

May 01, 2005

Fitrax: workout tracker

Ryan Leonard
Northeastern University

Recommended Citation

Leonard, Ryan, "Fitrax: workout tracker" (2005). *Honors Junior/Senior Projects*. Paper 17. <http://hdl.handle.net/2047/d10001724>

This work is available open access, hosted by Northeastern University.

Fitrax: Workout Tracker
MS10

Date Submitted: April
21, 2005
Advisor: Masoud Salehi

*Ryan Leonard

Table of Contents

Introduction	2
Background	3
Fitrax	4
Problem Formulation and Analysis	5
Design Strategies	6
Originally Proposed Design	8
Cost Analysis	10
Enhancements	11
Final Design	12
Appendices	16

Appendix 1 : IBM 4779 Hybrid Smart Card Device

Appendix 2 : Card Reader Hex Code

Appendix 3 : Card Reader C Code

Appendix 4 : User Interface Code

Introduction

Many people who go to the gym on a regular basis like to keep track of their workouts so that they can track their progress. The easiest way to do this right now is by carrying around a little notebook and pencil and recording what you do at each machine. Carrying around any extra items at the gym is a burden and our goal is to allow people who wish to track their workouts be able to do so with their Husky card, which you need to get into the gym.

Our proposal is to install card readers with keypads and displays for each machine. The user swipes their card and is logged into that machine and when they are done they swipe out. Each one of these readers has a unique ID and that way they know what type of questions they need to prompt the user with. For example, if one were doing a weightlifting exercise it would ask you how much weight you are lifting and how many repetitions, if it was a treadmill it would ask how long you ran for. The user's input is sent wirelessly back to a server where it is stored for processing. After a workout and all the input has been processed the user will be able to have a lot of useful information.

When the user leaves the gym they will be able to look up their workout online in their own personal database. The database will contain information on the workout you just completed, your past workouts, the improvements that you've made and other useful information. This new system that we have proposed would make it much more practical and easier to track your performance at the gym

Background

There are some very expensive systems used in gyms that are used to plan and track your workout. Fitlinxx Interactive Fitness Network has a system in place where computer touch screens are placed at workout stations will display your individual workout based on a login that you must enter. Your work out is compiled from analysis of a personal profile questionnaire and even reviewed by a trainer. When the workout is done the user returns the workout log to the club to tracking purposes. Trainer Richard Draggs at the

Boca Raton YMCA has seen memberships increase by almost 100 participants a month since the Fitlinxx system was introduced. He says that it helps make people who wouldn't workout regularly do so.[4] Our system varies from theirs in the implementation and complexity. The Fitlinxx system goes as far as to give you your workout. We are allowing you to do your own workout, just making it easier for you to record the results and have them permanently available to you. Our system is a much cheaper solution and is focused more on gathering information from your workout and all past workouts to track your progress over a long span of time. Card readers that have wireless communication capability built in do not exist at the time being. Many card readers have several methods for input such as LAN, serial, or modem but none have wireless. One of the main features of our system is that we want the data to be transmitted wirelessly from the card reader to a central server. To solve this we first looked into creating a Zigbee solution but the technology is still relatively new and incorporating it into a card reader looked like it would be extremely challenging. The next wireless solution that we came up with was to use a serial to Bluetooth converter with the card reader. The Bluetooth technology has been in use for several years now and we feel is more sturdy of a platform than Zigbee. The strength of the signal would be about 30 feet we believe so if it were to be deployed on a large scale we would need Bluetooth access points spread out across the gym. We looked into a variety of card readers and only found a few that had all the features we were looking for. We need a programmable input so that we are able to program questions to ask the user. We also needed some sort of display, a keypad, and a serial port. The TA 520 Data Collection terminal is one of the card readers that has all of these components. We spoke with one of the sellers of the product and he seemed interested in our project and said that he thought that the TA520 would suffice for everything that we need it to do. The only problem with that was that they are trying to sell them at very high price, higher than our budget for the project for one unit. In the end, we had to go with an older used card reader we found on eBay, as it was the only one we could afford. While this solution was cheaper, it forced us to connect our prototype to the server over a wired LAN.

Fitrax

The Fitrax: Workout Tracker is a card reader system that will allow you to keep track of your progress at the gym by simply sliding your card and entering some information. No longer will you be required to carry a notebook to the gym

to record your data for the day. With the Fittrax system you will be able to access any information acquired that day from the gym from your home computer. Also with the Fittrax system you will be able to view the information in many different ways as opposed to the chart you create in your notebook. The interface where you can retrieve all of your information will be very simple and user friendly. The rest of the design will be discussed later in this report.

Problem Formulation and Analysis

A number of decisions had to be made in order to reach our final design. Additionally there are a number of concerns must be accounted for throughout the design process. One level of concern we have come across is the idea of security. With implementing all of this data wirelessly throughout the gym we have to make sure that none of this information can be easily obtained. Also security via the web based retrieval of the data that was stored at the gym. We would implement this system through the MyNeu website where students can also check their email and many other Northeastern matters.

The placement of the card readers is also a factor in the design process. The Marino center has already been built and all the machines are in their respective places. We cannot simply place wired card readers throughout the entire facility, it simply wouldn't work. The card reader itself also must be of a certain variety; just a card reader will not suffice for our purposes. We need a card reader that has a display as well as a card reader that has input keys. Also this product will not just be used for the Marino center here at Northeastern University. After development of the product we would like to be able to sell to other gym facilities. So the technology cannot be extremely complicated to implement and maintain. In selling the technology to other facilities we would only have them install the card readers throughout their facility. Each card reader will come preprogrammed for a specific machine, the SQL database may increase or decrease depending on the facility. However since the Marino center is already a top notch gym most machines there are also featured at other workout centers. The web interface will change slightly for different places also how the end user will login will have to change.

The web interface that we design will also have to be quite user friendly. Our product is designed to have people leave their notebooks at home and also for the people who forget their notebooks and cannot remember what and how much they worked out last time they were at the gym.

Design Strategies

Initial brainstorming and discussion resulted in many proposals and design strategies for developing our workout tracking system. We recognized that we had to take input from the user at each machine and somehow send this data to a database for future analysis by the user as can be seen in Figure 1. Our first question was how would the user enter input and how would the system recognize different users? There were debates as to whether the user should use a Husky Card for input or carry around Palm Pilots to each station. Palm Pilots could be carried around with users and data could be entered directly into them. When the user completed his or her workout, the Palm Pilots would send the data entered into a database. The only problem with a Palm Pilot is that it is burdensome to carry around and would be expensive to give each person one.

With Husky Cards or gym access cards being used, a user could carry the card with him or her to each station and swipe into a card reader. The user would then enter data using the keypad and the card reader could then send the data entered to the database. With this strategy, the gym would have to pay to implement the card readers at each station and create a network for the readers to communicate with the server. In the end, we realized that the Husky Card method would be much more feasible due to the ease of use and lower cost to the gym.

