WinCITDll Specification and Test Program Notes
June 17, 2004

We created a DLL that contains all of the CIT handwriting recognition code and makes CIT recognition easily accessible through a small number of functions. The CIT code requires training data which creates user specific handwriting profiles. This training data can be generated either with the CIT Training tool or through the use of training calls in the DLL. In either case, the PostTrain program from CIT is required in order to incorporate training data into a profile.

The Function calls to the DLL are as follows:

Test Functions

```
int TestSquare(int input)
```

A trivial test case which returns the square of the input number, used only to test whether the DLL is functioning.

Memoryless Functions

The following functions are memoryless, meaning that a substantial CIT computation occurs each time they are called. This makes them very slow but they are useful for testing or if the Profile (explained later) in memory cannot be touched for some reason.

```
float ProbOfChoice(int choicenum, int* xs, int* ys, int arrsize, char *path)
```

Returns the "Prob"[1] of the choicenumth to most likely choice (IE: choicenum = 0 is most likely, choicenum = 1 is second most likely). The coordinates are passed in as two different arrays, one of x coordinates and one of y coordinates. Each array is of size arrsize and the elements obviously must correspond (x[0], y[0] is the first coord pair; x[1],y[1] is the second; x[n],y[n] is the nth). The path argument refers to the path to the training directory. This function returns -1 on failure.

```
int NameOfChoice(int choicenum, int* xs, int* ys, int arrsize, char *buf, char
*path)
```

Stores the name of the choicenumth best alternative in the buf which was passed in. Returns 0 on success and -1 failure. All arguments are the same as the ProbOfChoice function.

```
float ProbOfBest(int* xs, int* ys, int arrsize, char *path)
```

Returns the "Prob" of the best match. Equivalent to calling ProbOfChoice with a choicenum of 0. Returns -1 on failure.

```
int NameOfBest(int* xs, int* ys, int arrsize, char* buf, char* path)
```

Stores the name of the best match in buf and returns 0 on success and -1 on failure. Equivalent to calling NameOfChoice with choicenum = 0;

[1] The though "Prob" is the word used by the designers of CIT is isn't truly a prob but rather a floating point value which demonstrates confidence. It may for example, be greater than 1.00.

Memory Based Functions

Although the Memoryless functions work perfectly well, they perform may actions multiple times. For example querying the names of the top 10 alternatives with the memoryless functions requires that the symbol be recognized ten times, a process which entails a substantial overhead in both the recognition computation and the disk I/O required to load the training data from disk.

The functions in this section are designed to decrease computation and disk access overhead by storing cached copies of the recognition results (the Profile) and training data recognizer globally in the DLL. The simplest version takes a single set of points, and a path to the training files and performs the recognition computation to create a global profile. The global profile can then be queried for alternative names and probabilities without the computational overhead associated with the memoryless functions.

In addition to saving the computational overhead in querying alternatives, we would also like to avoid loading training data from disk each time we wish to recognize a symbol. We do this by providing a function which preloads all recognition training data into memory in the form of a "recognizer". We can then use this global recognizer to recognize symbols and store the results in the same global profile as discussed above. Combined, the caching of the profile and training data provide a very noticeable performance increase.

`void LoadAsProfile(int* xs, int* ys, int arrsize, char* path)`
This function takes the xs and ys arrays of arrsize and the path to the training data. It then uses the CIT to recognize them and stores the output in the DLL's global profile. Calling the function a second time will overwrite the previously stored data.

`float ProfProbOfChoice(int choicenum)`
This function gets the "Prob" of the choicenumth choice using the global profile. Since it retrieves the value from a preloaded profile, `LoadAsProfile` or `LoadProfWithRec` must previously have been called with the correct parameters. It hypothetically should return -1 on an error.

`int ProfNameOfChoice(int choicenum, char *buf)`
This function takes the name of the choicenumth most likely symbol in the global profile and copies that name into the supplied buffer. Once again, this retrieves the name from a preloaded profile so `LoadAsProfile` or `LoadProfWithRec` must already have been called. It hypothetically should return -1 on an error.

`void LoadAsRecognizer(char *path)`
This function takes the training data found at the given path and uses it to create a global recognizer which stores all training data in memory. To recognize a symbol, the programmer may call `LoadProfWithRec` which creates a global profile from the recognizer in RAM (IE: she doesn't need to call `LoadAsProfile` which loads the training data from disk). Each call to this function overwrites the previous global recognizer, but not the global profile. (That means that if you add new training data to the recognizer, you have to run `LoadProfWithRec` before results in

the profile to reflect the new training data.)

`void LoadProfWithRec(int *xs, int *ys, int arrsize)`
Using the given points and the already existing global recognizer, creates a global profile using the CIT FeatureRecognizer. The profile is stored in the same global profile as `LoadAsProfile` and the results can be accessed with the same methods, `ProfProbOfChoice` and `ProfNameOfChoice`. Each call to this function overwrites the previous global profile. A recognizer must be loaded using `LoadAsRecognizer` prior to calling `LoadProfWithRec`.

Training Functions
The functions in this section are used to create training data, but the data isn't actually integrated into the recognizer until the PostTrain program is run. The training section of the DLL consists of three functions: one to add additional samples of a given symbol to a global array of samples, one to write the training file for a set of samples, and one to clear the global array of samples.

