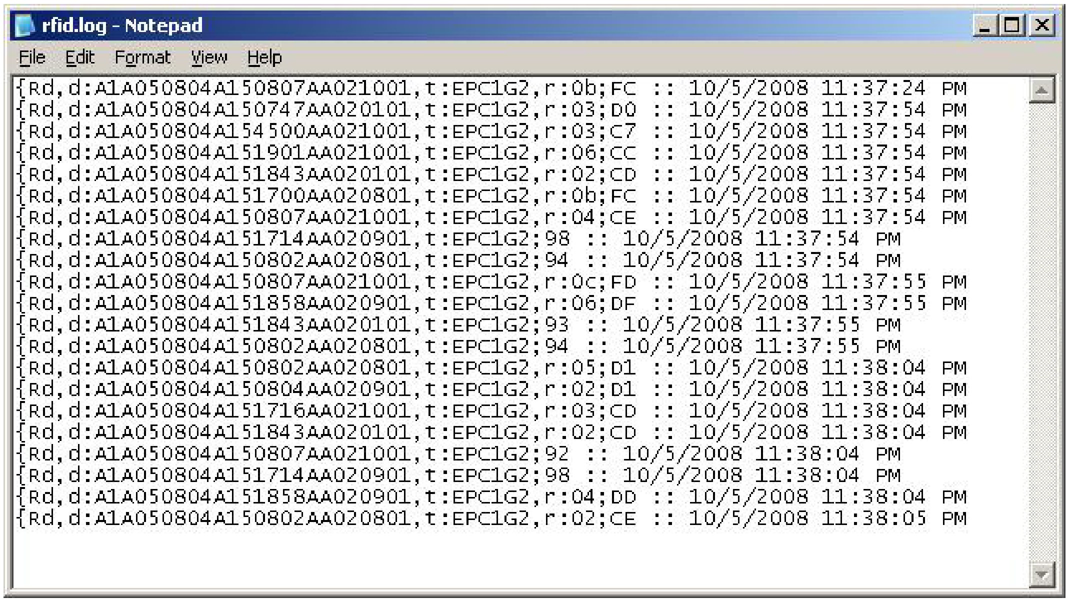
Enter a command or select a command from the history list:

```
Data                                                Date/Time
{Rd,d:A1A050804A150807AA021001,t:EPC1G2,r:0b;FC     10/05/2008 23:37:
{Rd,d:A1A050804A150747AA020101,t:EPC1G2,r:03;D0     10/05/2008 23:37:
{Rd,d:A1A050804A154500AA021001,t:EPC1G2,r:03;C7     10/05/2008 23:37:
{Rd,d:A1A050804A151901AA021001,t:EPC1G2,r:06;CC     10/05/2008 23:37:
{Rd,d:A1A050804A151843AA020101,t:EPC1G2,r:02;CD     10/05/2008 23:37:
{Rd,d:A1A050804A151700AA020801,t:EPC1G2,r:0b;FC     10/05/2008 23:37:
{Rd,d:A1A050804A150807AA021001,t:EPC1G2,r:04;CE     10/05/2008 23:37:
{Rd,d:A1A050804A151714AA020901,t:EPC1G2;98          10/05/2008 23:37:
{Rd,d:A1A050804A150802AA020801,t:EPC1G2;94          10/05/2008 23:37:
{Rd,d:A1A050804A150807AA021001,t:EPC1G2,r:0c;FD     10/05/2008 23:37:
{Rd,d:A1A050804A151858AA020901,t:EPC1G2,r:06;DF     10/05/2008 23:37:
{Rd,d:A1A050804A151843AA020101,t:EPC1G2;93          10/05/2008 23:37:
{Rd,d:A1A050804A150802AA020801,t:EPC1G2;94          10/05/2008 23:37:
{Rd,d:A1A050804A150802AA020801,t:EPC1G2,r:05;D1     10/05/2008 23:38:
{Rd,d:A1A050804A150804AA020901,t:EPC1G2,r:02;D1     10/05/2008 23:38:
{Rd,d:A1A050804A151716AA021001,t:EPC1G2,r:03;CD     10/05/2008 23:38:
{Rd,d:A1A050804A151843AA020101,t:EPC1G2,r:02;CD     10/05/2008 23:38:
{Rd,d:A1A050804A150807AA021001,t:EPC1G2;92          10/05/2008 23:38:
{Rd,d:A1A050804A151714AA020901,t:EPC1G2;98          10/05/2008 23:38:
{Rd,d:A1A050804A151858AA020901,t:EPC1G2,r:04;DD     10/05/2008 23:38:
{Rd,d:A1A050804A150802AA020801,t:EPC1G2,r:02;CE     10/05/2008 23:38:
```

☑ Write Log to File: C:\Documents and Settings\Joshua\Desktop\rfid.log

Connected to Reader - COM 4

Figure 11: Program with plenty of output, and a log file started.