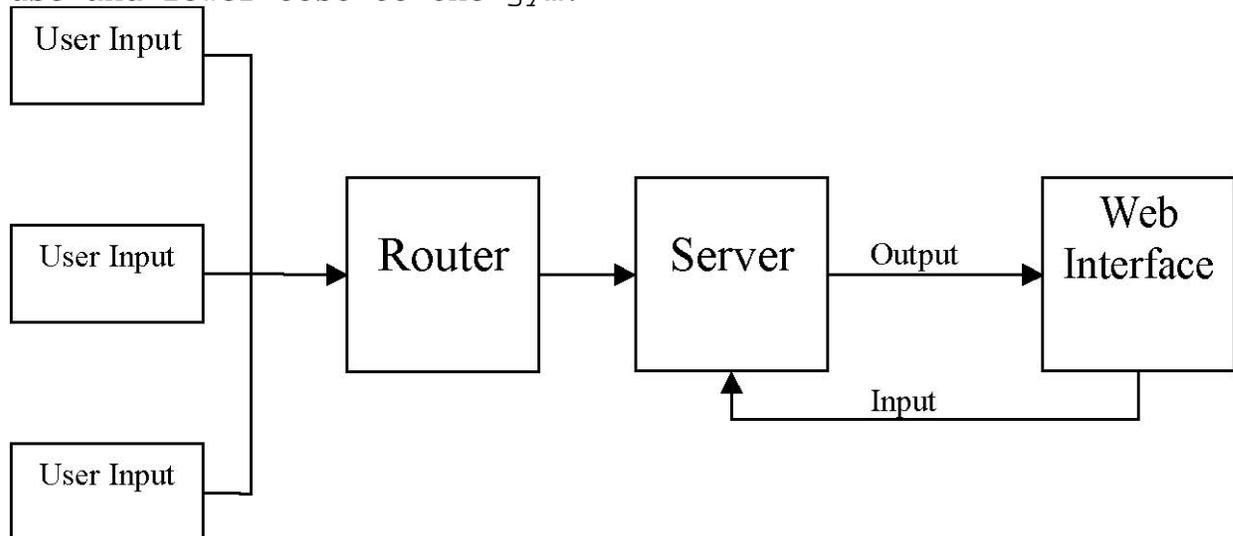


Figure 1: Data Flow Diagram

The second problem we faced was how to get this data entered into the card reader to the database. Our choices would be to either create a LAN using cables from each card reader and a router, or to create a wireless LAN using a wireless card in each card reader, access points, and a wireless router. If we chose to go with the cabled LAN, the solution would be cheaper, but very messy with cables running all over the gym. Also, we would run the risk of having heavy weights dropped on the cables, which would be annoying to replace. With a wireless LAN, however, a small wireless network card attached to the card reader could send a signal to the access points placed out of the way on the walls. Though slightly more expensive, the wireless solution would prove more reliable and less messy.

Now the question was what kind of wireless technology would we use with our card readers. Options that we considered were using typical Wi-Fi network cards found in laptops in PCs, ZigBee wireless technology, and Bluetooth wireless. Since we'd most likely be using the serial port in the card readers, we recognized that it would be difficult and expensive to use Wi-Fi network cards. Also, we're transferring very small amounts of data, therefore, we would have no need for the higher bandwidth offered. With ZigBee, we had the ability to transfer small amounts of data over long distances, but since ZigBee is relatively new, we found few affordable ZigBee products and no ZigBee network cards that works with serial ports. Bluetooth wireless, however, has been around for years and we found several Bluetooth devices that utilized serial ports. The only worry we had with Bluetooth was that it doesn't work over long distances. When we looked around, however, we found a Bluetooth wireless adapter and access point that worked at over 100 yards. While it says that it will work that far, we planned on placing access points about every 50-75 feet depending on how strong a signal we can get from the devices to the access points.

From there, we have to take the data sent to the server and store it in a database, then display it over the web to users who query for data. Since we're familiar with SQL, we decided to use Microsoft Server 2003 and Microsoft SQL Server 2005 for the server's operating system and database. We then decided that we would use C# programming to query the database based on input entered from the user and would then output the results to the interface.

The originally proposed final design can be seen in Figure 2. This shows the different technologies that

we're using and how they connect to each other.

Originally Proposed Design

Upon consideration of all strategies discussed in the previous section, we decided to use the following components and technologies to create our solution. A detailed description of the components can be found in Appendix 1. After a thorough search, we decided that purchasing two Tranz 330 Credit Card Processing Units for the card readers. The two Tranz 330 Credit Card Processing Units would then be upgrading them to TA520s from Time America. These card readers offered a card reader, programmable LED display, programmable keypad, and a serial port. Also, the units can be programmed to send out their data at programmed times. This will be useful, because our wireless access points can only accept seven clients at the same time.



Next, the Serial to Bluetooth converter by Socket was chosen as the best method for transferring our data wirelessly from the readers. The converter runs at a frequency of 2.4GHz and has the ability to send data over the distance we require for our design. The Bluetooth converter was also chosen, because we're only sending small amounts of data and Bluetooth is ideal for it.

To receive the signals sent from the Bluetooth converters, we chose Belkin's Bluetooth wireless access point. Belkin's Bluetooth access points were an affordable solution which offered everything we were looking for in a Bluetooth access point. The Belkin Bluetooth wireless access point also

features 128-bit encryption and a built-in web browser interface.

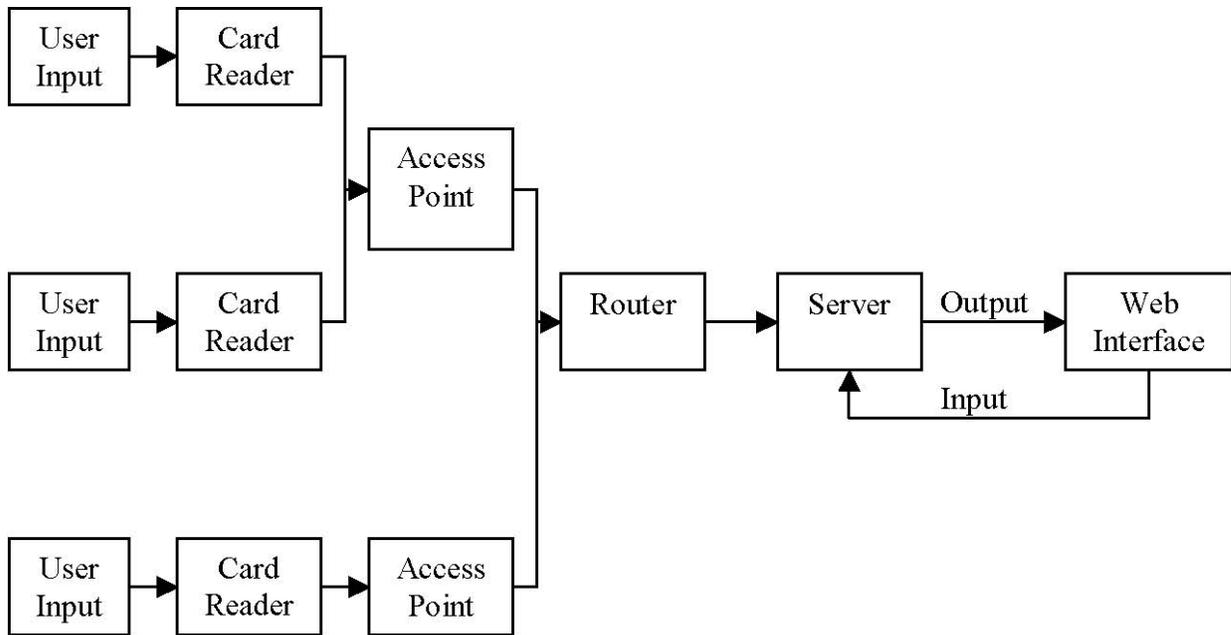


Figure 2: Original Design Strategy and Components

From the access points, the signals are then amplified and sent to our Belkin Wireless Router, which would be connected to our server machine. For our server machine, we'll be using a Microsoft Windows Server 2003 for the server's operating system and Microsoft SQL 2005 for the database. Software written in C# will be running on the server machine which will take input from users over a web interface created by us and will query the SQL database and format the returned data and output it to the interface.

Cost Analysis

There were not too many expenses involved in our project. We already had an extra PC lying around to act as the server. The only pieces of hardware that we had to purchase were the card reader and a new network card for the server that was compatible with Microsoft Window Server 2003. Aside from these pieces of hardware, we spent additional money on books to aid us in programming the interface and

the card reader. Our final cost analysis can be seen on the next page.

Description Distributor Cost

Hardware

IBM 4779 Hybrid Smart Card Device IBM \$100 Linksys

Network Card Linksys \$24.99

Books

Beginning Visual Web Programming in C# Apress \$39.99 Microsoft

Programming Microsoft Web Forms Press \$39.99 Beginning Visual C# 2005

Wrox \$39.99 Microsoft Microsoft Visual C# 2005 Press \$39.99 Beginning

ASP.NET 2.0 in C# 2005 Apress \$49.99

Total \$334.94

Enhancements

Throughout the design process our team has brainstormed many features that would make great additions to the Fitrax system. Some were able to make the final design, while others were left for future development.

