NITRO-SDK

Using the Pattern Recognition Library

Version 1.0.2

The contents in this document are highly confidential and should be handled accordingly.

NTR-06-0134-002-A2
Released: February 6, 2006

Table of Contents

1 Overview of the Pattern Recognition 4
1.1 Introduction 4
1.2 The Library Functionality 4
1.3 What the Library Can and Cannot Do 5
1.3.1 Applications that Are Possible with the API 5
1.3.2 Applications that Are Currently Possible Using Workarounds 5
1.3.3 Applications that Are Not Currently Possible 6
2 Library Usage Basics 7
2.1 Data Structures 7
2.1.1 Basic Data Types 7
2.1.2 Prototype List Type 7
2.1.3 Prototype Database Entry Type 8
2.1.4 Stroke Data Type 9
2.1.5 Recognition Algorithm-Dependent Data Types 10
2.2 Library Usage Examples 11
3 Various Setting Entries 15
3.1 Resampling Parameters 15
3.1.1 PRC_RESAMPLE_METHOD_NONE 15
3.1.2 PRC_RESAMPLE_METHOD_DISTANCE 15
3.1.3 PRC_RESAMPLE_METHOD_ANGLE 15
3.1.4 PRC_RESAMPLE_METHOD_RECURSIVE 16
3.2 Recognition Algorithms 17
3.2.1 The "Light" Algorithm 18
3.2.2 The "Standard" Algorithm 19
3.2.3 The "Fine" Algorithm 19
3.2.4 The "Superfine" Algorithm 20
4 Tricks and Tips 22
4.1 Parameter Settings 22
4.2 FAQ 22
A Appendix 24
A.1 Demos 24
A.1.1 `characterRecognition-1` 24
A.1.2 `characterRecognition-2` 24

Revision History

Version	Revision Date	Description
1.0.2	2/18/2005	Created cover and Revision History page.

1 Overview of the Pattern Recognition

1.1 Introduction

NitroSDK includes a pattern recognition library (`PRC*`) that provides rudimentary pattern recognition functionality. In this document, we will cover the basic usage of the pattern recognition library, the characteristics of the various recognition algorithms, and guidelines for tuning your application.

The pattern recognition library was designed to facilitate the use of the touch panel as an input device. If you need full-blown character recognition, including kanji, you should consider purchasing a third-party character-recognition middleware product.

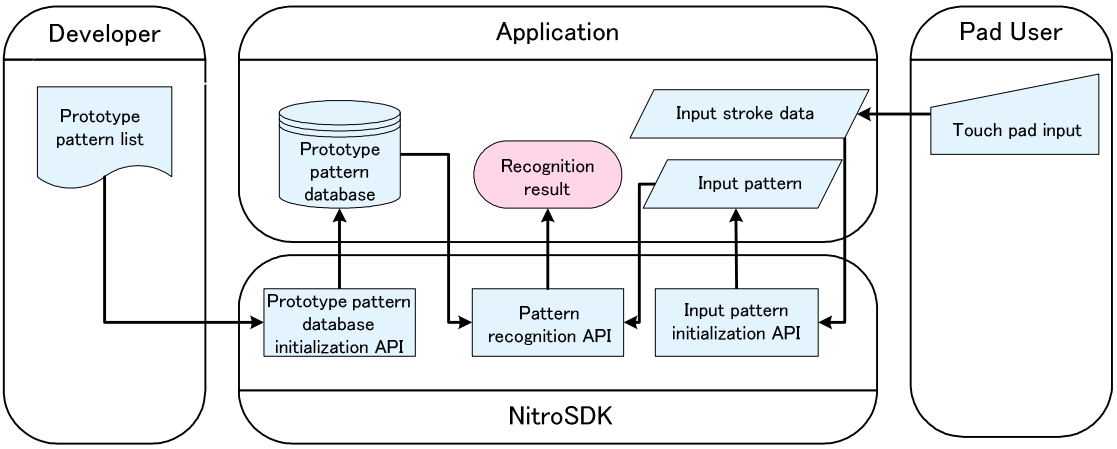
1.2 The Library Functionality

The functionality that is provided by the pattern recognition library is fairly elementary.

You must first prepare a list of pattern prototypes. Each entry in the prototype list contains a code number, a stroke count, and the coordinates of the vertices in the segments that make up each stroke.

The application program first creates the prototype database from the prototype (or prototype pattern) list. Then, it creates an array of input coordinates based on the touch panel input. When the application program begins the recognition process, it passes the input stroke data and the prototype database to the pattern recognition library.

The pattern recognition library performs matching and returns the prototype database entry that has the closest match. Finally, the application program reads the code number of the entry and uses it as the recognition result.



1.3 What the Library Can and Cannot Do

Below are some examples of what the pattern recognition library can and cannot do.

1.3.1 Applications that Are Possible with the API

- **A player writes a magic symbol on the touch panel screen during a battle, which causes a spell to take effect in the turn after he or she finishes writing the symbol.**
  This is a fairly easy to implement, as it is limited to a single stroke and it is clear that the recognition process should begin after the pen is lifted from the screen. The library returns both the recognition result and the degree of similarity. The application can be configured so that it will permit the spell to take effect only if there is a close match.

- **A name is entered one letter at a time in a designated input area of the touch panel.**
  The pattern recognition library can recognize alphabetic characters. It can also handle patterns that have multiple strokes.

- **When the player writes a map symbol on a map that is displayed on the touch panel, a building appears in the location where the symbol was written.**
  If a bounding box is defined before the touch panel coordinate data is passed to the pattern recognition library, the recognition results can be displayed in the input location.

1.3.2 Applications that Are Currently Possible Using Workarounds

- **Recognizing patterns from multiple, continuous stroke input**
  The recognition algorithms that are currently implemented require that the player write each line in the correct order. In other words, the library must know precisely the stroke that initiates recognition and the stroke that completes it. If extraneous strokes are input at any point in the process, recognition becomes impossible. If strokes at the beginning or the end are omitted, the pattern recognition library will return the entry that most closely matches the input. You can design your application to handle this result accordingly. However, if you design you application to avoid recalculation of preprocess, the restrictions will be applied to the recognition algorithm that can be used. (In particular, you must either fix the input size or use "Light," which does not require normalization of the size.)