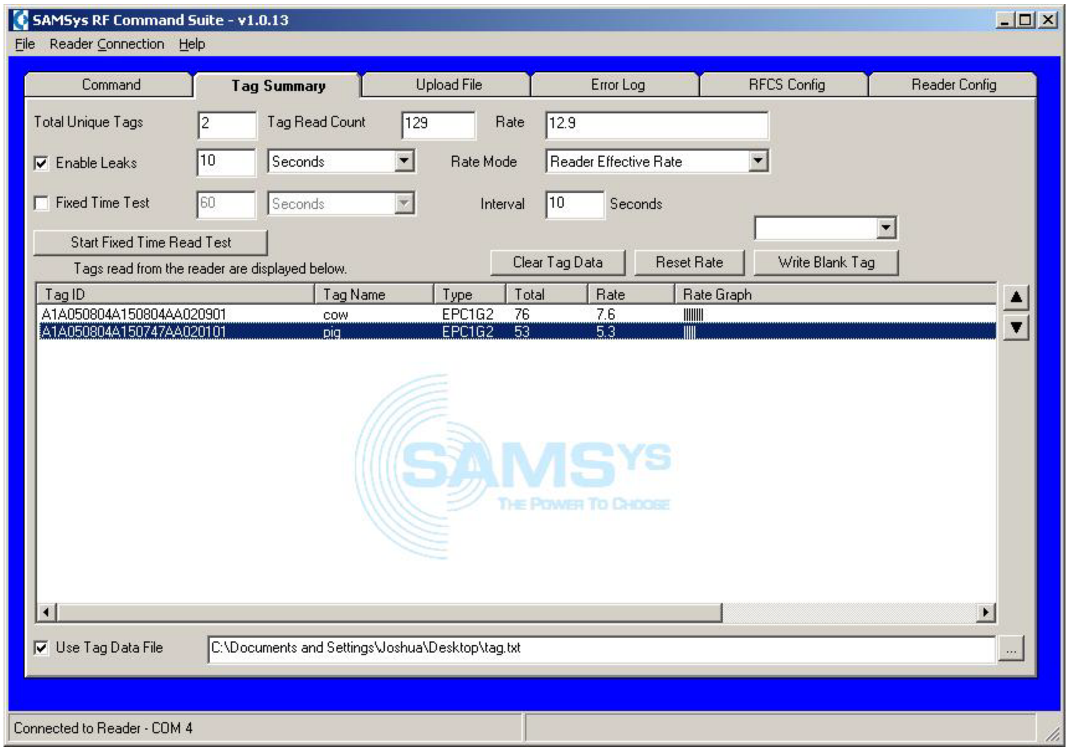


```
{Rd,d:A1A050804A150807AA021001,t:EPC1G2,r:0b;FC :: 10/5/2008 11:37:24 PM
{Rd,d:A1A050804A150747AA020101,t:EPC1G2,r:03;D0 :: 10/5/2008 11:37:54 PM
{Rd,d:A1A050804A154500AA021001,t:EPC1G2,r:03;C7 :: 10/5/2008 11:37:54 PM
{Rd,d:A1A050804A151901AA021001,t:EPC1G2,r:06;CC :: 10/5/2008 11:37:54 PM
{Rd,d:A1A050804A151843AA020101,t:EPC1G2,r:02;CD :: 10/5/2008 11:37:54 PM
{Rd,d:A1A050804A151700AA020801,t:EPC1G2,r:0b;FC :: 10/5/2008 11:37:54 PM
{Rd,d:A1A050804A150807AA021001,t:EPC1G2,r:04;CE :: 10/5/2008 11:37:54 PM
{Rd,d:A1A050804A151714AA020901,t:EPC1G2;98 :: 10/5/2008 11:37:54 PM
{Rd,d:A1A050804A150802AA020801,t:EPC1G2;94 :: 10/5/2008 11:37:54 PM
{Rd,d:A1A050804A150807AA021001,t:EPC1G2,r:0c;FD :: 10/5/2008 11:37:55 PM
{Rd,d:A1A050804A151858AA020901,t:EPC1G2,r:06;DF :: 10/5/2008 11:37:55 PM
{Rd,d:A1A050804A151843AA020101,t:EPC1G2;93 :: 10/5/2008 11:37:55 PM
{Rd,d:A1A050804A150802AA020801,t:EPC1G2;94 :: 10/5/2008 11:37:55 PM
{Rd,d:A1A050804A150802AA020801,t:EPC1G2,r:05;D1 :: 10/5/2008 11:38:04 PM
{Rd,d:A1A050804A150804AA020901,t:EPC1G2,r:02;D1 :: 10/5/2008 11:38:04 PM
{Rd,d:A1A050804A151716AA021001,t:EPC1G2,r:03;CD :: 10/5/2008 11:38:04 PM
{Rd,d:A1A050804A151843AA020101,t:EPC1G2,r:02;CD :: 10/5/2008 11:38:04 PM
{Rd,d:A1A050804A150807AA021001,t:EPC1G2;92 :: 10/5/2008 11:38:04 PM
{Rd,d:A1A050804A151714AA020901,t:EPC1G2;98 :: 10/5/2008 11:38:04 PM
{Rd,d:A1A050804A151858AA020901,t:EPC1G2,r:04;DD :: 10/5/2008 11:38:04 PM
{Rd,d:A1A050804A150802AA020801,t:EPC1G2,r:02;CE :: 10/5/2008 11:38:05 PM
```

Figure 12: Log file output from program. This is plain text.

A brief glimpse at the log in Figure 11 shows it is output pretty much the same output as it's listed in Figure 12.

The third problem can be addressed using another feature of the program. Figure 13 demonstrates the screen output for the "Tag Summary" tab.



Total Unique Tags: 2
Tag Read Count: 129
Rate: 12.9

☑ Enable Leaks: 10 Seconds
Rate Mode: Reader Effective Rate

☐ Fixed Time Test: 60 Seconds
Interval: 10 Seconds

Tags read from the reader are displayed below.

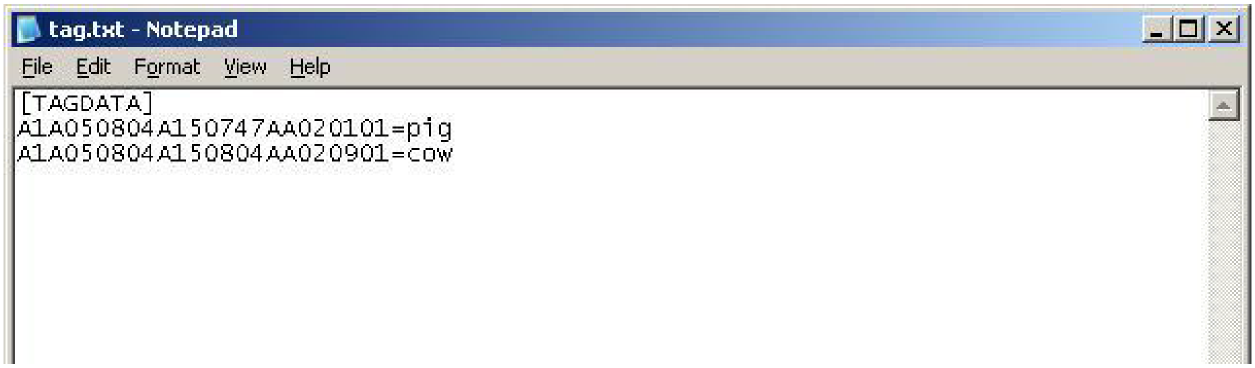
Tag ID	Tag Name	Type	Total	Rate	Rate Graph
A1A050804A150804AA020901	cow	EPC1G2	76	7.6	||||||||
A1A050804A150747AA020101	pig	EPC1G2	53	5.3	|||||

☑ Use Tag Data File: C:\Documents and Settings\Joshua\Desktop\tag.txt

Connected to Reader - COM 4

Figure 13: "Tag Summary" tab, indicating tags and their aliases.