Web based data α In many scenarios at the gym people do exercises that cannot be tracked via a swipe card, i.e. sit-ups and pushups. We will attempt to implement a way of modifying the SQL database via the web by the specific end user for that specific user. So if he or she does an exercise not recordable on the swipe cards they can go home login and add that exercise.

Graphs to display progress α With the data available from queries made by the user, we would like to automatically take this data and display the user's exercises in a graph over time to show their progress.

Scale Track α We would like to add a card reader by the scale in the locker rooms. This will allow end users to keep accurate tracks of their weight and to see if they are reaching or achieving any weight loss or gain goals.

Free Weights α Adding card readers to different locations in the free weight room near specific benches that allow users to select from a certain number of preprogrammed exercises. On the cable machines there are multiple different exercises that can be done all on one machine. On something like that you would have fifteen or so different exercises which can be selected from a list. And as stated before if your exercise is not on the list there would be another feature that would be editable from the web at you home.

Cardio Track α Implementing card readers on the

different cardio machines would allow gym users to keep track of how far, fast or long they are doing for cardio workouts.

BMI calculation α Body Mass Index (BMI) calculates how "healthy" a person is by certain standards including height and weight. This would be an added feature via the web so someone could calculate their BMI.

Final Design

The final design solution we settled on was very similar to our proposed design solution. The only major difference was that our card reader had to be connected to the server via a serial cable. This was due to the fact that the power supply to the card reader was attached to the serial cord. If we had a larger budget, we could have potentially purchased a newer card reader and implemented our wireless system. For the purpose of demonstration, however, our prototype had to be designed without one.

Card Reader Hardware and Software

The first step in working with the IBM 4779 Hybrid Smart Card Device was establishing communication between the computer and the device. The card reader has the capability to read Smart cards however we were only using magnetic, Husky cards in our project. The card reader itself is connected to the computer via a RS-232 serial port. Our original idea was to use a serial to Bluetooth adapter to make the device wireless however this was not possible because the card reader's adapter also has a power cord attached to it that was necessary for operation. Included in the sample code from the vendor was a program that would pass commands in the format of hexadecimal values from a text file to the card reader. After successfully compiling and building the sample code provided we constructed a text file that would pass all of the commands that we needed to use to the card reader. IBM's programming guide for the device contained all of the hex commands that the card reader would accept and the format of the response. This information was instrumental in constructing the hex commands that we passed to the reader. The snippets of the programming guide that we used along with a commented copy of the text file that was used for our demonstration is displayed in the appendix. The commands that we needed to execute were the basics; insert card, write to the LCD screen, allow the user to input data, magnetic arm and read, and eject card. Once the hex file was constructed the next task was to modify the source code so we could include

functionality we needed.

The source code that was provided was written in C. The modifications that we had to make to the source were not that invasive. We included functions that would store the user's 16 digit ID number from their Husky card, store the user's responses to the workout questions, log the date of the event and put all this into an output file. The commented source code that was used is provided in the appendix. In the A4779.c file we wrote the ReadId, WriteSetInfo, and InsertDate functions. We also modified the DoEftRead to pass it the request message which is the hex command that was sent to the reader. We need to know that because if it is 03, read keypad, that means there is user input that we want to save, and if it is 06, magnetics read, that means that we have to store the user's ID # from their Husky card. From the DoEftRead function we call our own functions if either of those conditions are met. The format for the output file that we used was .csv, or comma separated values. Comma separated values were chosen for our project because they were able to be easily imported into a SQL database. Interestingly enough, using comma separated values did present one design problem. The csv file was being stored remotely and the SQL server was accessing it through a Windows shared folder. For some reason, that we were unable to determine, once we shared the folder the output file was going into, anytime we would execute the program it would add a leading comma to the output file that could throw off the SQL importing function. This was an issue that was only presented because we had to construct a proof of concept model for our presentation, if Fitrax were to be deployed on a large scale this design issue would not be a problem.

SQL Database and Server

To store information gathered from the card reader a database was required. The most logical choice for a database was SQL. SQL (Structured Query Language) is a common programming language to create and manipulate databases. The card reader outputs information into a repository file in comma separated value (.csv) format. This information needed to be manipulated and inserted into the database. To do this I created a job that copies the repository file from the card reader to a new file on the server, and deletes the original card reader file. The file is then imported into the database, where the commas represent a new column, and a new row is a new row in SQL. Once the information has been moved into the database it is now possible to write code to query the information and return specific results based on the user's input. The GUI

allowed the user to choose start and end dates as well as what type or types of exercises to return information from. This information was inputted as variables from the GUI into a C# function. C# can use SQL statements to manipulate a SQL database. To query a database a SELECT command is used to the database, `SELECT * FROM tableName WHERE field1 > X ORDER BY field2;` Is an example of a simple SELECT command. That command will return all fields from all results from the table `tableName` when `field1` is greater than `X`, also ordering the results by the 2nd field. In C# we would make the fields and values the variables so we could make functions to actively query the database.

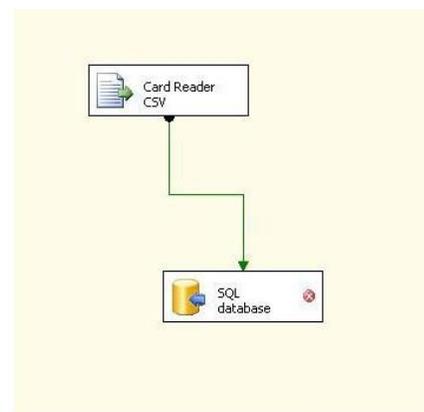


Figure 3: Data flow to server

A problem we encountered with the card readers is that people would be able to perform exercises that would not necessarily have a dedicated card reader. Weight lifting exercises such as dead lift or the shoulder press wouldn't necessarily have a certain area of the gym to enter this information. We determined to enter information like this we would add that functionality into the GUI. Adding information into SQL is similar to querying. The user would

enter the type of exercise preformed as well as the weight and repetition information that would typically be gathered by a card reader. This information is then sent over a secured connection to the database. Now the end user has the ability to query for this information. The SQL command used, `INSERT INTO tableName (field1, field2, field3) VALUES (value1, value 2, value3);` is an example of a SQL INSERT command. This will insert value1 into the table tableName in field1 and so on.

Table - dbo.CardReader		Add New Derived Table			
	ID_Number	Weight	Reps	Date	Machine_Number
▶	015242443510894	123	3	2006-04-05	1
	6015242443510894	123	3	2006-04-07	1
	6015242443510894	125	3	2006-04-09	1
	6015242443510894	125	3	2006-04-11	1
	6015242443510894	127	3	2006-04-13	1
	6015242443510894	68	3	2006-04-05	3
	6015242443510894	70	3	2006-04-07	3
	6015242443510894	70	3	2006-04-09	3
	6015242443510894	72	3	2006-04-11	3
	6015242443510894	72	3	2006-04-13	3
	-----	---	-	-----	-

Figure 4: Fitrax SQL Database

User Interface

In developing the GUI (Graphical User Interface), we used Microsoft Visual Studio .NET as our development environment and C# with Windows Forms as the programming language. As can be seen in Figure 5 below, we divided the GUI into three separate sections: Filter Options, Results, and Results Graph. In addition to these sections, we had an additional window for the "Add New Entry" button.

The Filter Options portion of the interface handled the input from the user. Parameters specified from the exercise list, start date, and end date are turned into SQL queries and sent to the SQL database. The data returned is then formatted and displayed in the interface to show the user his or her progress.

With exercises such as pushups and sit-ups, there aren't typically certain areas of the gym where these exercises are performed. Therefore, to enhance the GUI, we added an "Add New Entry" button. This button brings up a dialog that allows the user to enter in exercises that don't have a card reader and submit this data to the SQL database for future queries. After submission, the data is automatically added and will show up the next time the user generates results.

