Cyberon CSpotter SDK (32-bit IC Edition)

Programming Guide

Version: 2.0.0

Date of issue: September 22, 2016

Leading Speech Solution provider

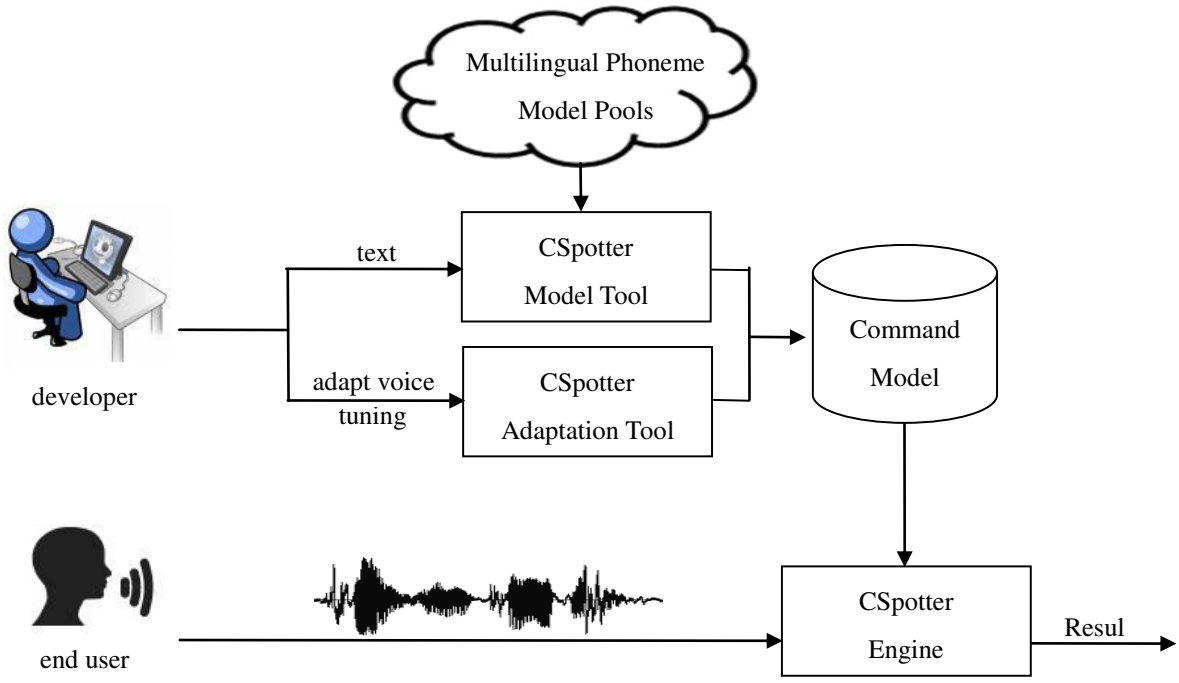
http://www.cyberon.com.tw/

Table of Contents

1. About Cyberon CSpotter SDK

CSpotter SDK is Cyberon's flagship high-performance embedded voice recognition solution specially optimized for mobile phones, automotives, smart home devices, consumer products, and interactive toys. Based on phoneme acoustic models, it enables developers to create applications of speaker-independent (SI) voice recognition capability without requiring costly data collection process for specific commands. Small footprint and compact algorithm design, without compromise in recognition accuracy, allow CSpotter running on resource-limited platforms, such as MCU and low-power consumption DSP. With cloud-based CSpotter Model Tool, developers can easily and quickly create their own voice command models simply by text input. CSpotter Adaptation Tool, the PC-based utility incorporated with speaker adaptation technology, is also available for developers to further improve accuracy with small amount of voice training data from target users. Additionally, for advanced version, speaker-dependent (SD) voice tag training function allows end users to create their own commands directly with voice input. Other important features include always-on keyword-spotting capability, highly noise immune, adjustable sensitivity, voice quality assessment, and more than 30 commonly used language versions available.

There are two versions of CSpotter SDK, standard and advanced versions. SD voice tag training and voice quality assessment functions are provided only in the advanced version. Note that sub-licensable version of CSpotter SDK from some IC vendors may support only standard version functions. Contact your IC vendors to get more information about the version of CSpotter SDK you are using.



2. Release History

Date	Version	Author	Description
2015/03/26	1.0.0	Calvin	**Purpose:** First release
2015/05/15	1.0.1	Calvin	**Purpose:** Remove CSpotter_Init() Add CSpotter_Init_Sep()
2015/06/30	1.0.2	Calvin	**Purpose:** Add CSpotter_GetMemoryUsage_Sep() Add CSpotter_GetResultEPD() Add CSpotter_GetNumWord ()
2015/07/03	1.0.3	Calvin	**Purpose:** Add advanced version APIs
2015/07/07	1.0.4	Calvin	**Purpose:** Calling flow chart of Advanced SDK API
2015/08/24	1.1.0	Calvin	**Purpose:** Reorganize chapters Add users implement flash operation functions Add CSpotter supported languages
2015/12/16	1.1.1	Calvin	**Purpose:** Update the specification Add CSpotter_GetMemoryUsage_Sep()
2016/04/07	1.1.2	Calvin	**Purpose:** Add CSpotter_SetRejectionLevel() Add CSpotter_SetResponseTime() Add CSpotter_SetCmdReward () Add CSpotter_SetCmdResponseReward() Add CSpotter_SetMinEnergyThreshd()

2016/09/22	2.0.0	Calvin	**Purpose:** Comment the size of utterance and Training buffer Update CSpotterSD_TrainWord() Delete CspotterSD_AddWord() Add CSpotterSD_GetTagID() Calling flow chart of Professional SDK API Add CSpotterSD_AddUttrStartForSpeaker() Add CSpotterSD_TrainWordForSpeaker() Add Cspotter_GetMemoryUsage_SepWithSpeaker() Add Cspotter_GetMemoryUsage_MultiWithSpeaker() Add Cspotter_GetSpeakerResult() Add Cspotter_SetSpeakerModel()

3. CSpotter Specifications, Related Files and Tools

3.1. Specifications

CSpotter algorithm is available for 16-bit and 32-bit IC platforms. The core engines are scalable with a wide range of configurations available, and can be ported to a variety of platforms with different resources and architectures. Here lists CSpotter specifications ported to some popular platforms. For 16-bit CSP16 and 32-bit CSP32, the standard versions of CSpotter algorithms without voice tag training function, given n_c, the number voice commands, each of which is 4 syllables in average, the technical specification is listed in the following table:

Algorithm	CSP16	CSP16	CSP32	CSP32
IC Architecture	16-bit, single-cycle 16x16 into 32 MAC	16-bit, single-cycle 16x16 into 32 MAC	32-bit, fix-point ALU	32-bit, fix-point ALU
Sample Rate	8kHz	8kHz	16kHz	16kHz
Feature Dimension	16	24	24	36
Code size	18KB	18KB	22KB	22KB
Data size	7.3KB + 2.6KB*n_c	11KB + 3.8KB*n_c	11KB + 3.8KB*n_c	17KB + 5.7KB*n_c
RAM size	4KB + 104B*n_c	4KB + 104B*n_c	6.6KB + 104B*n_c	8KB + 104B*n_c
Ported Platforms	16-bit DSP Tensilica 24-bit DSP ARM M0	16-bit DSP Tensilica 24-bit DSP ARM M0	ARM7 ARM M0, M3, M4	ARM7 ARM M0, M3, M4

For 16-bit CSP16A and 32-bit CSP32A, the advanced version of CSpotter algorithm equipped with voice tag training function, given n_c, the number voice commands, and n_v, the number of voice tags, the technical specification is listed below:

Algorithm	CSP16A	CSP16A	CSP32A
IC Architecture Requirement	16-bit, single-cycle 16x16 into 32 MAC	16-bit, single-cycle 16x16 into 32 MAC	32-bit with fix-point ALU
Input Sample Rate	8kHz	8kHz	16kHz
Feature Dimension	16	24	24
Code size	29KB	29KB	35KB
Data size	7.3KB + 2.6KB*nc+ 2KB* n_v	11KB + 3.8KB*nc+ 2KB* n_v	11KB + 3.8KB*n_c + 2KB*n_v
RAM size	Max{4KB + 104B*(n_c+ n_v), 4.8KB}	Max{4KB + 104B*(n_c+ n_v), 4.8KB}	Max{6.5KB + 116B*(n_c+ n_v), 9KB}
Ported Platforms	16-bit DSP Tensilica 24-bit DSP ARM M0	16-bit DSP Tensilica 24-bit DSP ARM M0	ARM7 ARM M0, M3, M4

For 16-bit CSP16P and 32-bit CSP32P, the professional version of CSpotter algorithm equipped with voice tag training and speaker verification function, given n_c, the number voice commands, and n_v, the number of voice tags, and n_s, the number of speaker tags, the technical specification is listed below:

Algorithm	CSP16P	CSP16P	CSP32P
IC Architecture Requirement	16-bit, single-cycle 16x16 into 32 MAC	16-bit, single-cycle 16x16 into 32 MAC	32-bit with fix-point ALU
Input Sample Rate	8kHz	8kHz	16kHz
Feature Dimension	16	24	24
Code size	33KB	33KB	42KB
Data size	7.3KB + 2.6KB*n_c + 2KB*n_v + 5KB*n_s	11KB + 3.8KB*n_c + 2KB*n_v + 5KB*n_s	11KB + 3.8KB*n_c + 2KB*n_v + 5KB*n_s
RAM size	Max{4.4KB + 104B*(n_c + n_v) + 300B* n_s, 4.8KB}	Max{4.4KB + 104B*(n_c + n_v) + 300B* n_s, 4.8KB}	Max{7KB + 104B*(n_c + n_v) + 300B* n_s, 9.5KB}
Ported Platforms	16-bit DSP Tensilica 24-bit DSP ARM M0	16-bit DSP Tensilica 24-bit DSP ARM M0	ARM7 ARM M0, M3, M4

Note that code size listed in this document is for CSpotter core engine only, and codes for recording, playback, voice compression, data communication, and application main function are not included.