Here, two example tags have been run across the reader several times. Notice that it says it has a Total of 76 and 53 for the tags. This is quite substantial; however it was only run across a few seconds each. This would result in several readings on the previous screen. Most notable here is that each tag ID has a "Tag Name" associated with it, which I placed before the screen shot. This is done by simply right clicking a tag ID and adding a name via a menu. For this to be useful, "Use Tag Data File" must be checked and a file indicated. Again I choose one of my own, an empty text file. Figure 14 indicates the results of the above data.



```
[TAGDATA]
A1A050804A150747AA020101=pig
A1A050804A150804AA020901=cow
```

Figure 14: Text file output from "Tag Summary" section of program. Permits aliases.

This concludes our familiarization with the device. We have figured out the box, wired it together and successfully interacted it with a computer. We've tested tags successfully, and acquired the data that shows tags and their times passing sensors, as well as a key to associate tag numbers with names. This is valuable progress towards the goal of a workable remote-access program to interact with this data.

2. Problem Description

The RFID project I propose is twofold. First, the program must be organized effectively using the data acquired in Section 1.3 with search abilities. Then, it needs to be made available via the Internet. The final result should be a remote-access program that can access a server and pull from it logs of what tags have passed through and when. It should have a form of organization that allows for a user-friendly interface, with alias capabilities to turn strange hexadecimal output into readable language, and socket-protocol accessibility from anywhere, anytime.

2.1 Project Stage I

The output of the RC Command Suite offers an ideal source of data for extrapolation. The second stage of this program is a fairly straightforward section, which includes simply taking the text files, parsing it into data and placing it into objects that can be properly organized and viewed by a user. This leaves us with three problems:

1. Decoding Data Files: This covers the pseudocode behind the extraction of the data from the text files. This includes both the log file and the tag file, which contains the tag number and time stamp and its associated English-equivalent, respectively.
2. Data Organization: Once this data is extracted, it must be organized into a class object. This will allow us to create an array of objects, each one pointing to a tag number.
3. User Interface: Finally, we must present it in a useful manner to a program user.

2.1.1 Decoding Data Files

An example of some output data, as taken from Figure 12, looks as follows:

{Rd,d:A1A050804A150807AA021001,t:EPC1G2,r:0b;FC :: 10/5/2008 11:37:24 PM

Each item is delimited by a return line, making it easy to navigate from item to item. Our algorithm for understanding the useful sections of this is quite simple. The pseudocode below describes how this is possible.

```
1.  Begin loop, while (next ‘:’ mark != 0 && !EOF).
2.  Locate the next ‘:’, save this location value in an integer.
3.  Increment the integer by one, so it indicates the first hex digit.
4.  Take the next 24 characters, completing the hex tag. All tags are 24 digits long.
5.  Find the next ‘::  ‘, again as an integer.
6.  Add four to the above value. Why 3? Because there’s a space after the ‘::’.
7.  Locate the next carriage return, an integer.
8.  Decrement this value by one.
9.  Take the string beginning at the first number and ending at the second.
10. Place these two strings into a class object, completing an organized tag item.
11. Continue loop from beginning.
```

How do we derive the name is simple. The following is an example from Figure 14:

A1A050804A150747AA020101=pig

The algorithm involves taking the tag from string “hextag” above, locating it in the tag file, and filling in what shows up after the ‘=’ sign until the carriage return.

2.1.2 Organizing Data

The second part is just effectively placing data into a “tag” object. A respective class could roughly appear as follows:

```
class tag {
        String name;
        String hex;
        Date d;
        //Various get and set functions
}
```

Theoretically, we should be able to just create an array of tags at this point and have ourselves a nice database of information:

```
tag myTags = new tag[20];
```

We've got a good format to continue on. Next, we need to show it to the user.

2.1.3 User Interface

The third problem simply involves outputting the data in an effective, organized manner to a user. The main screen should display several things. First, perhaps a series of buttons which allow search based on various criteria, allowing the user to view the whole list of tags and date entries, and filter it if necessary. Other buttons allow the user to "sweep" the log files, simply deleting it on the server side and allowing new entries to fill in, and finally a disconnect button, which pertains to the third stage of the project.

Another useful idea of the main screen is statistical and tracking information. The user can be presented with data about which tags have recently passed, what time, and which ones have been the most active. This provides a very comprehensive idea of the activity of the tags. Furthermore, we can provide simple accounting: How many different tags we've identified, and how many total date entries have been recorded, and how many have passed in the last hour. That information, combined with "most active", would allow an excellent monitoring system without requiring a query.

By this, assuming we were monitoring the progress of the RFID box remotely, we could see every time a new tag passed via a time/date box. It would also throw a recording to the "Last Three Tags" section. Picturing a tracking scenario, we could know very quickly once a tag has passed through a doorway/past a sensor, and the program would automatically update itself. If we gain an invested interest in a particular tag, we could query its name or hex equivalent, or see what's gone on through a particular passage of time.

2.2 Project Stage II

At the moment, our client side theoretically knows how to organize data and output it to the user, but it has no idea how to receive it. We're missing a very crucial step, it seems.

This aspect of the program seems easy to accomplish. We have two options here with which we can proceed, both with pros and cons. We can either present this as an applet on a website, allowing access to anybody who comes across it. There is a serious security problem here: We would have to potentially have login accounts and security protocols to cover our tracks here. We don't want anybody to be able to access the logs of our paper processing plant, after all. This is a feasible goal. However, by making this project an application instead, we can simply distribute it to those who are authorized to use it. The server prevents non-clients from accessing any information. Passwords would be easier to setup as well, if so desired. The only drawback is that it's less available on the fly, which is a minor problem.

Therefore, we will choose to take the route of an application. The problem of Internet connectivity can be solved using Java sockets. The server could theoretically just forward its information to the client, wherein the client organizes and interprets it, as discussed above. With all of our questions posed and analyzed, we're ready for some solutions.

<u>3. Solutions</u>

Now we can finally delve into the implementation of the programs. The client program is organized into six different classes. These can be organized into two separate major categories. Then, we'll analyze the server-side of the program, which is simple enough to have its own section without worrying about the specific parts.