- **Performing a calculation based on a formula written on the screen**
  If the recognition of a series of drawn patterns is attempted simultaneously, the library may have trouble determining where each pattern begins and ends. This is essentially the problem that is described in the paragraph above. If you limit your application to horizontally written formulas, the library may be able to discern individual symbols by determining where their bounding boxes overlap, but this will require a certain amount of creative coding.
  In trying to recognize a horizontal string of hiragana characters, the library may have trouble distinguishing “ﾚこ” from “1 こ”. However, this type of application could be implemented by using a combination of dynamic programming-based optimal splitting calculations and heuristics. We have not made any decisions regarding the implementation of series pattern recognition in future versions of the SDK.

- **Reading commands from specific stroke input, similar to the way that mouse gestures can be used for PC input**
  You can create an interface that interprets a stroke toward the left as a request to return to the previous screen and a stroke toward the right as a request to advance to the next screen. A hook-shaped pattern of pen movement (upward and to the left) might indicate that the screen should be closed. You can use the pattern recognition library to implement this, but if you need only recognition of up/down and left/right strokes, it may be simpler to code this yourself, or use only the resampling functionality of the library to remove noise from pen strokes. (See **`PRC_ResampleStrokes_*`**.) The choice will depend on the complexity of your application.

- **Moving an army based on the rotation angle of a symbol written on a map**
  All of the recognition algorithms that are currently implemented are sensitive to a pattern's orientation. The recognition algorithms will recognize a pattern that is written at a slight angle, but they cannot discern patterns that are written sideways or upside down. One way of permitting the rotation of a pattern is to rotate the pattern in sixteen different directions and attempt a match for each pattern orientation. The rotated pattern that best matches the pattern in the database is selected. Note that this process will increase the recognition calculation time by sixteen times since sixteen match attempts are performed rather than a single match attempt which would be the case in a simple one-to-one pattern match attempt. This process is best implemented on the application side.

1.3.3 Applications that Are Not Currently Possible

- **Asking the player to draw a Pokémon character and recognizing which one it is**
  All of the currently implemented recognition algorithms use stroke information to find a match. (This method is called "online character recognition.") They can only recognize patterns that are written in the correct stroke order. To improve the recognition of normal characters, you can store characters that are written with commonly made stroke order mistakes in the database of prototype patterns. However, the library cannot match line drawings that have no constraints on stroke order.
  It is possible to solve this problem by using a recognition algorithm that is based on bitmap images. (This is known as "offline character recognition.") But the degree of matching accuracy will suffer. We have made no decision regarding the implementation of this algorithm in the SDK.

- **Recognition of cursive writing**
  The recognition algorithms that are currently implemented rely on clear breaks between strokes. The recognition algorithms cannot recognize characters that are written without breaks, or have strokes that are not solid. If there are not many entries in the prototype database, in the case of cursive characters, you can store all likely combinations of characters as separate patterns in the prototype database, but this approach may cause the number of entries to grow beyond a manageable level. Thus, an algorithm-based recognition approach is more practical.
  We have made no decision regarding the implementation of a recognition algorithm that can handle joined characters and broken strokes in the SDK.

2 Library Usage Basics

2.1 Data Structures

First, we will examine the data structures that are used with the pattern recognition library.

2.1.1 Basic Data Types

```
#include <nitro/prc/types.h>

typedef struct PRCPoint
{
   s16              x;
   s16              y;
} PRCPoint;

typedef struct PRCBoundingBox
{
   s16              x1, y1; // Upper-left coordinate of bounding box
   s16              x2, y2; // Lower-right coordinate of bounding box
} PRCBoundingBox;
```

`PRCPoint` is a structure that expresses screen coordinates and `PRCBoundingBox` is a structure that defines the bounding box. Note that the origin (0,0) is at the upper left and the Y-axis runs downward.

2.1.2 Prototype List Type

```
typedef struct PRCPrototypeList
{
   const PRCPrototypeEntry *entries;
   int                     entrySize;
   const PRCPoint          *pointArray;
   int                     pointArraySize;

   int                     normalizeSize;
} PRCPrototypeList;
```

This data type is used for the list of prototype patterns.

Prototype list comprises an array of `PRCPrototypeEntry` (which is explained in the next paragraph), its size, an array of `PRCPoint` (used to store the vertex data in `PRCPrototypeEntry`), and its size.

The member `normalizeSize` defines the acceptable range of vertex coordinates. In the prototype list, all vertex coordinates must be within the bounding box that is defined by (0, 0) and (`normalizeSize` - 1, `normalizeSize` -1). Before it is used for actual recognition, this data is converted into a form that can be used by the prototype database.

2.1.3 Prototype Database Entry Type

```
typedef struct PRCPrototypeEntry
{
    BOOL             enabled;
    u32              kind;
    u16              code;
    fx16             correction;
    void*            data;
    int              pointIndex;
    u16              pointCount;
    u16              strokeCount;
} PRCPrototypeEntry;
```

This is the data type that is used for entries in the prototype database. Of its members, `code` and `data` can be freely used by the application as values that are linked to the entry. The member `code` is of type `u16` and can have a value of up to 65,535.

The members `enabled` and `kind` are referenced when the recognition function searches the prototype database for matches. Entries that have `enabled` set to `FALSE` are not considered for matching. The member `kind` uses a bit field to specify the type of pattern.

```
Example 1.
kind = 1 → Numeral
kind = 2 → Alphabetic character
kind = 4 → Half-size symbol
kind = 8 → Hiragana

Example 2.
kind = 1 → Level 1 spell
kind = 2 → Level 2 spell
kind = 4 → Level 3 spell
```

For example, if `kindMask` is set to `3` when the recognition function is called, matching will be limited to English letters/numerals or Level 1 and 2 spells.