`void AddToTraining(int *xs, int *ys, int arrsize)`
Takes a set of points which represent one character (symbol) and adds them to a global array containing other samples of the same character. To train a different character, the programmer must first call `ClearTraining` to avoid confusing samples of two different characters. A maximum of 128 samples for one training character may be used. If more than 128 samples are added, the additional samples will be ignored.

`int WriteTraining(char *name, char *path)`
This function writes all of the sample symbols supplied using `AddToTraining` since the last call to `ClearTraining`. The file will be formatted so as to be compatible with the PostTrain program, will be named "(name).0" and saved in the given directory. This function does not clear the samples buffer! To begin training a new symbol, the programmer must call `ClearTraining` separately. In order for the results of the training to be useful, the PostTrain executable must be called and the training data reloaded into Recognizer and/or Profile if applicable.

`void ClearTraining()`
This function removes all samples from the global array so the user can commence training a new symbol. If the sample array wasn't saved using `WriteTraining`, it will be lost forever.

C# Test Program
In order to test the DLL and with the embedded CIT system, we used it to interface with an MS Ink application written in C#. It is a heavily modified version of the "Ink Collection Sample" program provided with the Tablet PC API.

Our test program provides a canvas for the user to drawn on and the following general features added:

1. Clear the Canvas
2. Save the present contents of the window to a gif file

3. Print out the pixel coordinates of all hardware samples of drawing in the window
4. Run the DLL Test Function to verify that the DLL works

Additionally, we provide the following recognition features:

1. *Recognize*: Recognizes what has been written using the `LoadProfWithRec` call in the Dll for the fastest possible recognition (we use both the global recognizer and profile). This menu item prints out the top ten alternatives and their “Probs” in an MS Windows Dialog.
2. *Reload Recognizer*: Reloads the global recognizer using the training data directory. In practice this is only needed if the training data was changed outside of the program. Training additional symbols inside of this program automatically call PostTrain and reloads the Recognizer.

We also provide the following training functionality:

1. *Add Present*: Adds the present drawing to the samples list by calling `AddToTraining` and clears the screen.
2. *Save Training Data*: Brings up a dialog box which asks for a name for the symbol and then saves all samples to the training directory under the given name using `WriteTraining`. It automatically calls PostTrain, reloads the recognizer (using `LoadAsRecognizer`), clears the samples array (using `ClearTraining`) and clears the screen.
3. *Run PostTrain*: Runs PostTrain in the training directory and then reloads the recognizer using `LoadAsRecognizer`. (Note: Calling PostTrain requires blocking until PostTrain is finished before calling `LoadAsRecognizer`). In practice, the user should never have to call this function unless training data is being changed outside of the program. When Training data is added and saved, PostTrain is automatically called and the recognizer is automatically reloaded.
4. *Clear All Training*: Clears all of the training samples and allows the user to start over again. Especially useful he decides not to train a symbol or realizes he has been making a symbol in several different ways.

Note:
In order to produce the data for the CIT recognizer, we use the MS InkCollector to collect a set of Strokes and extract points from these strokes. The coordinates of the individual sample points are then converted into window pixels (as opposed to MS InkSpace) and passed into the WinCitDll in the form of C# arrays (there is lower level conversion from C# arrays to C arrays).

<u>Difficulties in the Ink to CIT conversion?</u>
Although this problem is largely resolved by training solely in our program and not using the provided CIT Training Program, we have discovered that characters which are easily recognized in the CIT Training Program appear to be recognized poorly using our Tablet PC Ink based program. These problems disappear once the symbols are retrained using our Ink based program. Nevertheless, the difference does appear somewhat odd. We have three hypotheses regarding the cause of this conundrum:

1) The CIT Training Program was written in Tcl/Tk and takes its samples from the mouse pointer. Although the pen is used as virtual mouse when writing in the training program, the sample rate is probably the same as the sample rate for the mouse. However, the Tablet PC Ink system may collect data at a higher data rate, since the screen presumably isn't bandwidth limited by a serial bus in the same manner as a mouse. Testing reveals that MS Ink may be collecting between 15% and 40% more samples than the CIT Trainer.

2) There is a possible user interface issue. Handwriting looks very different in the MS Ink API and the Tcl/Tk training program. The MS Ink handwriting is smooth, apparently fitted to Bezier Curves (regardless of whether this setting is on or off) and the width of strokes is pressure sensitive, meaning the text looks different depending on how hard the user pushes the pen. In the subjective opinion of this author, the MS Ink API feels similar to writing with a very high quality pen while the Training Program is similar to scribbling with a dull pencil or crayon. It is possible that the difference in the "look and feel" of these two programs may cause users to form their letters differently or at different speed.

3) As part of the CIT Recognizer, the program may use optional time vectors components indicating the time at which an individual sample was taken. Although this feature was apparently added to CIT very late in its development and is purely optional, it may have some importance in recognition accuracy.

In the MS Ink API, individual samples are stored in objects known as points which contain position information of the sample but no time. This limitation of the Microsoft APIs prevents us from providing CIT with sample times. It is possible the lack of timing affects recognition accuracy.