1. Stage I: Interpreting and Organizing Data
    1. Tag
    2. TagHandler
    3. MainPanel
    4. RFIDApp
2. Stage II: Internet/Socket
    1. ConnectPanel
    2. RecThread
3. Server-Side Analysis
    1. RFIDServer
    2. OutputThread

3.1 Stage I: Interpreting and Organizing Data

The four of interest at the moment are the Tag, TagHandler and the MainPanel classes, and at the root of everything, the RFIDApp class. This list excludes the classes that deal heavily in the internet aspect of the project. We will now discuss the bulk of the client side program.

3.1.1 Tag Class

The Tag class is quite simple, and will simply be outlined. It does nothing but store objects and access objects, and maintains a counter of the number of Date objects its recorded, all of which are very simple.

```
class tag {
      private static final int MAXENTRIES=200;

      private String name;
      private String hex;
      private Date[] entries;
      private int entryCounter;

      public Tag(String name, String hex, Date entry) {
            …
            entryCounter=1;
      }
      …
      public void addEntry(Date d) {
//Adds an entry. Sometimes a null parameter is passed by considerTag.
//We do not implement deletion of items on list, so no maintenance
            if (d != null) {
                  entries[entryCounter]=d;
                  entryCounter++;
            }
```

```
    }
}
```

3.1.2 TagHandler Class

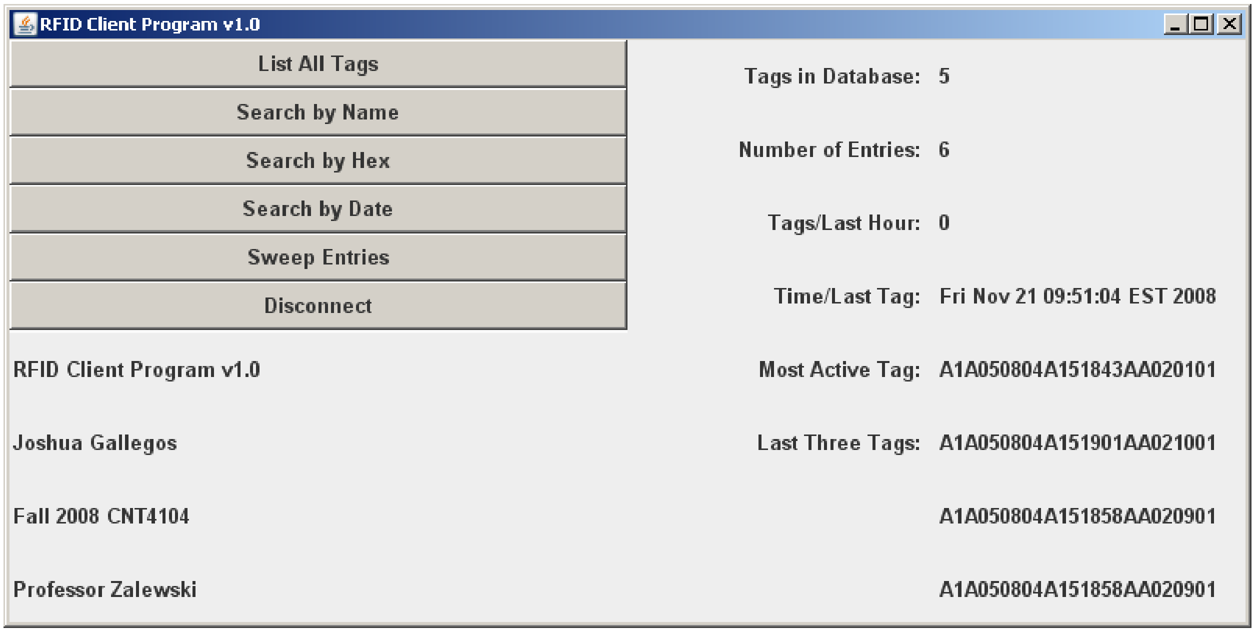
The TagHandler class is created by the main program to organize and access tag objects. It has a function called “considerTag” which takes a given Tag object and considers it. If the Tag hasn’t been identified before, a new slot is made for it. If it has been identified, then the new date entry associated with it is added to the particular Tag. It also serves to provide statistical and accounting information to the main screen, with various calls which send back the relevant data. Here is an overview of the TagHandler:

```
public class TagHandler {
    private static final int MAXTAGS = 50;
    private Tag[] tags;
    private int tagCounter;
    private int entryCounter;

    …

    public void considerTag(String name, String hex, Date d) {
        int index;
        index=locateTagByHex(hex);
//In both cases, if it can't find a hex, it doesn't exist.
//One field is empty or null below, depending on how it's being passed.
        if (index == -1) {
            addTag(name,hex,d);
        } else {
            tags[index].addEntry(d);
        }
        if (d != null) {
            lastTimeEntry = d.toString();
            entryCounter++;
            thirdLastEntry=secondLastEntry;
            secondLastEntry=lastEntry;
            lastEntry=hex;
        }
    }

    public String getMostActiveTag() {
        int max=0;
        String result="N/A";
        for (int i=0;i<tagCounter;i++) {
            if (tags[i].getEntryCount() > max) {
                max=tags[i].getEntryCount();
                result=tags[i].getHex();
            }
        }
        return result;
        }
        …
}
```

Large portions of the class have been removed for space-saving sake, but the general idea of it can be seen. Several querying functions exist, such as the one shown above, which returns the most active tag. It is all fairly simple game, and involves simple data manipulation and toying with the Tag class. The important considerTag can be seen, which provides us with “last tags” information and files new entries in.

3.1.2 MainPanel Class

This class is enormous, with tons of labels and buttons and the like. Most of it is trivial and unworthy of documentation.