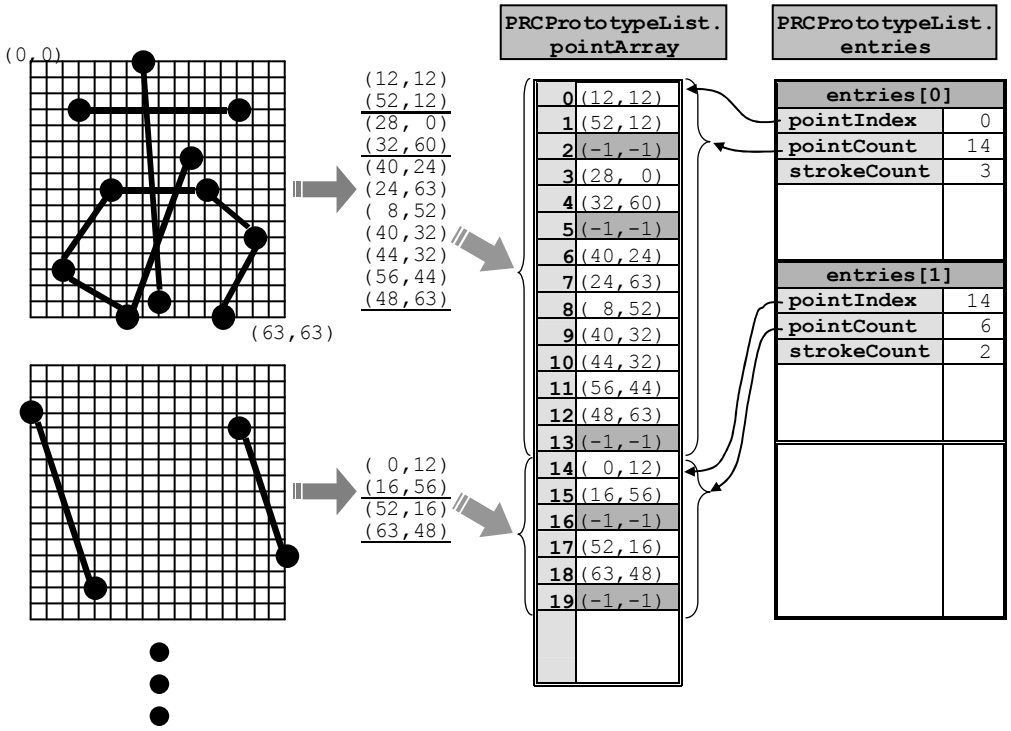
The `correction` value is used in the calculation of similarity between the input pattern and the entry. It is of type `fx16` and a value of 4,096 corresponds to 1.0. If set to 0, there is no correction. A negative value results in a low level of correction and a positive value results in a high level. If the correction value is set to 4,096, the post-correction similarity will always be 1.0 (the maximum). The following formula is used. (`score` is of type `fx32`.)

```
score = FX32_Mul(originalScore, FX32_ONE - correction) + correction
```

After processing occurs, a score that is below 0.0 becomes 0.0 and a score that is above 1.0 becomes 1.0. This figure is the final measure of similarity.

The members `pointIndex`, `pointCount`, and `strokeCount` specify the actual pattern that is defined by this entry. `pointIndex` is a subscript that specifies the location of this pattern in the pattern list's `PRCPrototypeList.pointArray`.

An example of a prototype list data structure is shown on the next page.



In this example, `PRCPrototypeList. normalizeSize is` 64.

The information that is contained in members `pointCount` and `strokeCount` of `PRCPrototypeEntry` is redundant, but you should include both members in your prototype database to speed up preprocessing.

2.1.4 Stroke Data Type

```
typedef struct PRCStrokes
{
    PRCPoint          *points;
    int               size;
    u32               capacity;
} PRCStrokes;
```

This structure is used mainly to manage the raw input coordinate data from the touch panel. `capacity` is the maximum number of points that can be stored and `size` is the current number.

The following operations are defined in the library.

```
PRCStrokes strokes;
PRCPoint points[1024];

// Initializes the strokes structure
PRC InitStrokes(&strokes, points, 1024);
// Adds a set of input coordinates (x, y) from the touch panel
PRC_AppendPoint(&strokes, x, y);
// Records the fact that the pen has been lifted from the screen
PRC_AppendPenUpMarker(&strokes);
// Checks if the structure has reached its capacity
PRC IsFull(&strokes);
// Clears the structure
PRC_Clear(&strokes);
// Checks if the structure is empty
PRC_IsEmpty(&strokes);

PRCStrokes anotherStrokes;
PRCPoint anotherPoints[2048];
PRC_InitStrokes(&anotherStrokes, anotherPoints, 2048);

// Makes a deep copy
PRC_CopyStrokes(&strokes, &anotherStrokes);

int i;
for ( i=0; i<strokes.size; i++ )
{
   if ( !PRC_IsPenUpMarker(&strokes.points[i]) )
   {
      // Ordinary processing
   }
   else
   {
      // The pen was lifted at this point
   }
}
```

2.1.5 Recognition Algorithm-Dependent Data Types

- `PRCPrototypeDB`
  `PRCPrototypeList` stores only the bare minimum of prototype data. To speed up the recognition process, you must preprocess the vertex data in the prototype list. Use `PRC_InitPrototypeDB` to preprocess the `PRCPrototypeList` (the prototype list) and produce `PRCPrototypeDB`, which is the actual prototype database that holds the data that is passed to the recognition functions.
  Its internal structure depends on the recognition algorithm that is used, but all of the recognition algorithms that are currently implemented use a common data structure. The items that are added to the initial data are: indices to the starting point of each stroke, the length of each line segment, the length of each stroke, the total length of the pattern, the ratio of line segment to stroke length for each line segment, the ratio of stroke to pattern length for each stroke, the angle of each line segment, the bounding box for each stroke, and the bounding box for the entire pattern.

- `PRCInputPattern`
  The input coordinate data from the touch panel that is stored in `PRCStrokes` must also be preprocessed before it is passed to the recognition functions. Touch panel input is often sampled once per frame, which results in too many points for the recognition algorithm to use. You must resample the

input pattern to extract the points that best define its features. To create the PRCInputPattern structure, the PRC_InitInputPattern function resamples the raw input stroke data and performs additional calculations that are similar to those calculations that are performed by PRC_InitPrototypeDB.

2.2 Library Usage Examples

The following pseudocode excerpts are examples of library usage.

```
#include <nitro/prc.h>

#define RAW_POINT_MAX 1024 // How many raw input points to save
#define POINT_MAX 40 // Maximum number of points to accept after resampling
#define STROKE_MAX 4 // Maximum number of input strokes to accept
```

You cannot call nitro.h from the PRC* header file. To use the pattern recognition library, you must explicitly place nitro/prc.h in an include statement. Here, instead of placing nitro/prc.h in an include statement, we can select the default pattern recognition algorithm by specifying nitro/prc/algo_*.h. For details, see the sections on the various recognition algorithms below.