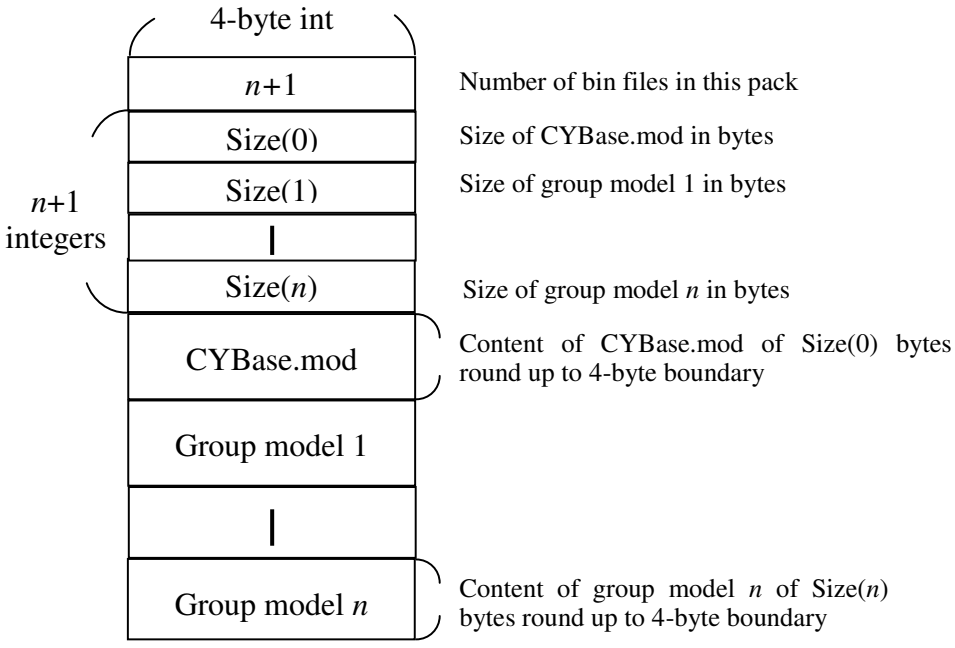
3.2. Related Files and Tools

Library

- **CSpotterSDK_16k24d_XXXX.lib**, the library for CSpotter standard version, where XXXX is the name of the IC platform running the CSpotter engine, 16k and 24d stand for 16kHz sampling rate input and 24-dimensional feature vectors respectively.
- **CSpotterSDK_A_16k24d_XXXX.lib**, the library for CSpotter advanced version.
- **CSpotterSDK_P_16k24d_XXXX.lib**, the library for CSpotter professional version.

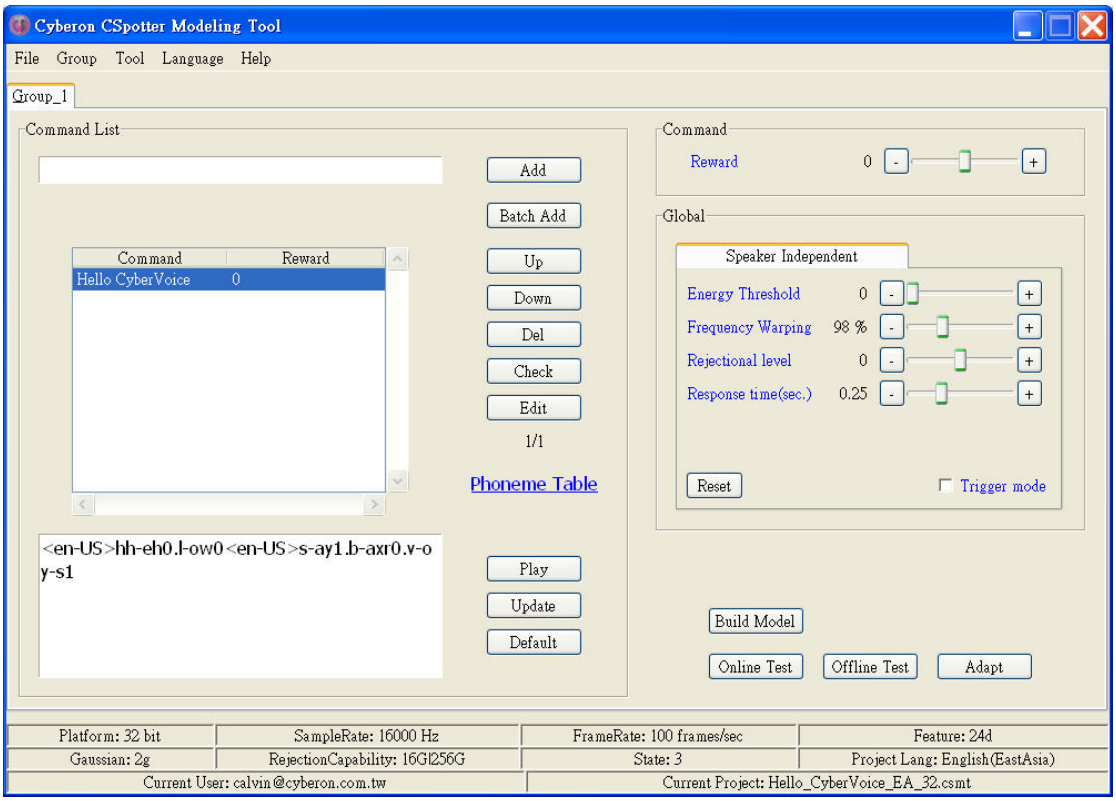
Data

- **CYBase.mod**: the background model. The file name CYBase.mod is reserved for CSpotter Model Tool and CSpotter Adaptation Tool, and should never be changed.
- **XXXX.mod**: the command group model (or called command model). "Group_*n*" is the default name for the *n*-th group of commands in a project when created with CSpotter Model Tool, and can be renamed. All the command group models share the same CYBase.mod in a project.
- **XXXX_pack.bin**: the binary file that packs all command group models together with the shared CYBase.mod in a project, where XXXX is the project name assigned by developer when creating it with CSpotter Model Tool. Before using the models in the packed binary file, developers need to unpack it. For a CSpotter project of *n* group models, the packing format is shown in the diagram below. Note that this packing file is 4-byte aligned in Little-Endian manner.



Tools

- **CSpotter Model Tool**, a Microsoft Win32-based tool for developers to create command models for CSpotter recognition engine. Prior registration is required before developers can use CSpotter Model Tool. Contact with Cyberon if you are new to CSpotter.



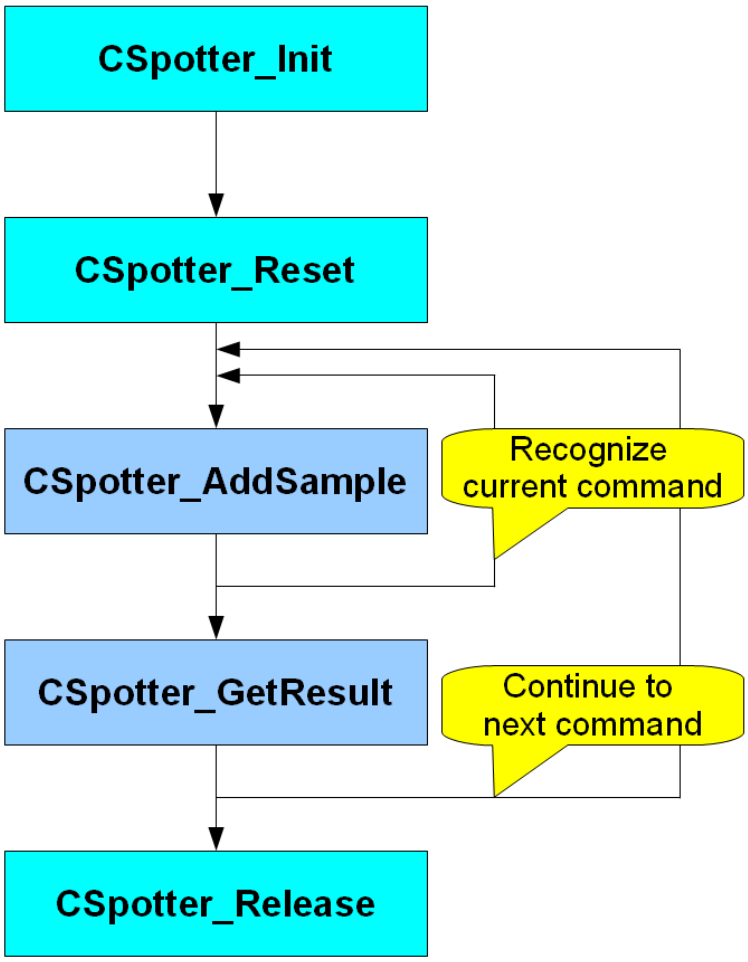
- **CSpotter_GetMemoryUsage**, a Microsoft Windows command line tool to calculate the required memory size for CSpotter standard version to perform voice recognition, given the input command models. Its usage is as follows:

  CSpotter_GetMemoryUsage <CYBase.mod> < model 1> [<model 2> ...]