RFID Client Program v1.0

Joshua Gallegos

Fall 2008 CNT4104

Professor Zalewski

Tags in Database: 5

Number of Entries: 6

Tags/Last Hour: 0

Time/Last Tag: Fri Nov 21 09:51:04 EST 2008

Most Active Tag: A1A050804A151843AA020101

Last Three Tags: A1A050804A151901AA021001
A1A050804A151858AA020901
A1A050804A151858AA020901

Figure 13: The main screen of the RFID Client Program

As shown in Figure 13, it’s organized primarily into a 2x2 grid-style layout, with further separate grid layouts to organize the various labels and buttons. Looking at the header of the class file reveals important information about the functionality here:

```
public class MainPanel extends JPanel implements ActionListener {
```

We have here a JPanel with an ActionListener for handling button presses. Other than this, we only have one useful function here, visuals aside: The update function. The commented part indicates that this particular function hasn’t been completed yet and doesn’t work.

```
public void update() {
        TagHandler t = applet.getConnectionPanel().getThread().getTagHandler();
        allTagsResultLabel.setText(Integer.toString(t.getTagCount()));
        numberEntriesResultLabel.setText(Integer.toString(t.getEntryCount()));
        activityResultLabel.setText(t.getMostActiveTag());
//      numberPassedHourResultLabel;
        timeLastPassedResultLabel.setText(t.getLastTimeEntry());
```

```
      firstTagPassLabel.setText(t.getLastEntry());
      secondTagPassLabel.setText(t.getSecondLastEntry());
      thirdTagPassLabel.setText(t.getThirdLastEntry());
}
```

This wordy stuff is simply us updating all the result labels on the main screen. It has to point across a few different objects, retrieving various functions, making it appear quite complex. However, it's just accessing mostly functions in the TagHandler class.

3.1.3 RFIDApp Class

The root of everything, this class is responsible for switching between the various panels, giving the user the proper display. It also is coded to provide a gradient background, a simple visual effect, and it is the base for the Colorize function, which has thus far not been mentioned. Colorize is an overloaded function that simply gives all objects a uniform look and feel, making it easier to color and size everything equally.

First, we'll look at the header, variables and constructor:

```
public class RFIDApp extends JApplet {
      …
      public RFIDApp() {
            setLayout(null);
            connectionScreen=new ConnectPanel(this);
            mainScreen=new MainPanel(this);

            connectionScreen.setBounds(220,100,connectionScreenWidth,connecti
      onScreenHeight);
            mainScreen.setBounds(0,0,640,320);
            add (connectionScreen);
            add (mainScreen);
            hideMainScreen();
      }
```

We create new screens, the connection screen (Not yet discussed due to it having an internet-related context, and therefore subject to Stage II of the project) and the main screen. We hide the main screen initially, since we only want to show it once we've found a server. Other functions, too simple to show, simply hide and show the main and connection screens, which are passed when we're connecting or disconnecting from the server.

This concludes our analysis of the client program in this section. What is missing? As mentioned before, we've saved the internet-related classes for later. Including the Tag class, we've covered four of the six classes of the program.

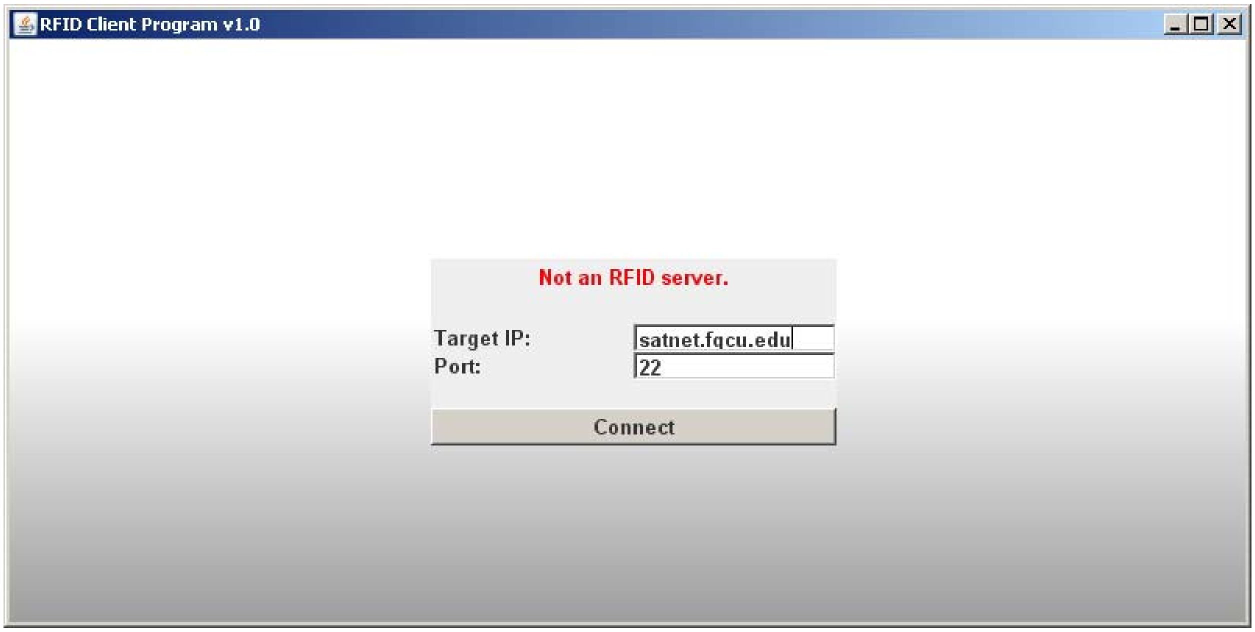
3.2 Stage II: Extension to Internet Capabilities

With all of the data and organization taken care of by our client program, we're left with two classes which are responsible for internet connectivity and receiving data.

3.2.1 ConnectPanel Class

A large section of this class is just graphical user interface, unworthy of documentation. Beyond that, we have an ActionListener associated to the connect button. The actionPerformed function calls our connect() function, assuming that the fields have been properly filled out.

The above function connects to the indicated location. If a connection is made, we go to RFIDApp's functions to hide the connection panel and show the main one. The thread is started, which handles the socket thereon. If an error occurs, then a status label at the top of the screen says so, as well as if the indicated server fails the chatter protocol. Figure 14 illustrates this:



Not an RFID server.

Target IP: satnet.fqcu.edu
Port: 22

Figure 14: The Connection Screen

As shown above, I've attempted to connect to the satnet SSH port. When the client attempts to chatter, the satnet server obviously doesn't respond correctly, and the connection attempt fails. The status label shows this result.

3.2.2 RecThread Class

Our final step in understanding the program is seeing how we receive data. We know how it was organized already, so we are simply left to understand how it is thrown to the TagHandler.