To use the pattern recognition library, a number of parameters must be defined as macro constants.

The value that is specified by RAW_POINT_MAX is the maximum number of input points that can be accepted by the touch panel. Because the pattern recognition library processes an array of an entire series of points as a single target, the application must be able to store all of the input points. If the touch panel accepts 60 points each second, and a single character requires at most 10 seconds to input, the application will need to store an array of 600 points.

During preprocessing, the raw input data that is handed off by the application is stripped down to its characteristic points. This is called resampling or characteristic point extraction. POINT_MAX and STROKE_MAX define the maximum number of points and strokes that are permitted after preprocessing occurs. If POINT_MAX is set to a value that is too low, a long and complex set of input data for a single character can be truncated in the middle. The proper setting for this constant will depend on the complexity of the input pattern that you require and on the number of points you want to preserve after preprocessing (PRC_InitInputPattern*).

```
extern PRCPrototypeList PrototypeList;

// Allocates a work region for extracting the prototype database
PRCPrototypeDB protoDB;
void* dictWork;
dictWork =
    OS_Alloc(PRC_GetPrototypeDBBufferSize(&PrototypeList));
PRC_InitPrototypeDB(&protoDB, dictWork, &PrototypeList);
```

Think of PrototypeList as the prototype list data that is defined in a separate file.

Use PRC_InitPrototypeDB to create PRCPrototypeDB (the prototype database) from PRCPrototypeList (the prototype list). You must allocate sufficient memory for PRCPrototypeDB based on the size of the prototype database. Allocate a region of memory based on the size that is obtained by PRC_GetPrototypeDBBufferSize and pass it during initialization.

PRC_InitPrototypeDB will count the total number of points and strokes in the prototype set, and then perform calculations that will speed up recognition processing.

These calculations include the creation of an index of all strokes that determines the length and angle of each segment and other data that are required by the recognition algorithms. This information is stored in `PRCPrototypeDB`.

`PRC_InitPrototypeDB` has a sibling function called `PRC_InitPrototypeDBEx`, which allows you to specify which prototypes to use based on a bit field. When using `PRC_InitPrototypeDBEx`, be sure to calculate the size of the work area by providing `PRC_GetPrototypeDBBufferSizeEx` with the same arguments that are used in `PRC_InitPrototypeDBEx`.

```
// Allocate a work area for other processing
void* inputWork;
inputWork =
   OS_Alloc(PRC_GetInputPatternBufferSize(POINT_MAX, STROKE_MAX));
void* recogWork;
recogWork = OS Alloc(
   PRC_GetRecognitionBufferSize(POINT_MAX, STROKE_MAX, &protoDB)
);
```

The code above allocates the work area that is needed for the recognition process. To pool multiple input patterns in parallel, you need to allocate one work area for extracting input pattern and another work area for comparison processing. You do not need to allocate new memory for each recognition process if you specify the largest values that you will need at the outset.

```
// Initialize the input stroke data
PRCPoint points[RAW POINT MAX];
PRCStrokes strokes;
PRC_InitStrokes(&strokes, points, RAW_POINT_MAX);
```

The code above initializes the structure that holds the raw data input from the touch panel.

```
while ( 1 )
{
```

This loop is entered each frame.

```
   int x, y;
   if ( !PRC_IsFull(&strokes) )
   {
      if ( there is (x,y) input from the touch panel )
      {
         // Append point (x,y) to the stroke
         PRC AppendPoint(&strokes, x, y);
      }
      else if (  there was input in the previous frame )
      {
         // Insert a "pen up" marker
         PRC_AppendPenUpMarker(&strokes);
      }
   }
```

The code above adds the input from the touch panel to the `PRCStrokes` structure. If the pen is lifted from the touch panel, you must call `PRC_AppendPenUpMarker` once (but no more than once) to append a "pen up" marker.

```
   if ( there is a request for recognition )
   {
      // Start recognition using the current contents of strokes
      // First, set the resampling process parameters
      PRCInputPatternParam inputParam;
      inputParam.normalizeSize = protoDB.normalizeSize;
      inputParam.resampleMethod = PRC_RESAMPLE_METHOD_RECURSIVE;
      inputParam.resampleThreshold = 3;
```

Here, we set the parameters that are required for the conversion of the raw stroke data to the `PRCInputPattern` type data that is used for the recognition process. If `normalizeSize` is set to a non-zero value, the bounding box of the input stroke will be normalized (expanded or contracted) to match the specified size. All of the recognition algorithms, except the Light algorithm, assume that the prototype database and the input pattern are the same size. Be sure to use normalization so that the input size will match the prototype database size.

`resampleMethod` and `resampleThreshold` are used to set both the algorithm and the parameters that are used to extract the characteristic points from the raw input data. For details, see the section below on resampling algorithms.

```
      // Use resampling on the raw input points to reduce the number of
datapoints;
      // Perform preprocessing to determine additional
      // information such as length and create inputPattern
      PRCInputPattern inputPattern;
      PRC InitInputPatternEx(&inputPattern, inputWork, &strokes,
          POINT_MAX, STROKE_MAX, &inputParam);
```

Using the work area that was allocated previously, process the raw input points and create a `PRCInputPattern` type input pattern data.

Based on the parameters from `PRCInputPatternParam`, `PRC_InitInputPattern` performs normalization and resampling to extract the characteristic points. It then calculates segment lengths and angles from these points and stores this information in the `PRCInputPattern` structure.

```
      // Perform recognition by comparing inputPattern with entries
      // in protoDB
      PRCPrototypeEntry* result;
      fx32 score;
      score = PRC_GetRecognizedEntry(&result, recogWork,
                                     &inputPattern, &protoDB);
```

This method completes the preparation for the recognition process. Next, we need to compare the input pattern data `(PRCInputPattern)` with the prototype database `(PRCPrototypeDB)` and find the database entry that most closely matches the input pattern data. The level of similarity is a type `fx32` with a range from 0 to 1. (If converted to an `int`, it would range from 0 to 4,096.)

The processing could take several tens of milliseconds or more, depending on the algorithm you select and the size of the prototype database. Therefore, we recommend using a separate thread for this processing. For an implementation example, see the demo (prc/characterRecognition-1).