  CSpotter can perform recognition for several input models simultaneously. For this tool, CYBase.mod and at least a command model are required. Note that all the command models must share the same background model CYBase.mod, and therefore they have to come from the same project created with CSpotter Model tool.

4. CSpotter SDK API Standard Version

4.1. Calling Flow Chart of Standard API



4.2. Initialize, Reset, and Release

CSpotter_Init_Sep

Purpose

Create a recognizer for recognizing a single group of commands.

Prototype

HANDLE CSpotter_Init_Sep(BYTE *lpbyCYBase, BYTE *lpbyModel, INT nMaxTime, BYTE *lpbyMemPool, INT nMemSize, BYTE *lpbyState, INT nStateSize, INT *pnErr);

Parameters

lpbyCYBase(IN): The background model, contents of CYBase.mod.

lpbyModel(IN): The command model.

nMaxTime(IN): The maximum buffer length in number of frames for keeping the status information of commands.

lpbyMemPool(IN/OUT): Memory buffer for the recognizer.

nMemSize(IN): Size in bytes of the memory buffer *lpbyMemPool*.

lpbyState(IN/OUT): State buffer for recognizer.

nStateSize(IN): Size in bytes of the state buffer *lpbyState*.

pnErr(OUT): The return code.

Return value

Return the handle of a recognizer when success or NULL otherwise.

Remarks

It is highly recommended that the value of *nMaxTime* should be greater than the maximum duration of all commands, and recognizer could keep the status information of commands during recognition. Note that higher value of *nMaxTime* will increase the memory usage.

A statically reserved buffer of memory pointed by *lpbyMemPool* is required to call this function. Developers can get the memory buffer size *nMemSize* in advance by using the command line tool *CSpotter_GetMemoryUsage*. This memory buffer can be recycled and used by other functions when the recognition task finishes after calling *CSpotter_Release(…)*.

Pointer *lbyState* is a reserved, and should never be released, buffer of memory to backup the current state of the recognizer. By passing this pointer to the initialization function, the recognizer can be rolled back to the state before it was released last time. **The buffer should be reset to zero for all its elements when it is employed for the first time and never be used by other application**. The buffer size *nStateSize* is about 96 bytes.

Pointer *pnErr* receives the return code after calling this function, *CSPOTTER_SUCCESS* indicating success, otherwise a negative error code is returned. This pointer can be NULL.

CSpotter_Init_Multi

Purpose

Create a recognizer for recognizing multiple groups of commands simultaneously.

Prototype

HANDLE CSpotter_Init_Multi(BYTE *lpbyCYBase, BYTE *lppbyModel[], INT nNumModel, INT nMaxTime, BYTE *lpbyMemPool, INT nMemSize, BYTE *lpbyState, INT nStateSize, INT *pnErr);

Parameters

lpbyCYBase(IN): The background model, contents of CYBase.mod

lppbyModel(IN): An array of command models to be recognized simultaneously.

nNumModel(IN): Number of models in array *lppbyModel*.

nMaxTime(IN): The maximum buffer length in number of frames for keeping the status of commands.

lpbyMemPool(IN/OUT): Memory buffer for the recognizer.

nMemSize(IN): Size in bytes of the memory buffer *lpbyMemPool*.

lpbyState(IN/OUT): State buffer for recognizer.

nStateSize(IN): Size in bytes of the state buffer *lpbyState*.

pnErr(OUT): The return code.

Return value

Return the handle of a recognizer when success or NULL otherwise.

Remarks

See remarks for function *CSpotter_Init_Sep(...)*.

The maximum number of command models is 10, and if the *speaker-dependent (SD) command model* is one of them, it must be the last command of the model.

CSpotter_Reset

Purpose

Reset the recognizer before performing recognition.

Prototype

INT CSpotter_Reset(HANDLE hCSpotter);

Parameters

hCSpotter (IN): Handle of the recognizer.

Return value

CSPOTTER_SUCCESS for success or negative error code otherwise.

CSpotter_Release

Purpose

Release a recognizer.

Prototype

INT CSpotter_Release(HANDLE hCSpotter);

Parameters

hCSpotter (IN): Handle of the recognizer.

Return value

CSPOTTER_SUCCESS for success or negative error code otherwise.

CSpotter_GetMemoryUsage_Sep

Purpose

Get current memory usage.

Prototype

INT CSpotter_GetMemoryUsage_Sep(BYTE *lpbyCYBase, BYTE *lpbyModel , INT nMaxTime);

Parameters

lpbyCYBase(IN): The background model, contents of CYBase.mod

lpbyModel(IN): The command model.

nMaxTime(IN): The maximum buffer length in number of frames for keeping the status of commands.

Return value

The memory size in bytes or error code.

CSpotter_GetMemoryUsage_Multi

Purpose

Get current memory usage.

Prototype

INT CSpotter_GetMemoryUsage_Multi(BYTE *lpbyCYBase, BYTE *lppbyModel[], INT nNumModel, INT nMaxTime);

Parameters

lpbyCYBase(IN): The background model, contents of CYBase.mod

lppbyModel(IN): An array of command models to be recognized simultaneously.

nNumModel(IN): Number of models in array *lppbyModel*.

nMaxTime(IN): The maximum buffer length in number of frames for keeping the status of commands.

Return value

The memory size in bytes or error code.

CSpotter_GetMemoryUsage_SepWithSpeaker

Purpose

Get current memory usage.

Prototype

INT CSpotter_GetMemoryUsage_SepWithSpeaker(BYTE *lpbyCYBase, BYTE *lpbyModel, INT nMaxTime, BYTE *lpbySpeakerModel);

Parameters

lpbyCYBase(IN): The background model, contents of CYBase.mod

lpbyModel(IN): The command model.

nMaxTime(IN): The maximum buffer length in number of frames for keeping the status of commands.

lpbySpeakerModel(IN): The pointer of training model buffer in **DATA FALSH**.

Return value

The memory size in bytes or error code.

Remarks

This function is supported only in the **professional version** of CSpotter SDK.

CSpotter_GetMemoryUsage_MultiWithSpeaker

Purpose

Get current memory usage.

Prototype

INT CSpotter_GetMemoryUsage_MultiWithSpeaker(BYTE *lpbyCYBase, BYTE *lppbyModel[], INT nNumModel, INT nMaxTime, BYTE *lpbySpeakerModel);

Parameters

lpbyCYBase(IN): The background model, contents of CYBase.mod

lppbyModel(IN): An array of command models to be recognized simultaneously.

nNumModel(IN): Number of models in array *lppbyModel*.

nMaxTime(IN): The maximum buffer length in number of frames for keeping the status of commands.

lpbySpeakerModel(IN): The pointer of training model buffer in **DATA FALSH**.

Return value

The memory size in bytes or error code.

Remarks

This function is supported only in the **professional version** of CSpotter SDK.

4.3. Recognition

Functions in this section are designed to perform recognition process for the recognizer. For some platforms with relatively limited RAM, the pseudo codes below demonstrate how to use **union** data type of C language to store memory buffers for CSpotter engine, playback, and functions of developer's application in the same location. CSpotter engine retains the memory buffer pointed by *lpbyMemPool* until *CSpotter_Release(…)* is called, after which the buffer is released and can be recycled and reused by other functions. The pseudo codes below show the calling sequence for always-listening voice recognition:

```
// Declare shared memory using data type union in C.
union ShareMem {
	BYTE lpbyMemPool[N];	// N can be obtained with tool CSpotter_GetMemoryUsage
	SHORT lpsPlayBuffer[…];
	<Other buffers used by application>
} ShareMem;

// Declare and zero the memory for backing up the recognizer state.
BYTE lpbyState[M] = { 0 };		// M is about 96 bytes

DoVR(…)
{
	// Create a recognizer
	hCSpotter = CSpotter_Init_Sep(…, ShareMem.lpbyMemPool, N, lpbyState, M, …);
	if (hCSpotter == NULL)
		goto L_ERROR;

	<Start Recording>

	CSpotter_Reset(…);

	while (1)
	{
		<Get PCM samples from recording device>
		if (CSpotter_AddSample(…) == CSPOTTER_SUCCESS)
		{
			nID = CSpotter_GetResult(…);
			CSpotter_GetResultEPD(…);			// Optional
			nScore = CSpotter_GetResultScore(…);	// Optional
			break;
		}
	}

L_ERROR:

	<Stop Recording>

	CSpotter_Release(…);
	// Share memory ShareMem can be used by other functions after CSpotter_Release(…).

	<Play Prompt using memory ShareMem.lpsPlayBuffer >
}
```

CSpotter_AddSample

Purpose

Add voice samples to the recognizer and perform recognition.