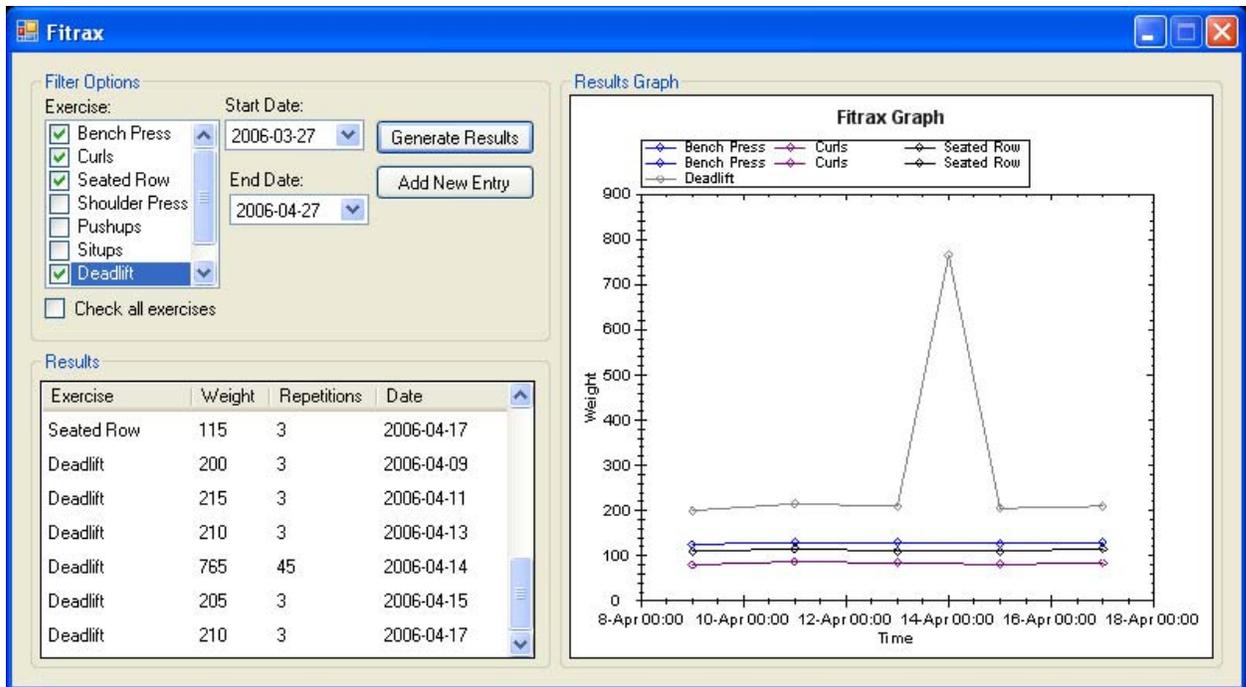


Figure 5: Fitrax User Interface

After the "Generate Results" button is pressed in the Filter Options, the table shown in the Results section of the interface is automatically updated. We programmed this table to display the appropriate results sorted first by exercise and second by date to make the data easier to read for the user. Each time the user generates results, the table is cleared and repopulated with the new data. The final portion of the user interface is the Results Graph. In programming this, we used a library that we found online called Zed Graph that allows the user to create graphs in windows application. We took the data from queries that goes to the results table and created a new line for each exercise on the graph. Each of the nodes on the lines represents a different date that the exercise was performed. To simplify things, the exercise lines were made to have different colors and were listed in a legend at the top of the graph.

Appendix 1: IBM 4779 Hybrid Smart Card Device

Input methods - Keypad -The most flexible method of entering data is certainly the keypad.

Data can be limited to just numeric with or without a decimal point, or data can include all alphanumeric characters including some special characters and even spaces. Keypad input also provides a backspace/erase key to correct the input before completing the transaction.

Magnetic Stripe - Low energy Magnetic striped badges, similar to common credit cards, provide consistent and reliable data input. Magnetic striped photo ID badges are normally used for employee identification and provide accurate input of the employee ID number into the terminal. Magnetic striped cards can also be recycled, re-encoded, and used again.

Bar Code - Bar codes are becoming more and more the industry standard for quick and accurate entry of data. Bar code photo ID badges are normally used for employee identification as well as provide accurate input of the employee ID number into the terminal. Bar Codes are easily printed on many computer laser printers and can be affixed to most anything such as work orders, inventory parts, equipment/machines, and more.

Computer Communication

Direct RS232 - Every terminal comes with a serial port. The serial port may be directly connected to the serial port on your host computer providing immediate access at any time. This is the most common communication connection. The terminal may be hundreds of feet away, depending upon the RS232 driver chip in your host computer. The Transaction Print option allows the terminal to be hooked to a serial printer allowing a supervisor to print out transaction lists at the terminal.

Appendix 2: Card Reader Hex Code

The following pages show the code used by the card reader to display messages on the LCD display.

```
#Write "Press # to start" to lcd display
020002005072657373202320746f207374617274 #Wait for user to press
# 03000003 #prompt user to insert card 200101 #Writes
0205000057454c434f4d45 #Welcome 02060100544f #To
02070200464954524158 #Fitrax #arm the magnetic reader, track 2
0402 #read the magnetic stripe, track 2 0602 #following pattern
clears the lcd screen 020000002020202020202020202020202020202020202020
020001002020202020202020202020202020202020202020202020202020
02000200202020202020202020202020202020202020202020202020 #Prompts user, writes
"Enter set info, #how much weight, Press # when done"
02000000456e7465722073657420696e666f
02000100486f77206d7563682077656966768743f
0200020050726573732023207768656e20646f6e65 #read the user's input
03000003 #clear the screen
02000100202020202020202020202020202020202020202020202020
02030000202020202020202020202020202020202020202020202020 #Prompts user for weight
02000000486f77206d616e7920726570733f #reads user's input 03000003
#clears the screen 0200000020202020202020202020202020202020202020202020
02000100202020202020202020202020202020202020202020202020
02000200202020202020202020202020202020202020202020202020
02000300202020202020202020202020202020202020202020202020 #Writes
0204010044617461207361766564 #Thank you
020502005468616e6b20796f7521 #data saved #Eject card
#clear screen for next user
02000000202020202020202020202020202020202020202020202020
02000100202020202020202020202020202020202020202020202020
02000200202020202020202020202020202020202020202020202020
02000300202020202020202020202020202020202020202020202020
```

Appendix 3: Card Reader C Code

The following pages show the code used to interact with the card reader and insert data into the SQL database.

Appendix 4: User Interface Code

The following pages show samples of the main sections of the C# code used to create the user interface.

```
/*-----*/
-----*/
/* */
/* This program accepts any hex string from a text file and

    */
/* sends the string to the 4779 device as a command. It then waits for
*/
/* the response from the device and displays the message. Based on the
*/
/* command being issued it will store data from the cardreader's
```

```

                */
/* response if necessary
*/
/*

                */
/* Written by Todd Arnold, July 11, 1995 */
/* Modified by Mark Marik, 4-12-96 */
/* Modified by Brian Conley, Matthew Kiebish 3/16/05

                */
/* */
/*-----
-----*/

#include
<windows.h>
#include
<process.h>
#include <stdio.h>
#include <search.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>

/*

* W i n d o w s M i s c e l l a n e o u s

*/
HANDLE KbdHandle;
HANDLE VioHandle;
LPWIN32_FIND_DATA FileInfo;

#include
<stddef.h>
#include
<fcntl.h>
#include
<share.h>
#include
<io.h>
#include
<dos.h>
#include
<ctype.h>
#include
<conio.h>

```

```

typedef unsigned int
UINT;
typedef unsigned
char UCHAR;
typedef unsigned
char byte;

#define      1
TRUE
#define FALSE 0
#define      200           // Max length of a line in the test file
MAX_LINE_LEN
#define      1           // Bad input caused program to abort
EXIT_FAILURE

/*-----*/
-----*/
/*      Function          prototypes          */
/*-----*/
-----*/

USHORT APIENTRY EftOpen      (HANDLE *);    /* handle pointer */
USHORT APIENTRY EftClose    (HANDLE);      /* device handle */
USHORT APIENTRY EftRead     (HANDLE,
                             PVOID,
                             USHORT,
                             PVOID);
USHORT APIENTRY EftWrite    (HANDLE,
                             PVOID,
                             USHORT,
                             PVOID);

USHORT length2;
void DoEftRead(byte *request_msg);

void disp_block(byte *indata, char *heading,
UINT num_bytes);
void issue_4778SA_command(byte *request_msg,
UINT msg_length);
void ReadId(char *eftData);
void WriteSetInfo(UCHAR location, char
*eftData);
void InsertDate();
void etas();
void minas();
void mera();
UCHAR digit_convert(char ch);
UCHAR hex2byte(char *hex_in);
UCHAR display_char(UCHAR ch);
UCHAR xchar2byte(char inchar);
UCHAR digit_convert(char ch);