The sibling function `PRC_GetRecognizedEntryEx` allows you to use a bit field to specify the types of patterns to recognize. `PRC_GetRecognizedEntries` returns the N entries that are the best matches. See the reference manual for details.

```
        // Output the result
        OS_Printf("code: %d\n", PRC_GetEntryCode(result));
```

As a recognition result, the function returns a pointer to a `PRCPrototypeEntry` in `PRCPrototypeList`. You can use `PRC_GetEntryCode` and `PRC_GetEntryData` to obtain the code and user data of the entry.

```
}
    Processes that wait for V-Sync
}
```

3 Various Setting Entries

3.1 Resampling Parameters

You can choose which of several algorithms to use for the resampling process that is conducted by `PRC_InitInputPattern`.

3.1.1 PRC_RESAMPLE_METHOD_NONE

No resampling is performed. This method removes only points that duplicate the immediately preceding coordinates. This method can be used when it is necessary to reprocess stroke data that has already been resampled.

3.1.2 PRC_RESAMPLE_METHOD_DISTANCE

This method resamples based on the distance traveled.

This method captures the starting and ending points of each stroke, and captures a point each time a stroke travels more than a predefined cumulative distance from the starting point. The distance that is measured is not the Euclidean distance, but the change in the X position plus the change in the Y position, or the “city block” or “Manhattan” distance. This distance calculation is less precise than with Euclidean distance, but is faster to process.

The `resampleThreshold` specifies the cumulative distance that the stroke has to travel before capturing the next point.

This is the fastest method to process, but strokes that are drawn slowly with a shaky pen may cause the threshold to be reached quickly, resulting in too many points being captured. Also, this method tends not to capture the best characteristic points.

3.1.3 PRC_RESAMPLE_METHOD_ANGLE

This method performs sampling based on the curvature of each stroke.

First, the starting and ending points of each stroke are captured. Then, the angle of the segment that is connected to the starting point is stored. The connecting segments are followed in succession until the angle difference reaches the threshold angle. The point immediately before the point where the threshold is exceeded is captured as the second point in the series. The angle of the segment that connects the two immediately preceding points is compared to the angle of the segment that connects the preceding point with the current point. If the difference is greater than the threshold angle, then the current point is captured. This process is repeated.

`FX_Atan2Idx`, which uses an internal table lookup, performs the angle calculations and this speeds up the process. `FX_Atan2Idx` is not highly accurate, but it is sufficient for this purpose.

The `resampleThreshold` specifies the threshold angle. The range of values is from 0 to 65,535, with 1.0 representing 1/65,535 of a full circle.

Each point that is captured must have a city block distance from the previous point that is greater than that specified by `PRC_RESAMPLE_ANGLE_LENGTH_THRESHOLD`. This is because valid angles cannot be measured at very short distances. Currently, `PRC_RESAMPLE_ANGLE_LENGTH_THRESHOLD` is fixed at 6. The distance calculations use the raw, non–normalized coordinates.

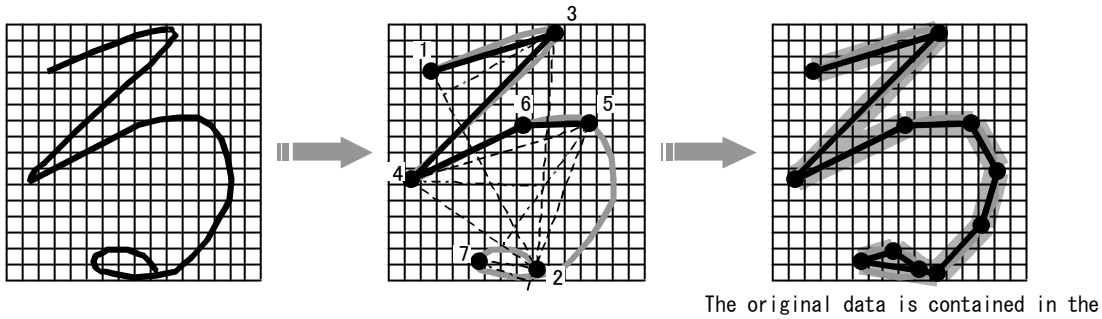
Even if you set the threshold high to reduce the number of resampled points, this method can still accurately capture points in small loops and extract good characteristic points. Conversely, if the threshold is set too low, slight stroke bends will be picked up. The calculation time is linear to the number of input points.

3.1.4 PRC_RESAMPLE_METHOD_RECURSIVE

The method processes the input recursively and captures the most characteristic points.

First, the starting and ending points are captured and defined as points A and B. All points between points A and B are tested and the point that is the farthest on a straight line that is drawn between points A and B is defined as point C. If the distance is more than the `resampleThreshold`, C is captured. Otherwise, the line from point A to point B is retained. If C is captured, the process is reiterated for points A and C and points C and B.

Ultimately, this process will completely capture all the original raw input stroke data in the region that is bounded by the `resampleThreshold` distance on both sides of the resulting line segments. However, this method will not capture all the points if the number of resampling points reaches the upper limit during the process.



By setting the `resampleThreshold` to a value that is smaller than the smallest loop you expect in the input pattern, you should be able to generate a relatively compact set of resampling data without missing any loops. If your resampling results are compact, the recognition process will be faster.

The calculation time for the resampling process itself is, in the worst case, proportional to the product of the number of input points and the number of resulting resampling points. With typical input data, such as hiragana characters, this method will take slightly longer than `PRC_RESAMPLE_METHOD_ANGLE`. This is based on the assumption that the parameters have been set to have both methods generate the same number of resampled points.

3.2 Recognition Algorithms

Currently, four pattern recognition algorithms have been implemented.

The first header file placed in an include statement is selected for the recognition algorithm.

```
#include <nitro/prc/algo_light.h>     → recognition algorithm  "Light"
#include <nitro/prc/algo_standard.h>  → recognition algorithm  "Standard"
#include <nitro/prc/algo_fine.h>      → recognition algorithm  "Fine"
#include <nitro/prc/algo_superfine.h> → recognition algorithm  "Superfine"
```

If you describe `#include <nitro/prc.h>`, all four of the above header files will be loaded. Because algo_standard.h is loaded first, "Standard" is the default recognition algorithm.