Prototype

INT CSpotter_AddSample(HANDLE hCSpotter, SHORT *lpsSample, INT nNumSamples);

Parameters

hCSpotter (IN): Handle of the recognizer.

lpsSample(IN): An array of 16kHz, 16-bit, mono-channel PCM raw data.

nNumSamples(IN): Number of samples in *lpsSample*.

Return value

Result	Comment
CSPOTTER_SUCCESS	A recognition result is concluded, and application can call *CSpotterGetResult(…)* to retrieve the result.
CSPOTTER_ERR_NeedMoreSample	Recognition result has not been found yet, and need to call this function again to add more samples to the recognizer.
CSPOTTER_ERR_Rejected	A rejected result is concluded, and application can call *CSpotterGetResult(…)* to retrieve the result.
Other negative error code	

Remarks

Application should call this function repetitively to add recorded PCM raw data into the recognizer for recognition to proceed until a recognition is found, at which moment this function returns *CSPOTTER_SUCCESS*, and the application can then call *CSpotter_GetResult(...)* to retrieve the recognized result. The recommended length of the input array of samples *lpsSample* is 160 samples (= 320 bytes).

CSpotter_GetResult

Purpose

Get the recognition result from the recognizer.

Prototype

INT CSpotter_GetResult(HANDLE hCSpotter);

Parameters

hCSpotter (IN): Handle of the recognizer.

Return value

Return the zero-based command ID when success or negative error code otherwise. If there are more than one command models being recognized simultaneously, the command ID is enumerated in order. For example, if there are 2 models containing n_1 and n_2 commands respectively, the ID for the third command in the second model is n_1+2.

CSpotter_GetResultEPD

Purpose

Get the boundary information of the current recognition result.

Prototype

INT CSpotter_GetResultEPD(HANDLE hCSpotter, INT *pnWordDura, INT *pnEndDelay);

Parameters

hCSpotter (IN): Handle of the recognizer.

pnWordDura(OUT): Duration of the result in number of samples.

pnEndDelay(OUT): Ending silence length in number of samples.

Return value

Return the command ID when success, or negative error code otherwise.

Remarks

EPD stands for end-point detection. CSpotter determines the completion of an input voice command by counting the length of the ending silence. This function retrieves the command duration and length of the ending silence. Developers can also calculate the command start time if necessary. In the application, number of added samples is recorded with variable *nTotAddSample*. Then the start time *nStartTime* is

nStartTime = *nTotAddSample* - **pnWordDura* - **pnEndDelay*;

CSpotter_GetResultScore

Purpose

Get the reliability score of the current recognition result.

Prototype

INT CSpotter_GetResultScore (HANDLE hCSpotter);

Parameters

hCSpotter (IN): Handle of the recognizer.

Return value

Return the non-negative reliability score of the recognition result when success, or negative error code otherwise. The physical meaning of this score is unclear, but it is somewhat positively related to the likelihood of the input speech matching against the recognized command.

Remarks

This function is supported only in the **advanced version** of CSpotter SDK.

CSpotter_GetNumWord

Purpose

Get the number of commands in the input model.

Prototype

INT CSpotter_GetNumWord(BYTE *lpbyModel);

Parameters

lpbyModel(IN): The command model.

Return value

Return the number of commands when success, or negative error code otherwise.

CSpotter_SetRejectionLevel

Prototype

INT CSpotter_SetRejectionLevel(HANDLE hCSpotter, INT nRejectionLevel);

Parameters

hCSpotter (IN): Handle of the recognizer.

nRejectionLevel (IN): Rejection level. The range is [-100, 100], higher rejection level will make the engine more "picky" to return a result.

Return value

Return 0 if successful, or negative error code otherwise.

Remarks

The rejection level is a global threshold that applies to the entire model. A higher level makes the engine more "picky" to return a result. It is recommended to perform sufficient amount of field tests from different users if the rejection level is changed from its default value.

CSpotter_SetResponseTime

Prototype

INT CSpotter_SetResponseTime(HANDLE hCSpotter, INT nResponseTime);

Parameters

hCSpotter (IN): Handle of the recognizer.

nResponseTime (IN): Response time. The range is [0, 100], lower value will make the engine quicker to return a result. The default is 25.

Return value

Return 0 if successful, or negative error code otherwise.

Remarks

The response time is a global attribute that applies to the entire model. It defines the duration of silence after the voice input for the engine to determine the end of a voice command. Though a longer response time makes the engine slower or more "picky" to respond to user's voice input, it can usually give more stable recognition results with less false triggers.

CSpotter_SetCmdReward

Prototype

INT CSpotter_SetCmdReward(HANDLE hCSpotter, INT nCmdIdx, INT nReward);

Parameters

hCSpotter (IN): Handle of the recognizer.

nCmdIdx (IN): The command ID. It is 0 based.

nReward(IN): Command reward. The range is [-100, 100], higher value will increase the chance of being accepted result. The default is 0.

Return value

Return 0 if successful, or negative error code otherwise.

Remarks

Sometimes a specific command's model is not good enough working with other commands. It is possible to adjust the command's performance by adding or deducting its rewards. A larger reward means the command can be more easily recognized. On the other hand, by doing so, it could also increase the risk of false triggers, which means the recognizer could falsely respond to out-of-vocabulary utterances.

CSpotter_SetCmdResponseReward

Prototype

INT CSpotter_SetCmdResponseReward(HANDLE hCSpotter, INT nCmdIdx, INT nReward);

Parameters

hCSpotter (IN): Handle of the recognizer.

nCmdIdx (IN): The command ID. It is 0 based.

nReward(IN): Response reward. The range is [-100, 100], lower value will make the engine quicker to return this result. The default is 0.

Return value

Return 0 if successful, or negative error code otherwise.

Remarks

The response time of a specific command can be estimated by the following formula:

Command response time = Global response time + Individual response reward

A reasonable response time can usually give more stable recognition results with less false triggers.

CSpotter_SetMinEnergyThreshd

Prototype

INT CSpotter_SetMinEnergyThreshd(HANDLE hCSpotter, INT nThreshold);

Parameters

hCSpotter (IN): Handle of the recognizer.

nThreshold (IN): Energy threshold. The range is [0, 30000], speech energy (RMS) must be over the threshold.

Return value

Return 0 if successful, or negative error code otherwise.

Remarks

Energy threshold that sets the minimal energy for a recognized command is a global setting applied to the entire model, hopefully to reject some falsely recognized commands caused by the background noise of relatively low volume.

The energy is calculated with root mean square (RMS) of all the samples of the recognized command. For low-volume background noise with relatively smaller variation in signal, its RMS energy is close to the arithmetic-mean energy and usually tends to be smaller, while for normal speech of larger signal variation, the RMS energy is higher.

CSpotter_SetEnableRejectedResult

Prototype

INT CSpotter_SetEnableRejectedResult(HANDLE hCSpotter, INT bEnable);

Parameters

hCSpotter (IN): Handle of the recognizer.

bEnable (IN): A boolean flag to enable or disable reporting rejected result.

Return value

Return 0 if successful, or negative error code otherwise.

Remarks

CSpotter_AddSample(…) can report rejection of a recognized command. By default this capability is disabled, and it always returns *CSPOTTER_ERR_NeedMoreSample* if there is no acceptable result recognized, and simply drops all recognition results not well matched with the acoustic model. After enabling this capability, *CSpotter_AddSample(…)* will return *CSPOTTER_ERR_Rejected* if a rejected result is concluded, at which moment application can call *CSpotter_GetResult(…)*, *CSpotter_GetResultEPD(…)*, and *CSpotter_GetResultScore(…)* to get the information of the rejected command.

CSpotter_SetRejectedResultEnergyThreshd

Prototype

INT CSpotter_SetRejectedResultEnergyThreshd(HANDLE hCSpotter, INT nThreshold);

Parameters

nThreshd (IN): The threshold for the average amplitude of the voiced part of the recorded data. This parameter ranges from 0 to 10000, and the default value is 500.