The good stuff has been isolated into the function interpret(string). The rest of it is regular old socket code. Interpret works as follows:

```
public void interpret(String s) {
String hex;
String name="";
String date;
Date d = null;
      int index=s.indexOf("=");
if (index != -1) {
      //Case 1 Tag Descriptor
      hex=s.substring(0,index);
```

```
            name=s.substring(index+1);
      } else {
            //Case 2 New Log Entry
            index=s.indexOf(":");
            index++;
            hex=s.substring(index,index+24); //All tags are 24 in length.
            index=s.indexOf(":: ");
            index += 3;
            date=s.substring(index); //Rest of it.
            d=new Date(Date.~~parse~~(date));
      }
      th.considerTag(name, hex, d);
      applet.mainScreen.update();
    }
```

The pseudocode for this was previously described in this document, and the final result is very similar. We simply parse up the data into parts and send it along to the TagHandler. We take advantage of an extremely useful function of Date, the .parse function. It allows us to throw the date string exactly how it appears in the tag and Date interprets it correctly.

3.3 RFID Server Program Analysis

The server is an extremely simple design. It's only real purpose is to read data from files and forward new data to the client, when connected. Although this section will inevitably tread on the internet-side of the project, it is minimal, since it only composes a short part of what's happening. This is the only comprehensive view of what's happening on the server side. It is also notable that there is no user interface. Once the program is running, it simply listens or transmits. The only output just shows when connections are made and lost, and serve no useful purpose other than curiosity.

The server is composed of two classes: RFIDServer.java, and OutputThread.java. We'll analyze the main class first, and then explain the thread.

3.3.1 RFIDServer.java

We start by declaring the location of our two files.

```
public static final String tagFile = "c:\\...\\tag.txt";
public static final String logFile = "c:\\...\\rfid.log";
```

On an infinite loop, we start by just listening:

```
sock = new ServerSocket(4949);
client=sock.accept();
```

Then, input and output readers are established, and once a connection is made, we proceed with a "chatter" protocol. This is used to verify that the client is actually an RFID program. How do we accomplish this? We simply transmit a strange segment of data. If we don't receive what we want to receive, then the connection is severed and the connector is told he's not authorized. This could only be used to verify that the client is an updated version, if we were creating multiple versions of the program.

Since I happen to have an elementary understanding of Vietnamese, a sufficiently strange manner in which to "chatter", the chatter protocol is as follows:

```
out.println("~`~`Em la mot nguoi ban khong phai?`~`~"); //Chatter
if (!input.contains("~`~`Troi oi! Em la mot nguoi ban. Con anh?`~`~")) {
      out.println("Unauthorized access!!!");
      System.err.println("Unauthorized access!!!");
      break;
}
```

For the curious, it roughly translates as, "You are a friend, no?" The reply reads, "Wow! I am a friend. And you?". Along with the "~`~`" part, it's hard to imagine someone getting this correct by random chance.

If a connection is successfully made, then the thread is started. The client does not have much reason to send data to the server, but two conditions exist in which it does, and the server now starts to listen for it:

1. ~`~`RFIDClientBye: Closes the connection to the client. Occurs on the "disconnect" button.
2. ~`~`RFIDClientSweep: Deletes log file. Occurs on "sweep" button.

Since we're on a loop, we automatically go back to listening after a connection is broken or an error occurs. This concludes the server's main class.

3.3.2: OutputThread Class

This is a very "wordy" class. The idea is quite simple however. For each file, the tag and the log file, we want to open it up, read it send along new entries to the client. We accomplish this "new data only" protocol on the server side by maintaining variables that contain the contents of the previous entry. Each time we open the file, we cycle through until we find the last entry, and continue from there.

In the case of the log file, we also must consider that the last entry might be identical to another entry. Therefore, we skip to the last occurrence of the last entry, which conveniently occurs concurrently since the data is chronologically ordered. This is accomplished easily:

```
      while (input != null && input.equals(lastEntry))
            input=inLog.~~readLine~~();
```

After each time we've read through the files, we wait a certain amount of time, saved in a class variable.

```
      private static final long UpdateTimer = 8000; //How often this
cycles...
```

If we were so inclined we could make this settable by the client, and allow updates to occur as frequently as desired. This would result in a little more network activity and drain on the server-side computer a bit more, especially if the log files got very long. An eight second delay seems reasonable, since we're tracking papers through a building in our example, hardly a second-by-second operation.

This concludes our server side program. It simply accepts connections, listens for two unique commands, and sends along new data from the two indicates files when it happens. All organization occurs on the client side. Our knowledge of the program is complete, and we can move onto an example case of how we could use this in real applications.

4. Example

This program has a wide variety of uses and could be placed into many different industries to improve productivity, help with loss prevention and generally lessen headaches. Anywhere with a specific inventory of items could be well served by the RFID technology.

The first question a potential client would ask is, what can it do for me? The answer is quite simple. The program provides:

- Tracking of inventory as it passes through a building, by setting up an array of sensors.
- Effective loss prevention, as items have better accountability.
- Time management properties, as an item "production time" can be monitored.
- Assurance of job quality, as items must pass through particular doors in time frames.

Therefore, with purpose and direction, we can propose an example company which this program could be marketed to. The RFID sensors would be set up throughout a building, most effectively placed at doorways. Tags must be placed on all inventory items. Now we can have very accurate traceability.

In this example, we will demonstrate a paper tracking application of the RFID technology. The theoretical customer needs to track with time precision, the passage and movement of sensitive papers through his office building. With sensors on all exits and key doorways, the customer can be offered the ability to query at any time a paper of particular interest.

Let's address the previously discussed four points of productivity for this application. First, we know it will track the paper: The doctrine will trigger all sensors as it passes throughout the building. An employee, who has been entrusted with the document, removes it from an archives room on the second floor. The paper throws the sensor at the doorway, at the elevator, again at the elevator on the first floor, and finally at his office. This covers our tracking.

What about loss management? Say this employee, who secretly has been conversing with the competition, decides he wants to fax this particular document to an outside party. He is flagged as entering the copying room, where he should not be, with the paper. The source of the leak is therefore immediately discovered, as he would have no good reason for entering that particular room.

What if he were to take the paper and take a lunch break before heading back to his office? This can be traced. This particular topic is not quite as applicable, however if the tag were instead placed on the employee instead of the paper, it would always provide time management: He would be tracked, paper or not, throughout the building. If he decided to head over into the break room for two hours, his unauthorized time mismanagement would be recorded. In this particular application, we would only know if he decided to head to his destination extremely inefficiently.

Finally, we better know that the job was done: The very fact that he went and picked up the paper, and not somebody else (as the final destination would be different), we know that the employee actually did the job and actually fulfilled the task. This is not absolutely positive, since the employee could potentially just take the paper to his office and do nothing with it before returning it, but we can account for him actually taking it. If he never picked the paper up, the program would know and catch the employee in a lie.