The following library functions and types vary depending on the recognition algorithm.

```
PRCPrototypeDB
PRCInputPattern
PRCPrototypeDBParam
PRCInputPatternParam
PRCRecognizeParam

PRC_Init
PRC_GetPrototypeDBBufferSize*
PRC_InitPrototypeDB*
PRC_GetInputPatternBufferSize
PRC_InitInputPattern*
PRC_GetInputPatternStrokes
PRC_GetRecognitionBufferSize*
PRC_GetRecognizedEntry*
```

Each recognition algorithm uses the identifiers above with the name of the algorithm appended to it as a suffix. The above identifiers will be treated as aliases of those in the first header file that is loaded. To use the recognition algorithms that were placed in an include statement after the first header, you must explicitly use types and function names with the suffix `_<algorithm name>`. (For example, `PRCRecognizeParam_Light`, `PRC_InitPrototypeDBEx_Fine`.)

However, of these functions and types in the current implementation, the following library functions are common to all the recognition algorithms.

```
PRCPrototypeDB
PRCInputPattern
PRCPrototypeDBParam
PRCInputPatternParam

PRC_Init
PRC_GetPrototypeDBBufferSize*
PRC_InitPrototypeDB*
PRC_GetInputPatternBufferSize
PRC_InitInputPattern*
PRC_GetInputPatternStrokes
```

These functions and types use the suffix `_Common`, which is referenced by all the algorithms. (For example: `PRCPrototypeDB_Common`.)

Apart from algorithms that are related to `PRC_GetRecognizedEntry*`, which performs the actual recognition, currently; all other algorithms use the same libraries. The `prc/characterRecognition-2` demo exploits this feature to access a shared prototype database and shared input pattern data by using all of the recognition algorithms simultaneously. Refer to the demo as an example of how to use multiple recognition algorithms simultaneously.

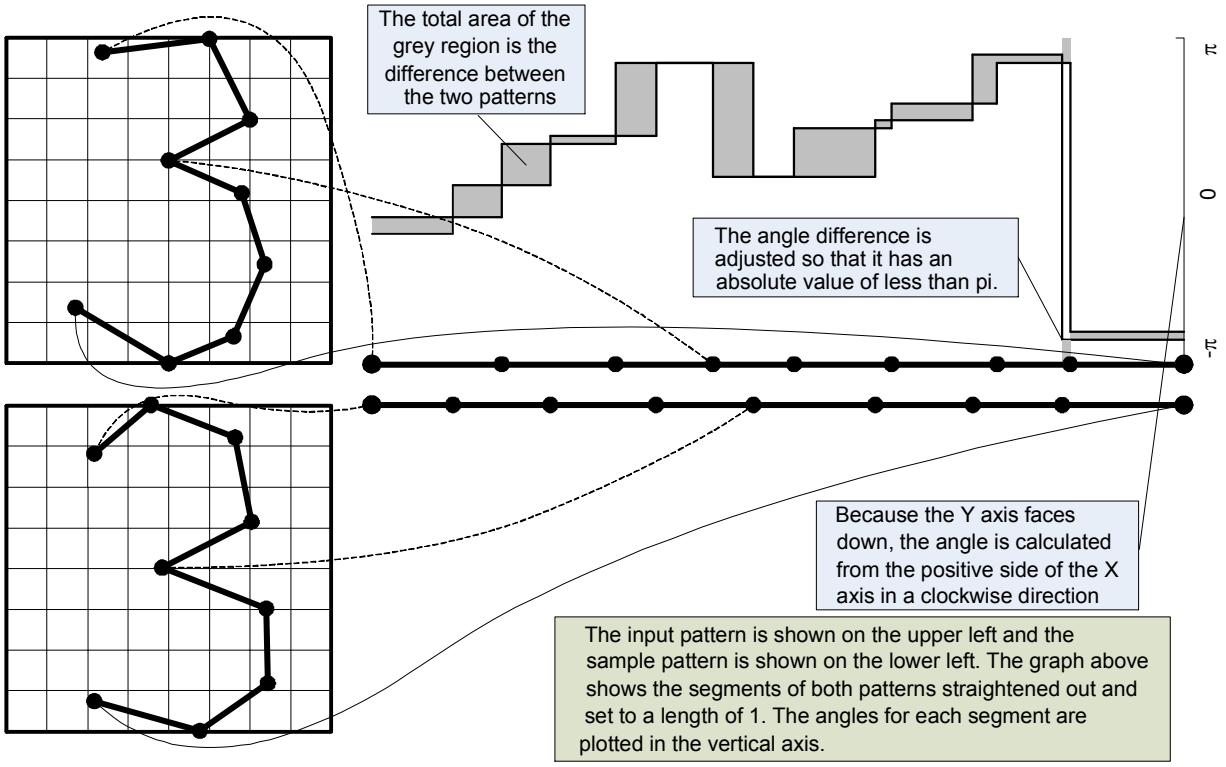
Below is an overview of each algorithm.

In the following discussion, we often use vague terms because the accuracy and calculation time for each algorithm depends greatly on a number of factors. Statements about processing speed are for reference purposes only. Select your recognition algorithm and set your parameters only after thorough testing with the data that is used with your application.

3.2.1 The "Light" Algorithm

The "Light" algorithm is the most lightweight recognition algorithm. It is ideal for situations where the patterns in the prototype database are distinct (making recognition errors unlikely) or when you want to recognize patterns that consist of only a single stroke.

The Light algorithm compares only angles. The strokes of the input pattern and prototype are expanded or contracted so that the total length of each is one. Then, the integral of the difference in angles is taken, and the degree of similarity is computed and returned. The values are adjusted so that a similarity of 0.0 is returned if all angles differ by 180° and a similarity of 1.0 is returned if all angles that match perfectly.



The preceding graph shows the angle difference in a graphic form.

When comparing patterns with multiple strokes, the Light algorithm performs the same calculations on each stroke, and then computes the weighted average of all the similarity scores, weighting each stroke in the prototype based on its length relative to the entire pattern. However, the Light algorithm does not examine the relative positions of each stroke and thus it has the inherent drawback of not being able to distinguish "T" from "+". This algorithm was designed mainly to recognize single stroke patterns at the fastest speed possible.

The calculation time will be proportional to the product of the number of points in the input pattern and the number of entries in the prototype database.