Return value

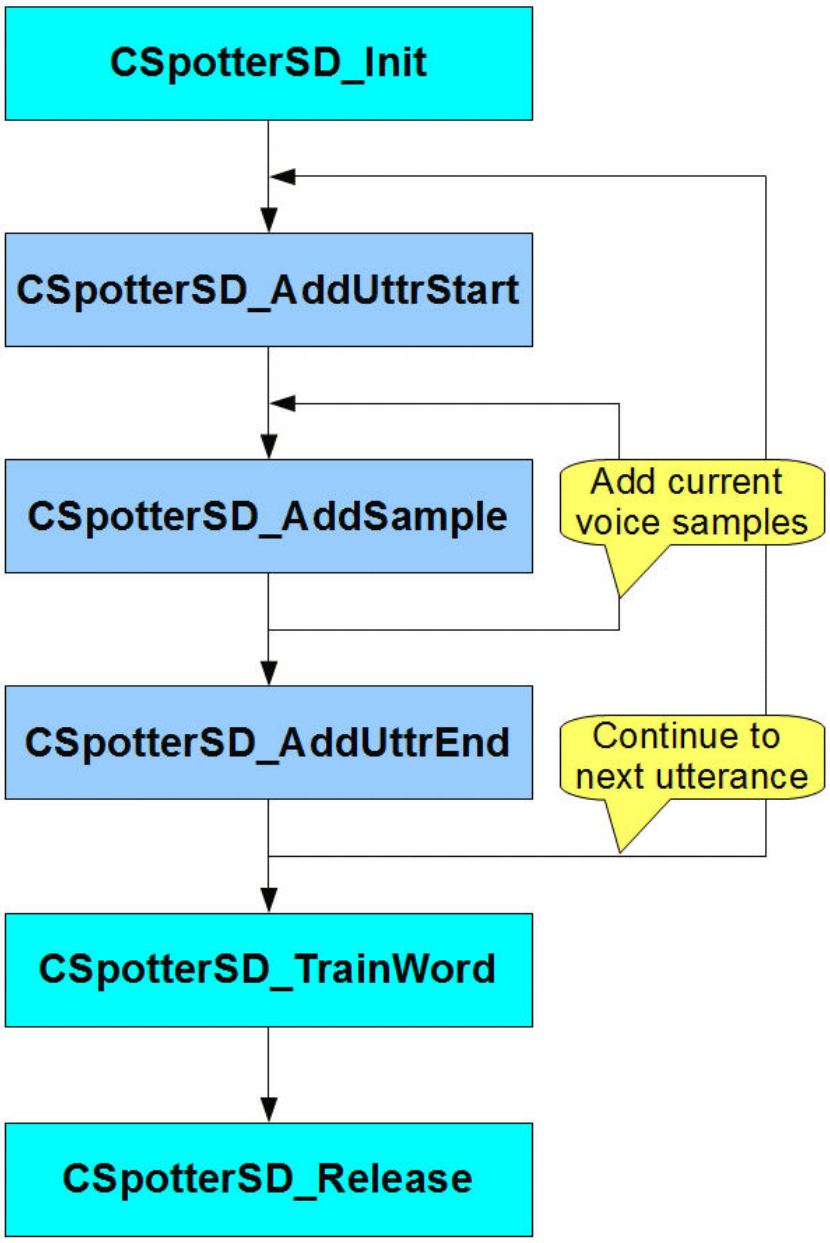
Return 0 if successful, or negative error code otherwise.

Remarks

Sometimes many rejected results are caused by the environmental noise. It would be quite annoying if the recognizer reports rejections too frequently. Developers can use this function to control the sensitivity and frequency of reporting rejections, and hence the interactivity of the product can be improved.

5. CSpotter SDK API Advanced Version

5.1. Calling Flow Chart of Advanced API



5.2. Initialize and Release

CSpotterSD_Init

Purpose

Create a CSpotter voice tag trainer.

Prototype

HANDLE CSpotterSD_Init(BYTE *lpbyCYBase, BYTE *lpbyMemPool, INT nMemSize, INT *pnErr);

Parameters

lpbyCYBase(IN): The background model for trainer, contents of CYBase.mod

lpbyMemPool(IN/OUT): Memory buffer for the trainer

nMemSize(IN): Size in bytes for memory buffer *lpbyMemPool*.

pnErr(OUT): The return code.

Return value

Return the handle of a trainer when success or NULL otherwise.

Remarks

A background model Cybase.mod is required to train a voice tag. The trainer extracts parameters from the input Cybase.mod and adapt them using the training utterances provided by the user. Models sharing the same CYBase.mod, including the speaker-independent (SI) ones crerated from CSpotter Model Tool and the speaker-dependent (SD) voice tags trained here, can be put together and recognized by CSpotter engine simulatenuously.

A statically reserved buffer of memory pointed by *lpbyMemPool* is required to call this function. This memory buffer can be recycled and used by other functions when the training task finishes after calling *CSpotterSD_Release(…)*.

Pointer *pnErr* receives the return code after calling this function, *CSPOTTER_SUCCESS* indicating success, otherwise a negative error code is returned. This pointer can be NULL.

CSpotterSD_Release

Purpose

Release the trainer.

Prototype

INT CSpotterSD_Release(HANDLE hCSpotter);

Parameters

hCSpotter(IN): Handle of the trainer.

Return value

CSPOTTER_SUCCESS for success or negative error code otherwise.

5.3. Training

Functions in this section are designed to perform training process for SD voice tag. The pseudo codes below show the calling sequence for training an SD voice tag:

```
// Declare shared memory using data type union in C.
union ShareMem {
	BYTE lpbyMemPool[N];	// N: obtained with CSpotterSD_GetMemoryUsage(…)
	<Other buffers used by application>
} ShareMem;

DoVR_train(…)
{
	<Prepare storage (possibly in flash) for utterance buffer>

	// Create a trainer
	hCSpotter = CSpotterSD_Init(…, ShareMem.lpbyMemPool, N, …);
	if (hCSpotter == NULL)
		goto L_ERROR;

	// Preparation stage: adding utterances to train a voice tag
	for (i = 0; i < k_nNumUttr; i++)
	{
		if (CSpotterSD_AddUttrStart(…) != CSPOTTER_SUCCESS)
			goto L_ERROR;

		<Start Recording>

		while (1)
		{
			<Get PCM samples from recording device>
			if (CSpotterSD_AddSample(…) != CSPOTTER_ERR_NeedMoreSample)
				break;
			// Use CSpotterSD_GetUttrEPD(…) to get the starting point of the input
			// utterance, and then start to compress it and write to data flash. (Optional)
			if (CSpotterSD_GetUttrEPD(…) == CSPOTTER_SUCCESS)
				<Compress recorded voice data and write it to data flash>
		}

		<Stop Recording>

		if (CSpotterSD_AddUttrEnd(…) != CSPOTTER_SUCCESS)
			goto L_ERROR;
		// Use CSpotterSD_GetUttrEPD(…) to get the ending point of the input utterance,
		// and move the compressed voice to external flash of larger size. (Optional)
		if (CSpotterSD_GetUttrEPD(…) == CSPOTTER_SUCCESS)
			<Move the compressed voice data from data flash to SPI flash>

		// Use CSpotterSD_SimilarityCheck(…) to check whether the input utterance is
		// similar with an existing voice tag or not. (Optional)
		if (i == 0 && CspotterSD_SimilarityCheck(…) >= 0)
			<Error Handling …>

		// Use CSpotterSD_DifferenceCheck(…) to check whether the input utterance is
		// different from the first one. (Optional)
		if (i > 0 && CspotterSD_DifferenceCheck(…) != CSPOTTER_SUCCESS)
			<Error Handling …>
	}

	// Training stage, and then add voice tag to the model for recognition
```

```
        if (CSpotterSD_TrainWord(…) != CSPOTTER_SUCCESS)
            <Error Handling …>

L_ERROR:

        CSpotterSD_Release(…);
}
```

CSpotterSD_AddUttrStart

Purpose

Prepare to add a new utterance for training.

Prototype

INT CSpotterSD_AddUttrStart(HANDLE hCSpotter, INT nUttrID, char *lpszDataBuf, INT nBufSize);

Parameters

hCSpotter(IN): Handle of the trainer.

nUttrID(IN): ID of the utterance ranging [0, 2].

lpszDataBuf(IN/OUT): The pointer of data buffer in **DATA FALSH** to store voice input.

nBufSize(IN): Size in bytes of the data buffer *lpszDataBuf*.

Return value

CSPOTTER_SUCCESS for success or negative error code otherwise.

Remarks

Parameter *nUttrID* ranges from 0 to 2. That is, a voice tag can be trained with at most 3 utterances. The more the training utterances, the more reliable the voice tag will be. Note that the utterance that has been added previously will be replaced if the same *nUttrID* is used.

The data buffer *lpszDataBuf* is a pointer to internal data flash, or it can be pointing to external flash through SPI bus, as long as the bus is fast enough with address mapping hardware equipped. Internally in this function, user-implemented functions *DataFlash_Write(…)* and *DataFlash_Erase(…)*, as described in the next section, are employed to access the data flash. If RAM is large enough, developers can also use RAM to simulate data flash when implementing these 2 functions. Note that CSpotter SDK assumes the page size for erasing flash is 1KB. For the consideration of efficiency, pointer *lpszDataBuf* has to be 1KB aligned and *nBufSize* a multiple of 1KB. If *lpszDataBuf* is NULL, this function returns the required size of the data buffer rounded to a multiple of 1KB. Currently the time duration of one voice tag is 3 second, which requires around 8KB data buffer. If given less than 8KB, the maximum length of voice tag shrinks by ratio. ex. 1.5 second voice tag for given 4KB data buffer

CSpotterSD_AddSample

Purpose

Add voice samples to the trainer for training.