```

```

int is_hex(char *in_str);
int all_hex_chars(char *instring);
UINT hex2int(char *in_str);
byte tbh2byte(char *inbyte);

/*-----*/
-----*/
/* 4779 data variables and buffers */
/*-----*/
-----*/
HANDLE EftHandle = -1;
HANDLE TempEftHandle = -1;
// our output file to store all the data from the cardreader
UCHAR outputFile[33]="C:\\CardReader\\Repository\\Data.csv";
UCHAR eftdata[512]; //
UCHAR eftdata2[512];
UCHAR request_msg[300]; // Allow a huge buffer for MACing
char buf[1024]; // Buffer for stream input and output
int tot_bytes_written; // Total bytes written to device from
test case

/*-----*/
-----*/
/* Strings corresponding to each of the 4779 return codes.
*/
/*-----*/
-----*/
char *rc_strings[] = { "No errors",

                        "Unknown
                        command",
                        "Data length
                        error",
                        "Unimplemente
                        d command",
                        "Invalid
                        value",

                        "Hardware
                        error",
                        "Sequence
                        error",
                        "Media error",
                        "Bad key",
                        "Bad password",
                        "Security
                        processor
                        error",
                        "Motor
                        timeout",
                        "Abort",
                        "Invalid PIN",

```

```
"Feature Not
Available" };
```

```
/*-----*/
-----*/
```

```
/* Main function. */
/*
/* This routine will read the test file name from the command line,
/* open that test file, read its contents, check for valid data within
/* the file, then send the command to the device.
/*
/* */
```

```
/*-----*/
-----*/
```

```
void main(int argc, char *argv[])
{
    UINT i; // For loop index variable
    UINT input_chars; // No. of chars
    read from a line in
    the test file
    UINT msg_bytes; // Half the number of
    ASCII bytes

    in test case file
    FILE *stream; // File ptr for test file
    char line[MAX_LINE_LEN]; // Max len of a line
    char converted_line[MAX_LINE_LEN/2+1]; // Max len of a
    converted line
    UINT line_num; // Current line no. being processde

    in test file
    UINT tot_input_chars; // Total no. of input chars read

    from file
    UCHAR *request_msg_ptr; // Offset into request_msg
    buffer
    UCHAR cmd_in_progress; // Command in progress flag

    UINT test_input = 7;

    // Verify the user gave us a command line argument

    int z = 0;
    while(z<1)
    {
```

```

    if (argc < 2) {
        printf("You did not enter a test file to read and send
        to the 4779.\n\n");
        exit(EXIT_FAILURE);
    }

    if((stream = fopen(argv[1],"rt")) != NULL) // Open
file as R/O
text

    { //
        setvbuf(stream,buf,_IOFBF,sizeof(buf)); //
        printf("Reading in test file: %s\n",argv[1]); //
        Display the test

file being read in
        line_num = 1; // Process
        first line
        number

            tot_input_chars = 0; //
        Initialize total no.
        of chars read from file
        tot_bytes_written = 0; //
        Initialize total no.

of bytes sent to device
        memset(request_msg,0,sizeof(request_msg)); // Clear out
        the buffer
        request_msg_ptr = request_msg; // Initialize offset

pointer
        cmd_in_progre
        ss = FALSE;

//-----
        -----//

// STOP AT END OF FILE //
// Test case file must have data start in column 1. //
// Test case data must be hexadecimal. //
// Each line my be up to MAX_LINE_LEN bytes long. //
// A blank line indicates the end of a command. //

// A new command may start in the line following the
blank line. //
//-----
        -----//
        while(fgets(line,MAX_LINE_LEN,stream) != NULL) // Get
        whole line until

```

```

EOF
    { // fgets inserts the
new line char if found
    //-----
    -----//
    // Check for blank line //
    //-----
    -----//
    if(!isspace(line[0])) // If not a blank line,

        process the line
            { //
            cmd_in_progress
            = TRUE; //
            Indicate
            command in
            progress
            if(line[strlen(
            line)-1]=='\n')
            // If the new
            line
            character
            exists,
            line[strlen(lin
            e)-1]='\0'; //
            then get rid of
            it
            //
            printf("\tLine
            %d:
            %s\n",line_num,
            line); //

            //-----
            -----//
            // Make sure line has just hex characters 0-9,
            a-f, or A-F //
            //-----
            -----//
            if (!all_hex_chars(line)) {

                printf("Line %d in %s
contains non-hex
characters.\n",line_num,argv[1]);

                exit(EXIT_FAILURE);
            }
            input_chars = strlen(line);
            msg_bytes = input_chars/2;
            tot_input_chars += msg_bytes;

```

```

//-----
-----//
// Make sure the message isn't too big for our
buffer //
//-----
-----//
if (tot_input_chars > 2*sizeof(request_msg)) {

printf("Your command string is too long
for my input buffer.\n");
exit(EXIT_FAILURE);
}

//-----
-----//
// Make sure line has even number of digits,
which is required //

// since each byte is represented by two hex
digits. //

//-----
-----//

if ((2*msg_bytes) != input_chars) {

printf("Line %d in
%s is not a multiple of two
chars in
length.\n",line_num,argv[1]);
exit(EXIT_FAILURE);
}

//-----
-----
//
// Convert message data from ASCII to hex, in
our request buffer
//
//-----
-----
//

memset(converted_line,0,sizeof(converted_line));

for (i=0; i<msg_bytes; i++)
converted_line[i] =
tbh2byte(line+(2*i));

```

```

for (i=0; i<msg_bytes; i++)
    request_msg_ptr[i] = converted_line[i];

    request_msg_ptr          +=          msg_bytes;

number    line_num++;      //      Increment      line
          of              test          file
    }

else {

number    line_num++;      //      Increment      line
          of              test          file

a        if(cmd_in_progress == TRUE) { // If done with
command  from              test          case

//-----
--//

// Send the data to the 4779 as a
request  message
//

//-----
--//

    issue_4778SA_command(request_msg,
        tot_input_chars);

total    tot_input_chars  =    0;    //      Initialize
no.      of              chars    read    from    file

the buffer  memset(request_msg,0,sizeof(request_msg));    // Clear out

request_msg_ptr = request_msg;    // Initialize

offset pointer
        cmd_in_progress = FALSE;
    }
}
}

```

```
        if(cmd_in_progress == TRUE) { // If done with a command
from test case

        //-----
        -----//
        // Send the data to the 4779 as a request message
        //
        //-----
        -----//
        issue_4778SA_command(request_msg,
tot_input_chars);

        cmd_in_progress = FALSE;
    }

}

e
l
s
e

p
r
i
n
t
f
(
"
C
o
u
l
d

n
o
t

o
p
e
n

%
s
\
n
"
,

```



```

rc = EftRead ( EftHandle, &eftdata, sizeof(eftdata) ,
              &length ) ;

    } while ( !length ) ;

if(length<1)
    printf("\nResponse length is
    invalid. Perhaps your device
    driver is not
    installed\n");

else {
    printf("\n4779 response: \n") ;
    printf(" Return code is %02X (%s).\n",
           eftdata[0], rc_strings[eftdata[0]]);
    printf("%d bytes of data were returned.\n", length-1);
    disp_block(eftdata+1, " ", length-1);
    }

//request msg of 06 means magnetic read, we want to read in
    the ID #

    if(request_msg[00] == 06)
        ReadId(eftdata);

//request msg of 03 means read keypad,
we want to store their input
if (request_msg[00] == 03)
    WriteSetInfo(outputFile, eftdata);

    if(eftdata[0]!=0) {
        printf("Hit enter to continue");
        getchar();
    }
}

// The ReadId function is passed a char pointer referring to
the eftData buffer