3.2.2 The "Standard" Algorithm

The Standard Algorithm was designed as a standard recognition algorithm. It is ideal for situations that require the player to enter a pattern, such as a magic symbol, correctly.

The Standard algorithm compares both angles and positions. Like the Light algorithm, it adjusts the length of the input pattern and the prototype so that they are both 1, and takes the integral value of the angle differences multiplied by the position differences. Distances are measured with the “city block” method, and approximated to the closest sampling point coordinates, rather than taking the difference between the exact points. Like the Light algorithm, the Standard algorithm adjusts the similarity values so that 0.0 indicates a lack of similarity and 1.0 indicates a perfect match, and returns it as a score.

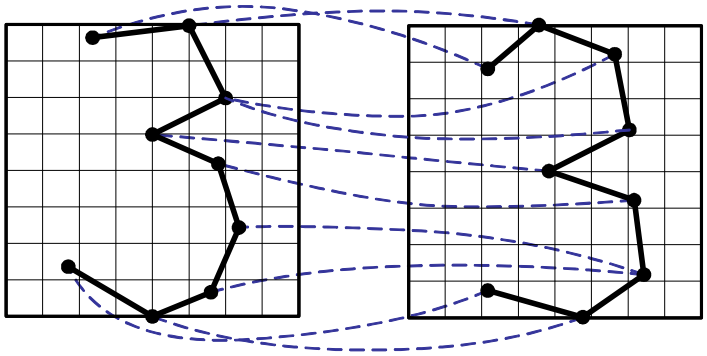
Because the Standard algorithm looks at positioning, it can easily recognize patterns with multiple strokes. In determining the similarity score, after performing the above calculations on each stroke, the Standard algorithm computes the weighted average of similarity scores, giving each stroke in both the database entry and the input pattern a weight that is proportional to its length relative to the entire pattern.

The calculation time for the Standard algorithm is two or three times longer than the calculation time that is required for the Light algorithm. Even if you set up a recognition thread that runs when the main thread is idle, the result should come back in an acceptable period of time.

3.2.3 The "Fine" Algorithm

The "Fine" algorithm was designed as an algorithm that can handle even distorted characters. It would be useful in situations where the application needs to salvage the character input data such as the distorted input from the user.

In addition to comparing both angles and positions, the Fine algorithm performs elastic matching. It does not match the input and prototypes by changing their size, but rather it expands and contracts the individual strokes and looks for matches that result in a high evaluation score.



The preceding is an example of elastic matching. Vertices in the input pattern (left side) are compared with vertices in the prototype database entry (right side). You can see that more than once vertices are sometimes mapped to a single vertex. By searching for the combinations that produce the highest score while permitting more than one points to be mapped to a single point, the Fine algorithm can easily handle distortions, such as those that are shown in the figure, and generate a high score for a “3” drawn so that the upper and lower sections are not the same size as the prototype. Elastic matching is good at correctly interpreting distorted input.

To compute the score, the following formula is used on each matching vertex.

```
(normalized size × 2 − the city block distance)×(π − the difference
between the angles of the segments entering the vertex)
```

Then, an average of the vertices is taken and the result is distributed over the range of 0.0-1.0. The vertices are matched in various ways to find the vertex that generates the highest score.

Elastic matching is performed using an algorithm that is based on Dynamic Programming (DP Matching). It does not implement a beam search. Accordingly, the calculation time is proportional to the number of points in the input pattern times the number of points in the prototype. In a typical application, the Fine algorithm usually takes several times longer than the Standard algorithm.

3.2.4 The "Superfine" Algorithm

The "Superfine" algorithm is the recognition algorithm that requires the longest processing time among the algorithms that are currently implemented. However, it is not always more accurate than the Fine algorithm. Use the Superfine algorithm when you find that the Fine algorithm is not accurate enough.

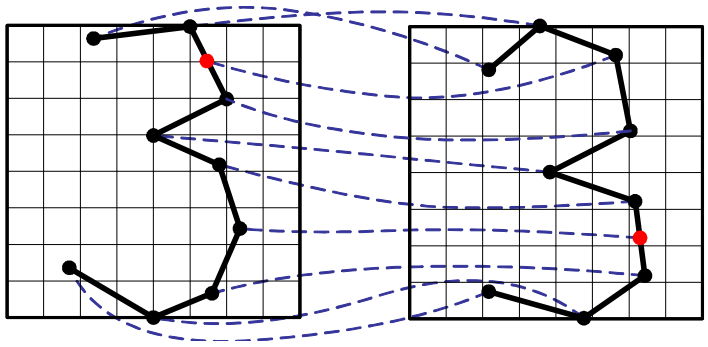
Like the Fine algorithm, the Superfine algorithm uses elastic matching. The Fine algorithm takes the evaluation values that are used by elastic matching and returns the values as the score, but Superfine uses elastic matching to obtain information on which points should map to which. Elastic matching determines the most likely vertex matches, and those vertices without a certain match are matched with hypothetical points on the other pattern based on the lengths of the segments before and after the vertex in question. The Superfine algorithm then computes a final score in the same manner as the Fine algorithm.

Unlike the Fine algorithm, the Superfine algorithm uses the following formula for each point to compute the final score.

```
(normalized size × 2 − the city block distance) × (cos of the difference between the angles of the segments entering the vertex)
```

The Superfine algorithm then computes a weighted average of all the points based on the lengths of the segments that are connected to each point relative to the entire pattern.

When finding vertex pairs using DP matching, the Superfine algorithm does not treat segment lengths in the same way as the Fine algorithm. The angle score is computed using a cosine function.



In the graph above, the red points are hypothetical points that result from interpolation. In addition to performing the operations that are used by the Fine algorithm, the Superfine algorithm must perform frequent division to generate the interpolated points. The calculation time that is required by the Superfine algorithm to generate the interpolated points is often several times longer than the calculation time that is required by the Fine algorithm.

4 Tricks and Tips

4.1 Parameter Settings

The easiest way to learn about parameter adjustments is to change the parameters in the demo (prc/characterRecognition-2) and watch the effect on memory usage, calculation time, and accuracy. Because performance will vary greatly depending on the nature of the prototype database, you need to make your parameter adjustments using data that is as close as possible to the prototype database that you will use in your actual application. The instructions for the `characterRecognition-2` demo are in the Appendix at the end of this document.