Prototype

INT CSpotterSD_AddSample(HANDLE hCSpotter, SHORT *lpsSample, INT nNumSample);

Parameters

hCSpotter (IN): Handle of the trainer.

lpsSample(IN): An array of 16kHz, 16-bit, mono-channel PCM raw data.

nNumSamples(IN): Number of samples in *lpsSample*.

Return value

CSPOTTER_ERR_NeedMoreSample indicates that the caller should call this function again, otherwise *CSPOTTER_SUCCESS* for successfully obtaining a recognition results, or negative error code otherwise.

CSpotterSD_AddUttrEnd

Purpose

Finish the adding process for training a voice tag.

Prototype

INT CSpotterSD_AddUttrEnd(HANDLE hCSpotter);

Parameters

hCSpotter (IN): Handle of the trainer.

Return value

CSPOTTER_SUCCESS for success or negative error code otherwise.

Remarks

Training utterances are added to the trainer by calling *CSpotterSD_AddUttrStart(...)*, *CSpotterSD_AddSample(...)* repeatedly, and *CSpotterSD_AddUttrEnd(...)*, which constitutes the data preparation stage before training a voice tag.

CSpotterSD_GetUttrEPD

Purpose

Get the boundary information of the currently added training utterance.

Prototype

INT CSpotterSD_GetUttrEPD(HANDLE hCSpotter, INT *pnStart, INT *pnEnd);

Parameters

hCSpotter (IN): Handle of the trainer.

pnStart(OUT): Starting point in samples

pnEnd(OUT): Ending point in samples

Return value

CSPOTTER_SUCCESS for success or negative error code otherwise.

Remarks

Usually this function is employed when developers want to store user's voice data for playback purpose. Values of *pnStart* and *pnEnd* are valid only when the function returns *CSPOTTER_SUCCESS.*

CSpotterSD_SimilarityCheck

Purpose

Check whether the added utterance is similar with an existing voice tag or not.

Prototype

INT CSpotterSD_SimilarityCheck(HANDLE hCSpotter, char *lpszModelAddr, INT nSimChkLevel);

Parameters

hCSpotter(IN): Handle of the trainer.

lpszModelAddr(IN): The pointer of model buffer in **DATA FALSH**.

nSimChkLevel(IN): The similarity check level.

Return value

Return the voice tag ID or error code.

Remark

Similarity check aims to check the similarity between training voice tag and current commands in model. It will return the tag ID of similar voice tag, or return *CSPOTTER_ERR_Rejected*

Apply nSimChkLevel to determine the degree of stringency while checking command similarity.

CSpotterSD_DifferenceCheck

Purpose

Check whether the added utterance is different from the first one.

Prototype

INT CSpotterSD_DifferenceCheck(HANDLE hCSpotter, INT nDiffChkLevel);

Parameters

hCSpotter(IN): Handle of the trainer.

nDiffChkLevel(IN): The difference check level.

Return value

CSPOTTER_SUCCESS for success or negative error code otherwise.

Remark

While training voice tag, the input utterances must be the same content to improve the robustness of the voice tag. If the current utterance is the same as the first utterance, it will return *CSPOTTER_SUCCESS*, or return *CSPOTTER_ERR_Rejected*.

Apply nDiffChkLevel to determine the degree of stringency while checking the difference between the utterances.

CSpotterSD_TrainWord

Purpose

Train a voice tag into a command model for recognition.

Prototype

INT CSpotterSD_TrainWord(HANDLE hCSpotter, char *lpszModelAddr, INT nBufSize , INT nTagID, INT nRejLevel, INT *pnUsedSize);

Parameters

hCSpotter (IN): Handle of the trainer.

lpszModelAddr(IN/OUT): The pointer of model buffer in **DATA FALSH**.

nBufSize(IN): Size in bytes of the model buffer pointed by *lpszModelAddr*.

nTagID(IN): The voice tag ID.

nRejLevel(IN): Rejection level. The range is [0, 100]. The voice tag will be rejected if the quality is not good enough.

pnUsedSize(OUT): Size in bytes of the voice tag pointed by *lpszWordAddr*.

Return value

CSPOTTER_SUCCESS for success or negative error code otherwise.

Remarks

This function solely constitutes the training stage. Call this function after the data preparation stage consisting of *CSpotterSD_AddUttrStart(...)*,*CSpotterSD_AddSample(...)*, and *CSpotterSD_AddUttrEnd(...)* calls. Note that the output voice tag still cannot be used by CSpotter engine directly.

Model buffer pointed by *lpszModelAddr* is in the format of an acoustic model containing only user trained voice tags.*lpszModelAddr* can be pointing to internal data flash, external SPI flash with address mapping mechanism supported, or RAM simulating flash. For more information, please see remarks for *CspotterSD_AddUttrStart(…)*.

nBufSize contains 1KB header(H), and 2KB for each voice tag(T) times the maximum number(N) of voice tag.

$$nBufSize = H + N \cdot T$$

nTagID allows developers give unique ID for each voice tag. It is not the command ID, by definition, which is the order of the voice tag in the command model. Developers can get the command ID of the voice tag by calling *CSpotterSD_GetCmdID(…)*.

Normally, every add/remove command results in changed command ID. When recognizing, developers need to apply *CSpotterSD_getTagID(…)* to get tag ID after a recognition result is found from the recognizer to determine what the voice tag is.

The quality of voice tag may differ from environment. Apply nRejLevel to adjust the stringency while checking voice tag's quality. Larger number means higher quality requirement.

CSpotterSD_DeleteWord

Purpose

Remove a voice tag from the model for recognition.

Prototype

INT CSpotterSD_DeleteWord(HANDLE hCSpotter, char *lpszModelAddr, INT nTagID, INT *pnUsedSize);

Parameters

hCSpotter (IN): Handle of the trainer.

lpszModelAddr(IN/OUT): The pointer of model buffer in **DATA FALSH**.

nTagID(IN): The voice tag ID.

pnUsedSize(OUT): Size in bytes of the model pointed by *lpszModelAddr* after removing the voice tag.

Return value

CSPOTTER_SUCCESS for success or negative error code otherwise.

Remarks

When a voice tag is successfully removed from a model, the command ID for the other survival voice tags may be changed.

Parameter *lpszModelAddr* can be pointing to internal data flash, external SPI flash with address mapping mechanism supported, or RAM simulating flash. For more information, please see remarks for *CspotterSD_AddUttrStart(…)*.

5.4. Recognition

Functions in this section are designed to perform recognition process for the recognizer. The pseudo codes below show the calling sequence for recognizing the voice tags after traning:

```
// Declare shared memory using data type union in C.
union ShareMem {
        BYTE lpbyMemPool[N];      // N can be obtained with tool CSpotter_GetMemoryUsage
        SHORT lpsPlayBuffer[…];
        <Other buffers used by application>
} ShareMem;

// Declare and zero the memory for backing up the recognizer state.
BYTE lpbyState[M] = { 0 };          // M is about 96 bytes

DoVR(…)
{
        // Create a recognizer
        hCSpotter = Cspotter_Init_Sep(…, SDModel, ShareMem.lpbyMemPool, N, lpbyState,
        M, …);
        if (hCSpotter == NULL)
                goto L_ERROR;

        <Start Recording>

        CSpotter_Reset(…);

        while (1)
        {
                <Get PCM samples from recording device>
                if (CSpotter_AddSample(…) == CSPOTTER_SUCCESS)
                {
                        nID = CSpotter_GetResult(…);
                        CSpotter_GetResultEPD(…);                  // Optional
                        nScore = CSpotter_GetResultScore(…);       // Optional

                        // Optional, advanced version only
                        nTagID = CspotterSD_GetTagID(SDModel, nID);
                        break;
                }
        }

L_ERROR:

        <Stop Recording>

        CSpotter_Release(…);
        // Share memory ShareMem can be used by other functions after CSpotter_Release(…).

        <Play Prompt using memory ShareMem.lpsPlayBuffer >

}
```

CSpotterSD_GetCmdID

Purpose

Get the command ID of a voice tag.

Prototype

INT CSpotterSD_GetCmdID(char *lpszModelAddr, INT nTagID);

Parameters

lpszModelAddr(IN): The pointer of model buffer containing the input voice tag.

nTagID(IN): The voice tag ID.

Return value

Return the command ID when success or error code otherwise.

CSpotterSD_GetTagID

Purpose

Get the voice tag ID.

Prototype

INT CSpotterSD_GetTagID(char *lpszModelAddr, INT nCmdID);

Parameters

lpszModelAddr(IN): The pointer of model buffer containing the input voice tag.

nCmdID(IN): The command ID.

Return value

Return the voice tag ID when success or error code otherwise.