```

```

and
// it's purpose is to store the user's ID number from the
Husky card in the
output file
void ReadId(char *eftData)
{
    FILE *f;
    UCHAR cardId[17]; // char array to store the id #
    int i, j;

    // this for loop
    populates the char
    array cardId with the
    data from the
    reader
    for(i = 2; i <= 17;
    i++){
    cardId[i - 2] =
    eftdata[i];
    }

    cardId[16] = NULL;

    f = fopen(outputFile, "a");
    if(f==NULL)
        printf("No good");

        fprintf(f, "%s,", cardId);
        fclose(f);

        //clear
        out the eftdata
        buffer so no part
        of the result is
        used
        later
        for ( i = 2; i <=
        17; i++){
            eftdata[i]=
            NULL;
        }
    }

// The WriteSetInfo function takes a location that
refers to a file and
// a char pointer referring to the eftData buffer.
It's job is to store

```

```

// the user's input from the cardreader.
void WriteSetInfo(UCHAR location, char *eftData)
{
    FILE *f;
    UCHAR setInfo [4];
    int i;
    f = fopen(outputFile, "a");
    if(f==NULL)

        printf("No good");

    i = 0;
    //populate the setInfo array
    with the response from the
    cardreader in the
    buffer

    while (eftData[i+1] != NULL){
        setInfo[i] = eftData[i+1];
        i++;
    }
    //set the part of the setInfo array not used to null so
    outputting to the

file
    //doesn't have
    unwanted characters
    setInfo[i] = NULL;
    if (setInfo[3] !=
    NULL);

        setInfo[3] = NULL;

    //print the setInfo array in the output file
    fprintf(f, "%s,", setInfo);
    fclose(f);

    //clear out setInfo array
    for(i=0; i<=3; i++){
        setInfo[i] = NULL;
    }

    //clear out the eftData buffer so that no part of the first
    response is

used in the second
    i = 1;
    while(eftData[i
    ] != NULL){

```

```

    eftData[i] = NULL;
    i++;
}

}

// The InsertDate function is designed to store the date of
// the user's workout
// For the construction of our database we thought it best
// to have machine
// number as a field, however we were only using 1 machine,
// so for the purpose
// of the demonstration we hardcoded in the machine number
// after the date, 1
// in this case
void InsertDate()
{
    FILE *f;
    time_t tval;
    struct tm *now;
    tval = time(NULL);
    now = localtime(&tval);
    f = fopen(outputFile, "a");

    // Print the date in
    // the output file, we have
    // to use +1900 for the year
    // because
    // the function returns
    // the number of years past
    // 1900, not the current
    // year
    fprintf(f, "%d/%02d/%02d,1
\n", now->tm_mon+1, now-
>tm_mday, now-
>tm_year+1900);
    fclose(f);
}

/*-----
-----*/

/*
/* Issue the passed string as a command to the 4779, then wait for a
/* response message and display it.
/*
/*
/*-----
-----*/

```

```

-----*/

void issue_4778SA_command(byte *request_msg, unsigned int
msg_length)

{
    unsigned
    int i;
    USHORT
    bytes_writ
    ten;
    int rc;

    // open the device driver

    rc = EftOpen(&TempEftHandle);

    if (rc == 0)
    {
        EftHandle = TempEftHandle;
    }
    else
    {
        printf("Could not open IBMEFT$");
        exit(EXIT_FAILURE);
    }

    // Issue the passed string as a command to the 4779

    printf("\nSending command to the 4779. Command string
is:\n " ) ;
    for (i=0; i<msg_length; i++) printf("%02X",
request_msg[i]);
    printf("\n");

    printf("Message length = %d\n",msg_length);

    rc = EftWrite(EftHandle, request_msg, msg_length,
&bytes_written) ;

```

```

if((bytes_written == -1) || (rc != 0)) {
    printf("Write error to device driver\n");
    printf("Hit enter to continue");
    getchar();
    exit(EXIT_FAILURE);
}

else {
    printf("Bytes written = %d",bytes_written);
    tot_bytes_written += bytes_written;
}
//modified DoEftRead function to pass it the request msg
DoEftRead(request_msg) ; // Now read the
response

    EftClose(EftHandle);
}

/*-----*/
-----*/

/*
/* Input is an ASCII representation of a hex digit - a character between
'0'
/* and '9', or 'A' to 'F' (case is ignored). Output is the numeric
/* equivalent, returned in a byte (unsigned char).
/*
/*
/*-----*/
-----*/
unsigned char xchar2byte(char inchar)
{
    if ((inchar >= '0') && (inchar <= '9')) /*
        Numeric 0-9 */
        return((byte) (inchar-'0'));
    else /* Char A-F */
        return((byte)10 + (byte) (toupper(inchar)-
            'A'));
}

/*-----*/
-----*/
/* */
/* Convert an ASCII representation of a byte into the
numeric equivalent. */

```

```

/* The input is a two-character string, where each character
is either '0' */
/* through '9', or 'A' through 'F' (case ignored). */
/* */
/*-----*/
-----*/
byte tbh2byte(char *inbyte)
{
    return(xchar2byte(inbyte[1]) +
16*(xchar2byte(inbyte[0])));
}

/*-----*/
-----*/
/* */
/* Verify that each character in the passed string is a
valid "hex digit", */
/* meaning that it is "0"- "9", "a"- "f", or "A"- "F". */
/* */
/*-----*/
-----*/
int all_hex_chars(char *instring)
{
    UINT i;
    int return_value = TRUE;

    for (i=0; i<strlen(instring); i++)

        if (!isxdigit(instring[i])) {
            printf("Character %c in column %u is not
hexidecimal\n",instring[i],i);
            return_value = FALSE;
        }

    return
(return_value)
;
}

/*-----*/
-----*/
/* */
/* This function converts a character representing a hex
digit into the */

```

```

/* corresponding numeric value. Only values '0'-'9', 'a'-
'f', and 'A'-'F' */
/* are permitted - the result is zero for any others. */
/*-----*/
-----*/

```

```

unsigned char digit_convert(char
ch)
{
if (('0'<=ch) & (ch<='9')) /* If
character is '0'-'9'... */
return (ch-'0');
else if (('A'<=ch) & (ch<='F')) /*
If character is 'A'-'F'... */
return ((UCHAR)0xA+ch-'A');
else if (('a'<=ch) & (ch<='f')) /*
If character is 'a'-'f'... */
return ((UCHAR)0xA+ch-'a');
else return (0); /* If character is
invalid... */
}

```

```

/*-----*/
-----*/
/*
/* This function converts a two character string
representing a byte of
/* data into the corresponding numeric value. Both
characters of the
/* string must be '0'-'9', 'A'-'F', or 'a'-'f'.
/*-----*/
-----*/

```

```

unsigned char hex2byte(char *hex_in)
{
char ch;
unsigned char out_val;

out_val = (digit_convert(ch=*hex_in)) << 4;
out_val = (digit_convert(ch=*(hex_in+1))) + out_val;

return
(out_val)

```

```

;
}

/*-----
-----*/

/*
/* This function returns 1 if all characters in the passed string are
/* valid hex digit representations ('0'-'9', 'a'-'f', 'A'-'F'). It
/* returns 0 if one or more characters in the string are not.
/*
*/

/*-----
-----*/

int
is_hex(char
*in_str)
{

unsigned int index; /* Index for chars. in the string */
    int return_value; /* Function result */
char ch; /* Temp. character variable */

    return_value = 1; /* Assume it's OK for now */

for (index=0;
in_str[index]!=0; index++)
/* Step through characters
*/
{
ch=in_str[index]; /* Get the
current character */

    if ( !((('0'<=ch) & (ch<='9')) /* See if current
char. is valid */
& !((('a'<=ch) & (ch<='f'))
& !((('A'<=ch) & (ch<='F')))) return_value = 0;

}

return
(return_value