The following diagrams give an idea of what this sort of thing might look like. First, we examine mock maps of the buildings where sensors are placed, and we theoretically show tag readings and how tracing actually would occur on the client side, as shown in Figure 15.

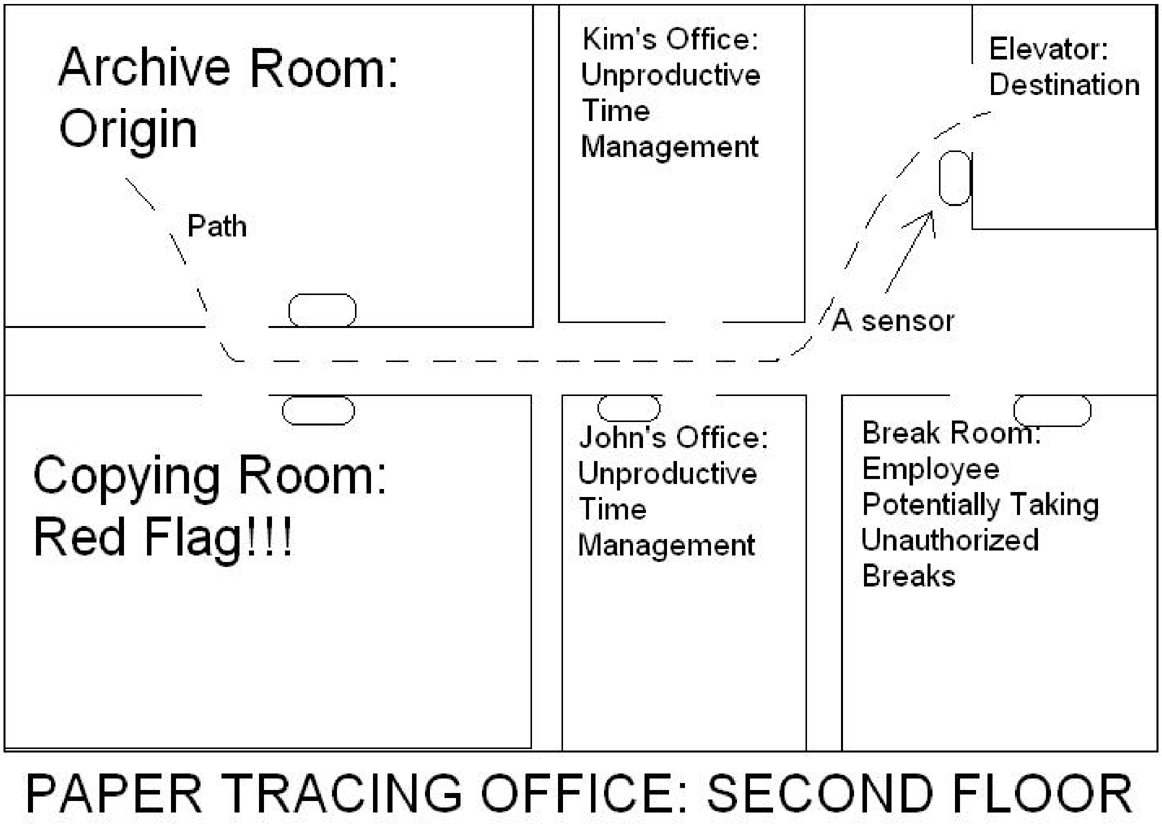

Archive Room: Origin
Path
Kim's Office: Unproductive Time Management
Elevator: Destination
A sensor
Copying Room: Red Flag!!!
John's Office: Unproductive Time Management
Break Room: Employee Potentially Taking Unauthorized Breaks
PAPER TRACING OFFICE: SECOND FLOOR

Figure 15: Example of an office environment using the RFID technology.

We see here an example of what the path of the employee might look like. He should first be throwing the sensors when he leaves the archive room. A timestamp would be associated with this. There are several other places where we would not want him to go, and if his timestamps showed passage, he would be potentially in the wrong: He absolutely should not be entering the copying room, where he may be faxing or reproducing the sensitive document. Furthermore, if he went into a neighboring office or a break room to fraternize or lounge with the document in hand, it would be noted. A record of his movement in the program should look like this, when querying the tag: A1A050804A150807AA021001

…..11/20/08 1:58pm Sensor #1: Archive Room
…..11/20/08 2:00pm Sensor #2: 2nd Floor Elevator

In this example, of course, we have placed a slight extension on the sample program that has been created for this program: We're in a multi-sensor environment, where the test program only accounts for one sensor. This idea would be a very simple extension onto the program that's currently employed. Several other small extensions might also be advisable: Scheduled task capabilities, which would automatically flag irregularities in a paper's scheduled path. This would automatically sense that users have entered the copy room or some such action and alert the administrator.

By examining the above data, we can see that the employee did what he was supposed to: He moved from the archive room to the elevator in an acceptable amount of time, and did not take any unauthorized actions. Of course, if he did enter an office, it does not necessarily mean wrongdoing: Perhaps a colleague has simply asked his opinion on something. However, if mischief was suspected, the employee could at least be asked about the occurrence.

We also must examine the reliability of this data before we can attempt to market it to the customer. The following chart shows the effectiveness of three tags, by demonstrating the number of readings over a "short pass" in front of the RFID sensor:

# Readings/Distance	Tag #1	Tag #2	Tag #3
0 ft	4	2	5
2.5 ft	4	1	4
5 ft	5	0	3
7.5 ft	3	0	2
10 ft	3	0	3
12.5 ft	0	0	1
15 ft	0	0	0

Clearly, some tags are superior to others. The first tag had more readings than the third one, on average. However, the third tag also had superior distance. It read as often, but the third one caught more overall. The second tag was clearly barely effective, usable only in the closest of quarters. What does this tell us? First of all, there is a good amount of range with the average tag, making door mounts an effective solution: It is reasonable to assume that a user would pass within about 10 feet. Second, tags need to be tested from time to time to ensure effectiveness, since some tags clearly lose potency and are not made equal. A range of 2.5 feet would be entirely unacceptable, since it would be quite easy to dodge the sensor by holding the document at one's side, high or low depending on the sensor height.

Another point of interest would be "How often does the tag fail with distance?" This graph would look extremely similar to the one above, however, since there appears to be extraordinary reliability and consistency in the device. If it reads at 10, it'll always read at 10. If it doesn't read at 12.5, it pretty much won't read at all at 12.5 feet. There is little lottery with this device. Therefore, it would be pointless to create a new table, since it would be simply replacing numeric values with "Yes" or "No", effectively.