Remark

Developers can get the tag ID after a recognition result is found from the recognizer.

5.5. User-Implemented Flash Operation Functions

CSpotter trainer needs data flash to store the training utterances to train a voice tag. To optimize the resource usage to the most extent, we leave the flexibility to application developers to and manipulate the data flash. It is therefore developers' responsibility to correctly implement the flash access functions listed in this section. Though these functions are intended for accessing flash, developers can actually use RAM to simulate flash in the implementation if RAM is large enough.

DataFlash_Write

Purpose

Write data into data flash.

Prototype

INT DataFlash_Write(BYTE *lpbyDest, BYTE *lpbySrc, INT nSize);

Parameters

lpbyDest (OUT): The pointer of destination data buffer in **DATA FALSH**.

lpbySrc (IN): The pointer of source data buffer.

nSize(IN): Size in bytes of the source data buffer *lpbySrc*.

Return value

0 for success or negative error code otherwise.

DataFlash_Erase

Purpose

Erase the flash given the starting address and its size.

Prototype

INT DataFlash_Erase(BYTE *lpbyDest, INT nSize);

Parameters

lpbyDest (OUT): The pointer of destination data buffer in **DATA FALSH**.

nSize(IN): Size in bytes of the destination data buffer *lpbyDest*.

Return value

0 for success or negative error code otherwise.

Remarks

Trainer assumed the flash page size is 1KB currently. In other words, the input value of *nSize* is always a multiple of 1KB and pointer *lpbyDest* is 1KB aligned.

6. Cspotter SDK API Professional Version

6.1. About Professional Version

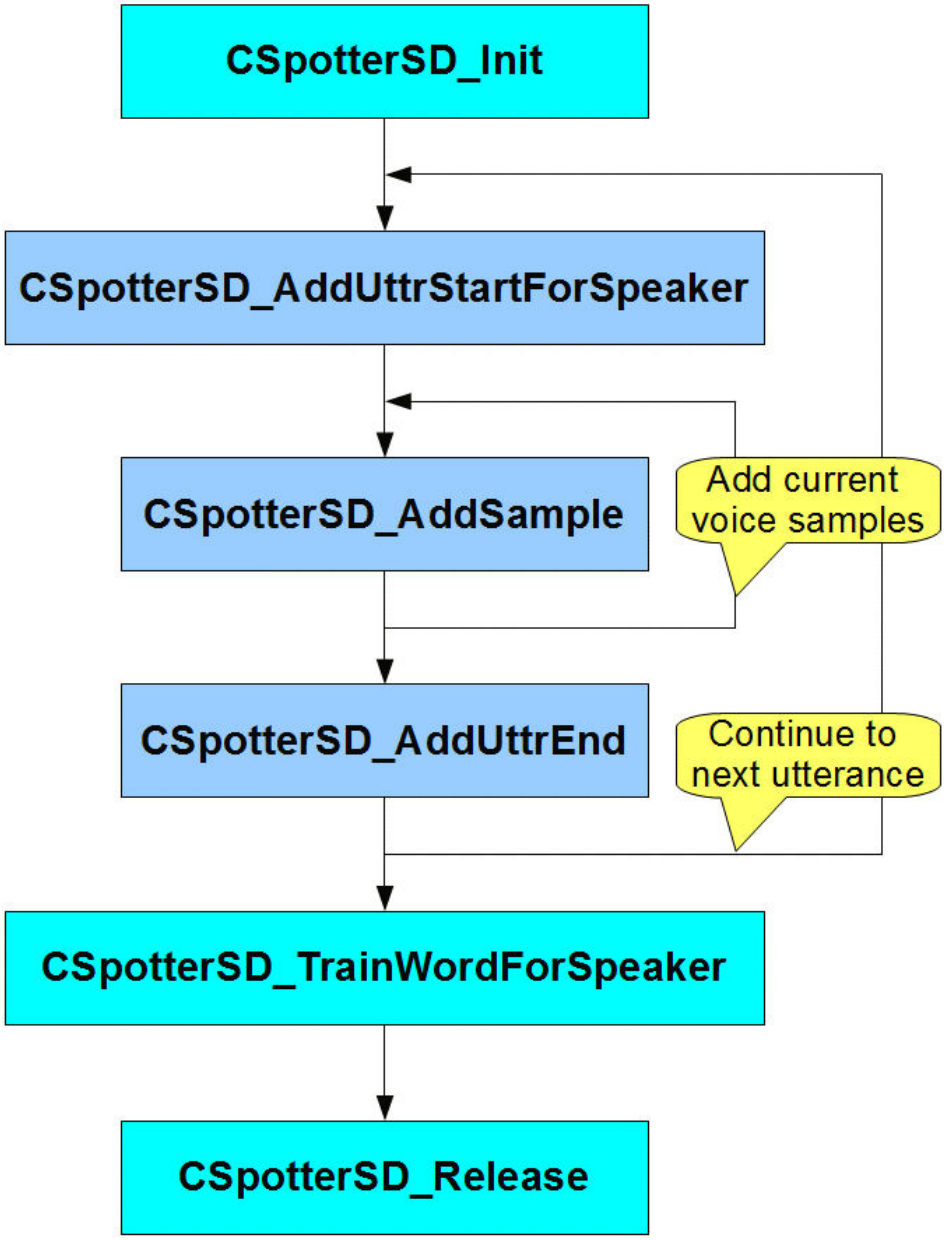
In the section of Professional Version talk about recognizer of speaker verification (SV) and scenarios of speaker verification. By definition, speaker verification is the process of accepting or rejecting the identity claim of a speaker. The professional version SDK is a set of extension functions based on advanced version.

Current SV technology belongs to context dependent category. That is, the recognizer will check speech content ("what" is being said) first and then speaker ("who" is saying it). In additional, SV model is not regular command model (SI/SD), SV model would not recognize the exact context of the command, it mainly use for determine the individual speaker.

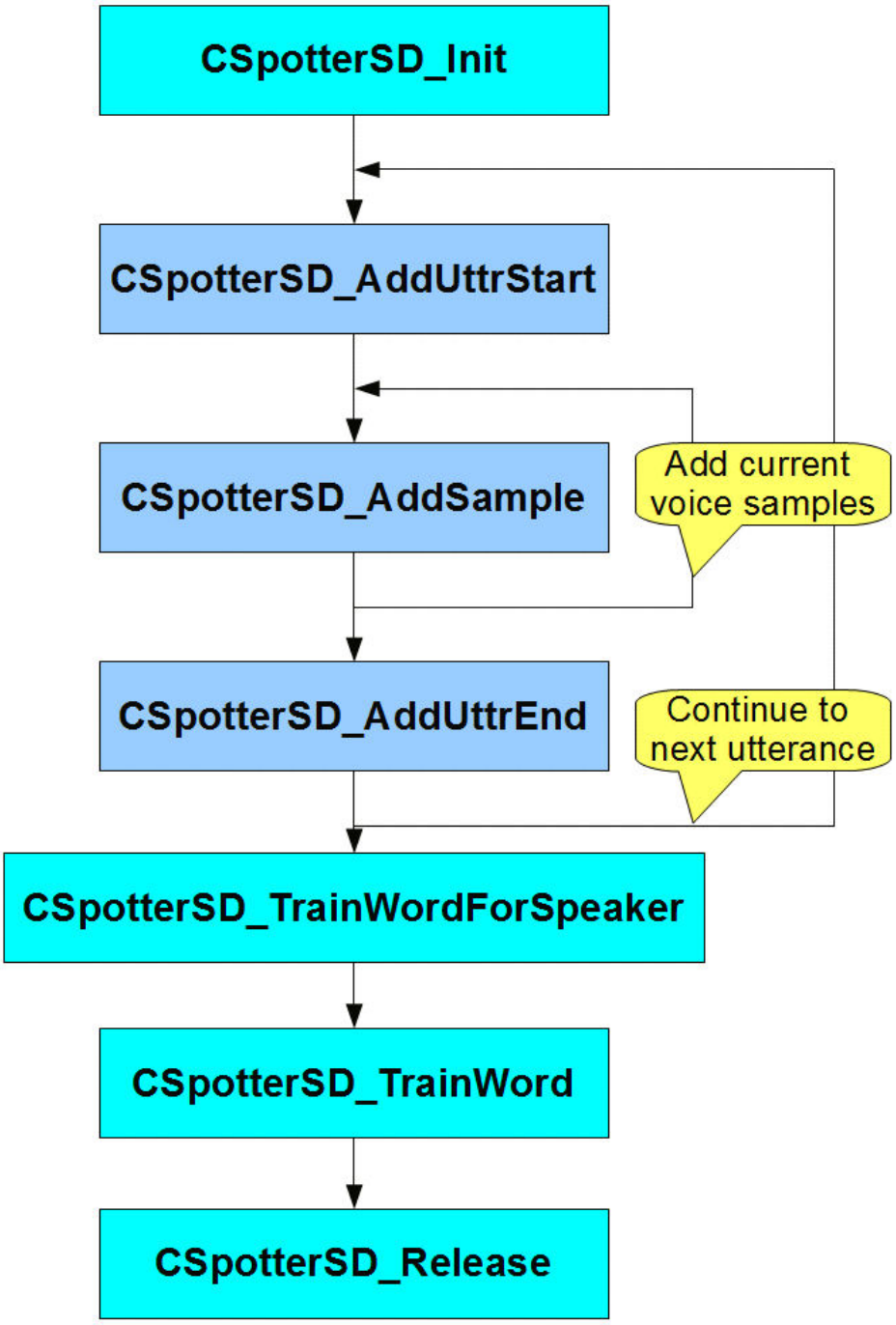
The scenarios of current SV technology are SI+SV or SD+SV. Developers may found more details in 6.2 Flow Chart.