```

```
};
```

```
/*-----  
-----*/
```

```
/*  
/* This function accepts a four character string containing characters */  
/* representing hex digits, and converts the entire string into an */  
/* unsigned integer value. */  
/* */
```

```
/*-----  
-----*/
```

```
unsigned int hex2int(char *in_str)
```

```
{  
    unsigned  
    int  
    temp_int;  
    char  
    *str_ptr;
```

```
    temp_int = hex2byte(str_ptr = in_str) << 8;  
    temp_int = temp_int + hex2byte(str_ptr += 2);  
    return (temp_int);
```

```
}
```

```
/*-----  
-----*/
```

```
/* */  
/* This function displays num_bytes bytes of data in both  
hex and ASCII */  
/* formats. 16 bytes of data are displayed on each line (or  
less, if there */  
/* are not 16 bytes to display). String "heading" is  
displayed at the front */  
/* of every line - a typical heading would simply be  
indentation spaces. */
```

```
/* */  
/*-----  
-----*/
```

```
void disp_block(byte *indata, char *heading, unsigned int
```

```

num_bytes)
{
    byte *pointer;
    unsigned int
    index;
    char ch;
    byte bytes;
    byte seg1_bytes,
    pad_spaces;
    unsigned int
    total_blocks;
    unsigned int
    block_cnt;
    unsigned int
    bytes_in_block;

    total_blocks = num_bytes / 16;
    if ((num_bytes % 16) != 0) ++total_blocks;

    pointer = indata; /* Assign the pointer */

    for (block_cnt=1; block_cnt<=total_blocks; block_cnt++)
    {
        if (block_cnt == total_blocks) /* Last block? */
        {
            bytes_in_block = num_bytes % 16;
            if (bytes_in_block == 0) bytes_in_block = 16; /* If
            mult. of 16... */
        }
        else bytes_in_block = 16;

        if (bytes_in_block > 16) bytes = 16;
        else bytes = (byte)bytes_in_block;

        if (bytes > 8)
        {
            seg1_bytes = 8;
            pad_spaces = (byte) (3*(16-(UINT)bytes));
        }
        else
        {
            seg1_bytes = bytes;

```

```

pad_spaces = (byte) (3*(16-(UINT)bytes));
}

printf("%s",heading);

for (index=0; index<seg1_bytes; index++) printf("%02X
",*(pointer+index));
printf(" ");
for (; index<bytes; index++) printf("%02X
",*(pointer+index));

for (index=0; index<pad_spaces; index++) printf(" ");
printf(" *");

for (index=0; index<seg1_bytes; index++)
printf("%c",display_char(ch=*(po
inter+index)));
printf(" ");
for (; index<bytes; index++)
printf("%c",display_char(ch=*(pointer+index)));

for (index=0; index < (UINT) (16-bytes); index++)
printf(" ");
printf("*\n");

if
(block_cnt !=
total_blocks)
pointer += 16;
}

/*-----
-----*/

/*
/* This function returns a displayable value for a character passed      If
in.
/* the character is in the range X'20'-X'7F', it returns the          */
character.
/* Otherwise, it returns a period.                                     */
/*
*/
*/

```

```
/*-----  
-----*/
```

```
unsigned char display_char(unsigned char ch)
```

```
{  
  if ((ch >= 0x20) && (ch <=  
    0x7F)) return (ch);  
  else return ('.');
```

```
}
```

```
* 1 - Bench Press
* 2 - Curls
* 3 - Seated Row
* 4 - Triceps *
*/
```

```
using System;
using
System.Collections.Generic;
using
System.ComponentModel;
using System.Data;
using
System.Data.SqlClient;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.IO;
using ZedGraph;
```

```
namespace Fitrax
```

```
{
    public partial class Form1
    : Form
    {

        Form2 aForm = new Form2();
        Form3 loginForm = new Form3();

        public Form1()
        {
            InitializeComponent();
            this.Reset();
        }

        // Initialize
        public void Reset()
        {

            startDate.Format =
            DateTimePickerFormat.Custom;
            startDate.CustomFormat = "yyyy-MM-dd";
            endDate.Format =
            DateTimePickerFormat.Custom;
            endDate.CustomFormat = "yyyy-MM-dd";

            loginForm.Show();
        }

        // Open Add Entry Dialog
        private void addEntryBtn_Click(object sender,
        EventArgs e)
```

```

    {
aForm.userID = loginForm.userID.ToString();
aForm.Show();
}

        // Check or uncheck all exercises
        private void checkExercises_CheckedChanged(object sender,
        EventArgs e)
        {

if (checkExercises.Checked == true)
    for (int i = 0; i < exercisesList.Items.Count; i++)
        exercisesList.SetItemChecked(i, true);
    else

        for (int i = 0; i < exercisesList.Items.Count; i++)
            exercisesList.SetItemChecked(i, false);
        }

        private void
resultsGrid_CellContentClick(object sender,
DataGridViewCellEventArgs e)
{

    }

        private void startDate_ValueChanged(object sender,
        EventArgs e)
        {

    }

        private void endDate_ValueChanged(object sender,
        EventArgs e)
        {

    }

        // Generate results based on parameters
        private void resultsBtn_Click(object sender,
        EventArgs e)
        {

            string IDNum =
loginForm.userID.ToString();//"6015242443510894";

            this.resultsGrid.Rows.Clear();

foreach (object exerciseChecked in
exercisesList.CheckedItems)
{
string machineID;

            switch
(exerciseChecked.ToString())
            {

```

```

        case "Bench
            Press":
            machineID =
                "1";
            break;

        case "Curls":
            machineID =
                "2";
            break;

        case "Seated
            Row":
            machineID =
                "3";
            break;

        case "Shoulder
            Press":
            machineID =
                "4";
            break;

        case "Pushups":
            machineID =
                "5";
            break;

        case "Situps":
            machineID =
                "6";
            break;

        case "Deadlift":
            machineID =
                "7";
            break;

        case "Squats":
            machineID =
                "8";
            break;

        default:
            machineI
            D = "0";
            break;
    }

    string mySelectQuery = "SELECT * FROM CardReader WHERE
    ID_Number

= " + IDNum.ToString();
    mySelectQuery += " AND Date >= '" +

```

```

        startDate.Text + "'";
        mySelectQuery += " AND Date <= '" + endDate.Text
        + "'";
        mySelectQuery += " AND Machine_Number = " +
        machineID;

        SqlConnection myConnection = new SqlConnection("Data
Source=192.168.2.5;" +
        "Initial
Catalog=CardReader;" +
        "User ID=UserA;" +
        "Password=Glr@ffe;");

        String[, ,] graphData = new String[10,
        2, 99];
        int nGraphDataCount = 0;
        int i1 = 0;
        int i2 = 0;
        int i3 = 0;
        int i4 = 0;
        int i5 = 0;
        int i6 = 0;
        int i7 = 0;
        int i8 = 0;

        PointPairList bench = new PointPairList();

        SqlCommand myCommand = new SqlCommand(mySelectQuery,
myConnection);
        myConnection.Open();
        SqlDataReader myReader;
        myReader =
        myCommand.ExecuteReader();
        string machineName;

        while (myReader.Read())
        {
            machineName = myReader.GetString(4);

            switch
            (machineName
            )
            {

                case "1":
                    machineID = "Bench
                    Press";
                    break;

                case "2":
                    machineID =
                    "Curls";
                    break;
            }
        }
    }
}

```

```

        case "3":
            machineID =
                "Seated Row";
            break;

        case "4":
            machineID = "Shoulder
                Press";
            break;

        case "5":
            machineID =
                "Pushups";

            break;

        case "6":
            machineID =
                "Situps";
            break;

        case "7":
            machineID =
                "Deadlift";
            break;

        case "8":
            machineID =
                "Squats";
            break;

        default:
            machineID =
                "Other";
            break;
    }

    string[] row0 = { machineID,
myReader.GetString(1),
myReader.GetString(2), myReader.GetString(3) };

    {
        DataGridViewRowCollection rows =
            this.resultsGrid.Rows;
        rows.Add(row0);

        int nExerciseNum =
            myReader.GetString(4).ToString()[0];
        nExerciseNum -= 48;

        switch (nExerciseNum)
        {
            case 1:
                graphData[nExerciseNum,
                    0, i1] =
                    myReader.GetString(3);
                graphData[nExerciseNum,