If certain patterns are recognized too frequently, you can adjust their correction values in the database to prevent this. However, you can easily end up making a large number of unnecessary minor adjustments. You can use the same code value for several database entries. If you have patterns that are not being recognized, it might be easier to add new prototypes to the database until you begin to have matches for those patterns.

4.2 FAQ

Q. You can specify `kindMask` with both `PRC_InitPrototypeDB*` and `PRC_GetRecognizedEntry*`. Which should I use to select a certain type of pattern?
A. This depends on how often you want to change your selection criteria. Specifying with `PRC_InitPrototypeDB` will reduce the memory that is required for extracting the prototype database, but you will not be able to easily change the set of patterns you want to target.

Q. I want a lightweight algorithm that will recognize patterns with multiple strokes. Can the Light algorithm be used for this purpose? I don’t need a high level of recognition accuracy, but the inability to distinguish "p" from "b" is going to be a problem.
A. There is a way to use the Light algorithm for the recognition of patterns that have multiple strokes. This is accomplished by not using PenUpMarker. Normally, when the pen is lifted, a `PRC_AppendPenUpMarker` is used to show the stroke was completed, but if you omit this operation, the pattern recognition library will treat a series of strokes as a single connected stroke. By populating your prototype database with patterns that have a single unbroken stroke, the Light algorithm will be able to perform recognition that reflects the positional relationships of multiple strokes.

This technique is also useful for handling joined characters and lines that fade out part way. Nonetheless, the possibility of unintentional matches will naturally increase. Select your patterns accordingly to avoid this.

Q. I want to use the resampling results for processing outside of the game. Is that possible?
A. For `PRCInputPattern`, use `PRC_GetInputPatternStrokes`. This creates a pointer that points directly to the data that is contained in `PRCInputPattern`, so there is no need to initialize the first parameter with `PRC_InitStrokes`. If you want to change the contents, you can copy the structure with `PRC_CopyStrokes` before using the contents.

If you only want to perform resampling, you can use `PRC_ResampleStrokes*`. The results of this function will be returned as an index array. Use the application to convert the results to the `PRCStrokes` type.

A Appendix

A.1 Demos

The pattern recognition library demos are stored in the `$NITROSDK_ROOT/build/demos/prc/` directory of the NitroSDK.

A.1.1 characterRecognition-1

Several problems can occur when using the pattern recognition library. The calculation time can sometimes exceed a single frame and can vary greatly depending on the complexity of the input pattern. Therefore, set up a pattern recognition thread that is separate from the main thread. Ideally, the application should perform pattern recognition during the idle period after main thread processing is finished and before the V-Blank interrupt is generated. The `characterRecognition-1` demo is an example of an application that uses a separate thread.

Perform recognition with the A Button and clear the screen with the B Button.

The prototype database has 161 entries that can be used for testing. The prototype database contains Arabic numerals, lowercase alphabets, hiragana characters, and some symbols. Because there are multiple patterns for each of the numerals, the total number of characters that can be recognized is 117. This prototype database is used only for demonstration purposes. For your own application, you should build a new prototype database using sampling points and standard patterns that meet your requirements for speed and accuracy.

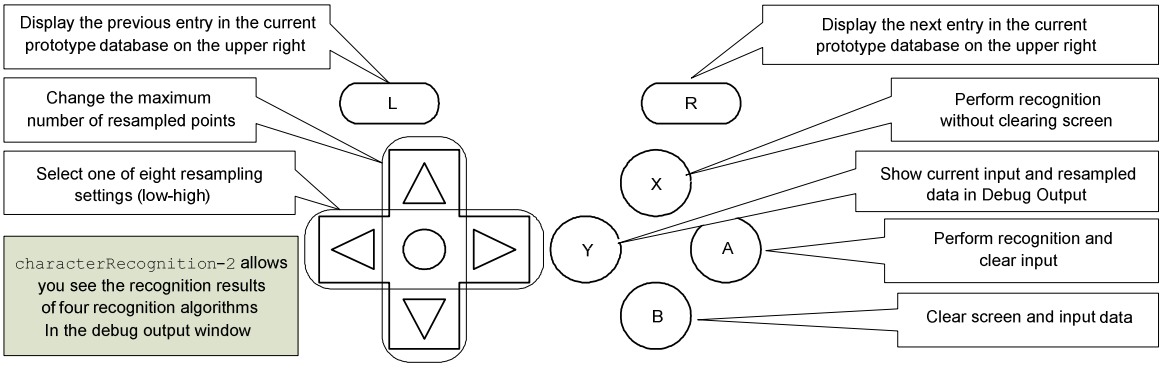
A.1.2 characterRecognition-2

This demo application is designed to compare the various pattern recognition algorithms. It allows you to use a prototype database on the production unit to see the effect of changing `maxPointCount` (the largest number of sampling points accepted) on the size of the work area and understand how the adjustment of resampling parameters can affect recognition time and results.

There are eight threshold combinations (from low to high) that have been tuned to generate a similar number of sampling points using the three sampling algorithms. These settings can be changed during runtime.

This demo uses the prototype database that is used in the `characterRecognition-1` demo. By repopulating the database with your actual application data, you can use this demo application to help determine the optimal parameter settings.

Start the application and draw a pattern on the touch panel. When you press the A Button, four patterns will appear on the screen. The three patterns on the left are the sampling results. The three respective patterns are: `PRC_RESAMPLE_METHOD_DISTANCE`, `ANGLE`, and `RECURSIVE`. The rightmost pattern is the recognition result prototype data. The Debug Output window displays detailed recognition results for each algorithm.



This demo can also be used as a basic pattern creation tool.

Set the sampling parameters using the +Control Pad (Left/Right) and draw a pattern with the pen. Press the Y Button and the resampling result pattern data for each of the three resampling algorithms will appear in text form in the Debug Output. You can cut and paste the text data one line at a time for various patterns into a text file and run the following demo sample to obtain a C source code listing for that prototype list that can be read by the pattern recognition library.

```
$ perl $NITROSDK_ROOT/tools/bin/pdic2c.pl <normalized size for output>
<prototype database  text data>
```

Use the source code to check the operation of the pattern recognition library.

For details on the input format used by `pdic2c.pl`, see the reference manual.