Therefore, we have demonstrated a very effective tool for a valuable application. We could market this program to a firm which has very sensitive documents, where the relatively small cost of the sensors and tags would be made up for in the extra security and accountability of documents. There would be very little loss of papers, since a lot of mystery of the movement has been unveiled. Many companies could stand to benefit from this automated security and tracking program.

5. Conclusion

RFID technology offers a very simple capability: When a tag passes nearby, it records it with a unique identifier and a timestamp. All of the applications of this sort of technology are extensions on this very basic ability. With some programming savvy, this sensor can be turned into a very productive tracking system, which could offer invaluable security and accountability to a potential user.

Applications in the current world are limited, although they could potentially be much more widespread. In most major retailers, things such as video games are currently use RFID to prevent theft of video games and other sensitive, expensive electronic devices. When the tag nears a sensor which is placed at exits, the sensors go off annoyingly. When the items are purchased, the tags are killed magnetically. Intelligent sensors could even identify the item as it approaches the door, and not go off if the item doesn't register in the store's inventory, preventing nuisance alarms from going off. Furthermore, it could alert security personnel exactly what item is being stolen.

The military uses the tags, and it's placed on some livestock as well. It could be placed on grocery items we purchase every day, allowing instant scanning and totally of a grocery cart: A customer could just walk up and immediately be prompted with a total, cutting the grocery line from fifteen minutes into the time it takes to swipe a credit card! Clearly, many vast applications exist, not limited to the office scenario we've described or the limited applications it's currently employed with. It is a powerful technology that will no doubt make a second coming in the future, and can provide extremely time-efficient solutions to major firms out there.

6. References

1. Porter, Robert, Evan Flechsig, and Olexiy Kovtunenko. "Data Acquisition through RDIF." Report. FGCU. 9 April 2008
2. "RFID.“ Wikipedia, The Free Encyclopedia. 12 Sept. 2008 <http://en.wikipedia.org/wiki/rfid>.
3. Serial-to-USB driver <http://www.floridaprobe.com/downloads/rshack.zip>.
4. "What is RFID?" Technovelgy.com - where science meets fiction. 12 Sept. 2008 <http://www.technovelgy.com/ct/technology-article.asp?artnum=1>.

Appendix A: User Manual

This manual will briefly explain how to get from start to finish on this project. Since a lot of the material is already covered otherwise, little explanation is needed.

A.1: Getting Started

The first step is to get the device running. The device actually comes with an instruction manual located on the accompanying CD. Find the file, "MP9320_2.7_Users_Guide_V6.0.pdf", which gives detailed instructions on getting started with the software and hardware. Additionally, Section 1.2 details some of the hardware side of this, including a schematic (Figure 8) of how it should be wired up. Section 1.3 continues to explain a little bit about the software, and how it works.

It is very easy to sum this information up however: Plug into the main box the square-shaped sensor, the power supply and the serial cable. If done properly, the device should have two green lights on the top during idle operation, where red lights would indicate problems. The software must be installed per the simple directions.

There are two important points of interest here. First, when setting up the RFID box, make sure that all of the resistors are in place on the sensor plugs that aren't being used. This is vital to the operation of the device, per the indication in the user manual, and the box will throw a fault light if this is not done.

Second, once the program is set up properly, the logs must be created for the main screen, the "Command" tab, where tag readings are actually seen. Without this log, the project program cannot function, since it operates by tapping into the log files. Additionally, the "Tag Summary" tab includes another aspect of the program, where tags can be actually named. To utilize this non-essential feature, a tag file must be created. Section A.2, which discusses the program created in this project, will explain where to put these logs by default.

To summarize: The device must be assembled properly and hooked up to the computer. The software in the CD must be installed, following the instructions in the provided user manual on the CD. Once proper operation has been confirmed, logs must be started, allowing records of tag readings. At this point, it should be possible to wave a tag in front of the sensor, get a reading on the SAMSys program, and see it in the accompanying log. Figure 11 demonstrates this stage of the project, showing many readings from many tags and a properly appropriated log file, as seen at the bottom.

A.2: The Program

The files for this program can be accessed at http://satnet.fgcu.edu/~dxgalleg/rfid.rar. The program is incomplete as of this stage of the project, however it has significant portions complete.

Once everything has been readied, the program is ready for execution. This is a server/client-style program, and for intended use, the computer with the hardware device must be running the server side. There is only one point of interest when running the server side of the program: in RFIDServer.java, the two finalized variables at the top need to be customized to the computer's log locations, as created in the SAMSys software. By default, it points to C:\, which is fine. The only requirement is that the two locations agree, or no data will be transmitted, since the server program will simply think there are no log files to be read.

With the server-side running, the client side must be executed. This can be done from the same computer for testing purposes, but it is designed for across-internet usage. When running the client side program, target IP and port number fields are available. Default is port 4949, but can be changed on the server side if so desired. The default target is “localhost”, or in other words the local machine, for testing purposes. This must be changed if targeting an outside computer.

For obtaining the address of the target computer, open the command prompt and type in “ipconfig”. The address listed under “IP Address” should be put into the target text field in the client program.

Click “connect” once the fields are appropriately filled out. The program should connect, if the server is running properly, and a new screen should show up with buttons and statistical information about the tags. This is the main thrust of the program. It offers accurate and useful information about the tags being read in. The program is set to read new information every second. This number can be changed on the server-side of the program in the OutputThread.java class, by changing the finalized variable at the top. For impatient users, this number can be reduced, although 1 second should be quite sufficient. For bandwidth-conscious users, the number should be extended to something like 10 or 20 seconds. Of course, readings won’t come across at lightning speed, but for slow purposes it will work fine.

The buttons are not entirely complete. If any search query buttons are pressed, it will quickly become apparent that the program has not entirely been completed. The next semester should yield more results, as this stage of the project was extraordinarily time consuming. Therefore, the search buttons should be ignored for now. The “sweep” button effectively deletes the log files on the server side, so entries previous to the sweep are effectively ignored. “Disconnect” is self-descriptive, and breaks the connection to the server gracefully.

With the program understood and the device operating correctly, this concludes the user manual. It is quite simple to operate at this point. It only needs to be wired up, with the SAMSys software running and the server and clients running properly.