```

```

        1, i1] =
myReader.GetString(1);
        i
        1
        +
        +
        ;

        b
        r
        e
        a
        k
        ;

case 2:
graphData [nExerciseNum,
0, i2] =
myReader.GetString(3);
graphData [nExerciseNum,
1, i2] =

myReader.GetString(1);
        i
        2
        +
        +
        ;

        b
        r
        e
        a
        k
        ;

case 3:
graphData [nExerciseNum,
0, i3] =
myReader.GetString(3);
graphData [nExerciseNum,
1, i3] =

myReader.GetString(1);
        i
        3
        +
        +
        ;

        b
        r
        e

```

```
a  
k  
;
```

```
case 4:
```

```
graphData[nExerciseNum,  
0, i4] =  
myReader.GetString(3);  
graphData[nExerciseNum,  
1, i4] =
```

```
myReader.GetString(1);
```

```
i  
4  
+  
+  
;
```

```
b  
r  
e  
a  
k  
;
```

```
case 5:
```

```
graphData[nExerciseNum, 0, i5] =  
myReader.GetString(3);
```

```
graphData[nExerciseNum, 1, i5] =
```

```
myReader.GetString(1);
```

```
i  
5  
+  
+  
;
```

```
b  
r  
e  
a  
k  
;
```

```
case 6:
```

```
graphData[nExerciseNum,  
0, i6] =  
myReader.GetString(3);  
graphData[nExerciseNum,  
1, i6] =
```

```
myReader.GetString(1);
```

```
i
```

```
6  
+  
+  
;  
  
b  
r  
e  
a  
k  
;
```

```
case 7:  
graphData [nExerciseNum,  
0, i7] =  
myReader.GetString(3);  
graphData [nExerciseNum,  
1, i7] =
```

```
myReader.GetString(1);
```

```
i  
7  
+  
+  
;  
  
b  
r  
e  
a  
k  
;
```

```
case 8:  
graphData [nExerciseNum,  
0, i8] =  
myReader.GetString(3);  
graphData [nExerciseNum,  
1, i8] =
```

```
myReader.GetString(1);
```

```
i  
8  
+  
+  
;  
  
b  
r  
e  
a  
k  
;
```

```

    }
}
}

CreateGraph(zedGraphControl1, graphData);

myReader.Close();
myConnection.Close();
}

        StreamWriter writer = new StreamWriter("Results.txt");
        writer.WriteLine("Start Date: " + startDate.Text);
        writer.WriteLine("End Date: " + endDate.Text);
        foreach (object exerciseChecked in
            exercisesList.CheckedItems)

            writer.WriteLine(exerciseChecked.ToString());
writer.Close();
}

        private void
        exercisesList_SelectedIndexChanged(object sender,
        EventArgs
        e)
        {

            }

        private void zedGraphControl1_Load_1(object sender,
        EventArgs e)
        {

            }

private void Form1_Load(object sender,
EventArgs e)
{
// Setup the graph

        // CreateGraph(zedGraphControl1,
        string[]);
        // Size the control to fill the form with a
        margin
        SetSize();

    }

private void Form1_Resize(object sender,
EventArgs e)
{
SetSize();
}

        private void SetSize()

```

```

{
    zedGraphControll1.Location = new Point(360, 10);
    // Leave a small margin around the outside of the
    control
    zedGraphControll1.Size = new
    Size(ClientRectangle.Width - 20,

                                                ClientRectangle.Height
- 20);
}

// Build the Chart
private void CreateGraph(ZedGraphControl zg1, String[, ]
graphData)
{

    // get a reference to the GraphPane
    int i = 0;
    GraphPane myPane = zg1.GraphPane;
    //zg1.MasterPane.Add(myPane);
    // Set the Titles
    myPane.Title = "Fitrax Graph";
    myPane.XAxis.Title = "Time";
    myPane.YAxis.Title = "Weight";

    // Make up some data arrays based on the Sine
    function
    double x, y1, y2;
    PointPairList list1 = new PointPairList();
    PointPairList list2 = new PointPairList();
    PointPairList list3 = new PointPairList();
    PointPairList list4 = new PointPairList();
    PointPairList list5 = new PointPairList();
    PointPairList list6 = new PointPairList();
    PointPairList list7 = new PointPairList();
    PointPairList list8 = new PointPairList();
    PointPairList list9 = new PointPairList();

    // list1.Add(2, 1);
    for (int j = 1; j < 10; j++)
    {

        while (graphData[j, 0, i] != null)

            {

                // x = graphData[0, i];
                //x = (double)new
                XDate(graphData[0, i]);

                string xDate1 = graphData[j, 0, i];
                int xYear1 = (((xDate1[0] - 48) * 1000) +
                ((xDate1[1] - 48)
* 100) + ((xDate1[2] - 48) * 10) + (xDate1[3] - 48)); //int xnYear =
                xYear1;
                int xYear2 = (((xDate1[5] - 48) * 10) + (xDate1[06]
- 48));

```

```

        int xYear3 = (((xDate1[8] - 48) * 10) + (xDate1[9] -
48));

    x = (double)new XDate(xYear1, xYear2, xYear3);
    y1 = Int32.Parse(graphData[j, 1, i]);

    switch (j)
    {

        case 1:
            list1.Add(x, y1);
            break;

        case 2:
            list2.Add(x, y1);
            break;

        case 3:
            list3.Add(x, y1);
            break;

        case 4:
            list4.Add(x, y1);
            break;

        case 5:
            list5.Add(x, y1);
            break;

        case 6:
            list6.Add(x, y1);
            break;

        case 7:
            list7.Add(x, y1);
            break;

        case 8:
            list8.Add(x, y1);
            break;

    }

    i++;
}

/* for (int i = 0; i < 6; i++)

    {
        x = (double)i + 5;
        y1 = 1.5 + Math.Sin((double)i * 0.2);
        y2 = 3.0 * (1.5 + Math.Sin((double)i * 0.2));
        list1.Add(x, y1);
        list2.Add(x, y2);

    }

```

```

*/
//Set the XAxis to date type

// Generate a red curve with diamond
// symbols, and "Porsche" in the legend
if (graphData[j, 0, 0] != null)
{

    switch (j)
    {
    case 1:
        LineItem myCurve1 =
        myPane.AddCurve("Bench Press",
        list1, Color.Blue,
        SymbolType.Diamond);
        break;

        case 2:
            LineItem
            myCurve2 =
            myPane.AddCurve("Curls",
            list2,
            Color.Purple,
            SymbolType.Diamond);
            break;
            case 3:
                LineItem
                myCurve3 =
                myPane.AddCurve("Seated
                Row",
                list3,
                Color.Black,
                SymbolType.Diamond);
                break;
                case 4:
                    LineItem
                    myCurve4 =
                    myPane.AddCurve("Shoulder
                    Press",
                    list4,
                    Color.Red,
                    SymbolType.Diamond);
                    break;
                    case 5:
                        LineItem
                        myCurve5 =
                        myPane.AddCurve("Pushups",
                        list5,
                        Color.Orange,

```

```

        SymbolType.Dia
        mond);
        break;
        case 6:
        LineItem
        myCurve6 =
        myPane.AddCurv
        e("Situps",
        list6,
        Color.Green,
        SymbolType.Dia
        mond);
        break;
        case 7:
        LineItem
        myCurve7 =
        myPane.AddCurv
        e("Deadlift",
        list7,
        Color.Gray,
        SymbolType.Dia
        mond);
        break;
        case 8:
        LineItem
        myCurve8 =
        myPane.AddCurv
        e("Squats",
        list8,
        Color.Brown,
        SymbolType.Dia
        mond);
        break;
    }
}

i = 0;

}
// Generate a blue curve with circle
// symbols, and "Piper" in the legend
// LineItem myCurve2 =
myPane.AddCurve("Machine 2",
// list2, Color.Blue, SymbolType.Circle);

// Tell ZedGraph to refigure the
// axes since the data have changed
myPane.XAxis.Type = AxisType.Date;
zgl.AxisChange();

this.Refresh();
}
}
}
}

```