Note: In SI+SV scenario, the content of training utterances must be the same as SI command.

6.2. Calling Flow Chart of Professional API
Scenario : SI + SV



Scenario : SD + SV



6.3. Training

Functions in this section are designed to perform training process for SV voice. The pseudo codes below show the calling sequence for scenario SI+SV:

```
// Declare shared memory using data type union in C.
union ShareMem {
	BYTE lpbyMemPool[N];	// N: obtained with tool CSpotterSD_GetMemoryUsage
	<Other buffers used by application>
} ShareMem;

DoVR_train(…)
{
	// Create a trainer
	hCSpotter = CSpotterSD_Init(…, ShareMem.lpbyMemPool, N, …);
	if (hCSpotter == NULL)
		goto L_ERROR;

	// Preparation stage: adding utterances to train a voice tag
	// k_nNumUttr shouldn't be less than 3.
	for (i = 0; i < k_nNumUttr; i++)
	{
		if (CSpotterSD_AddUttrStartForSpeaker(…) != CSPOTTER_SUCCESS)
			goto L_ERROR;

		<Start Recording>

		while (1)
		{
			<Get PCM samples from recording device>
			if (CSpotterSD_AddSample(…) != CSPOTTER_ERR_NeedMoreSample)
				break;
		}

		<Stop Recording>

		if (CSpotterSD_AddUttrEnd(…) != CSPOTTER_SUCCESS)
			goto L_ERROR;
	}

	// Training SV stage, and then add speaker tag to the model for recognition
	if (CSpotterSD_TrainWordForSpeaker(…, SVModel, …) != CSPOTTER_SUCCESS)
		<Error Handling …>

L_ERROR:

	CSpotterSD_Release(…);
}
```

The pseudo codes below show the calling sequence for scenario SD+SV:

```
DoVR_train(…)
{
    <Prepare storage (possibly in flash) for utterance buffer>

    // Create a trainer
    hCSpotter = CSpotterSD_Init(…, ShareMem.lpbyMemPool, N, …);
    if (hCSpotter == NULL)
        goto L_ERROR;

    // Preparation stage: adding utterances to train a voice tag
    // k_nNumUttr shouldn't be less than 3.
    for (i = 0; i < k_nNumUttr; i++)
    {
        if (i < 3)
        {
            if (CSpotterSD_AddUttrStart(…) != CSPOTTER_SUCCESS)
                goto L_ERROR;
        }
        else
        {
            if (CSpotterSD_AddUttrStartForSpeaker(…) != CSPOTTER_SUCCESS)
                goto L_ERROR;
        }

        <Start Recording>

        while (1)
        {
            <Get PCM samples from recording device>
            if (CSpotterSD_AddSample(…) != CSPOTTER_ERR_NeedMoreSample)
                break;
            // Use CSpotterSD_GetUttrEPD(…) to get the starting point of the input
            // utterance, and then start to compress it and write to data flash. (Optional)
            if (CSpotterSD_GetUttrEPD(…) == CSPOTTER_SUCCESS)
                <Compress recorded voice data and write it to data flash>
        }

        <Stop Recording>

        if (CSpotterSD_AddUttrEnd(…) != CSPOTTER_SUCCESS)
            goto L_ERROR;
        // Use CSpotterSD_GetUttrEPD(…) to get the ending point of the input utterance,
        // and move the compressed voice to external flash of larger size. (Optional)
        if (CSpotterSD_GetUttrEPD(…) == CSPOTTER_SUCCESS)
            <Move the compressed voice data from data flash to SPI flash>

        // Use CSpotterSD_SimilarityCheck(…) to check whether the input utterance is
        // similar with an existing voice tag or not. (Optional)
        if (i == 0 && CspotterSD_SimilarityCheck(…) >= 0)
            <Error Handling …>

        // Use CSpotterSD_DifferenceCheck(…) to check whether the input utterance is
        // different from the first one. (Optional)
        if (i > 0 && CspotterSD_DifferenceCheck(…) != CSPOTTER_SUCCESS)
            <Error Handling …>
    }
```

```
        // Training stage for speaker, and then add speaker tag to the model for recognition
        // it must be called before CSpotterSD_TrainWord(…)
        if (CSpotterSD_TrainWordForSpeaker(…, SVModel, …) != CSPOTTER_SUCCESS)
                <Error Handling …>

        // Training stage, and then add voice tag to the model for recognition
        if (CSpotterSD_TrainWord(…, SDModel, …) != CSPOTTER_SUCCESS)
                <Error Handling …>

L_ERROR:

        CSpotterSD_Release(…);
}
```

CspotterSD_AddUttrStartForSpeaker

Purpose

Prepare to add a new utterance for speaker.

Prototype

INT CSpotterSD_AddUttrStartForSpeaker(HANDLE hCSpotter);

Parameters

hCSpotter(IN): Handle of the trainer.

Return value

CSPOTTER_SUCCESS for success or negative error code otherwise.

Remarks

Current SV technology belongs to context dependent category. That is, the recognizer will check speech content ("what" is being said) first and then speaker ("who" is saying it). In additional, SV model is not regular command model (SI/SD), SV model would not recognize the exact context of the command, it mainly use for determine the individual speaker.

The training speaker tag flow is similar to training voice tag flow. *CSpotterSD_AddUttrStart(…)* allows recording maximum 3 utterances. For utterances exceed 3, call *CSpotterSD_AddUttrStartForSpeaker(…)* to record them. The quality of speaker tag has positive correlation with the amount of training utterances. More training utterances results in high speaker tag quality. The number of training utterances should be not less than 3.

CSpotterSD_TrainWordForSpeaker

Purpose

Train a speaker tag into a speaker model for recognition.

Prototype

INT CSpotterSD_TrainWordForSpeaker(HANDLE hCSpotter, char *lpszModelAddr, INT nBufSize, INT nTagID, INT *pnUsedSize);

Parameters

hCSpotter (IN): Handle of the trainer.

lpszModelAddr(IN/OUT): The pointer of model buffer in **DATA FALSH**.

nBufSize(IN): Size in bytes of the model buffer pointed by *lpszModelAddr*.

nTagID(IN): The speaker tag ID.

pnUsedSize(OUT): The model size.

Return value

CSPOTTER_SUCCESS for success or negative error code otherwise.

Remarks

This function solely constitutes the training stage for speaker. Call this function after the data preparation stage. For more information, please see remarks for *CSpotterSD_TrainWord(…)*.

Note: When training voice tag and speaker tag at the same time, this function must be called **before** *CSpotterSD_TrainWord(…).*

nBufSize contains 1KB header(H), and 5KB for each speaker tag(T) times the maximum number(N) of speaker tag.

$$nBufSize = H + N \cdot T$$

nTagID allows developers give unique ID for each speaker tag. For more information, please refer to remarks for *CSpotterSD_TrainWord(…)*.

6.4. Recognition

Functions in this section are designed to perform recognition process for the recognizer. The pseudo codes below show the calling sequence for recognizing command with speaker verification:

```
// Declare shared memory using data type union in C.
union ShareMem {
        BYTE lpbyMemPool[N];      // N can be obtained with tool CSpotter_GetMemoryUsage
        SHORT lpsPlayBuffer[…];
        <Other buffers used by application>
} ShareMem;

// Declare and zero the memory for backing up the recognizer state.
BYTE lpbyState[M] = { 0 };          // M is about 96 bytes

DoVR(…)
{
        // Create a recognizer
        hCSpotter = Cspotter_Init_Sep(…, SDModel, ShareMem.lpbyMemPool, N, lpbyState,
        M, …);
        if (hCSpotter == NULL)
                goto L_ERROR;

        if (CSpotter_SetSpeakerModel(…, SVModel) != CSPOTTER_SUCCESS)
                goto L_ERROR;

        <Start Recording>

        CSpotter_Reset(…);

        while (1)
        {
                <Get PCM samples from recording device>
                if (CSpotter_AddSample(…) == CSPOTTER_SUCCESS)
                {
                        nID = CSpotter_GetResult(…);
                        CSpotter_GetResultEPD(…);                    // Optional
                        nScore = CSpotter_GetResultScore(…);

                        // Optional, advanced version only
                        nTagID = CSpotterSD_GetTagID(SDModel, nID);
                        // Optional, professional version only
                        nSpkID = CSpotter_GetSpeakerResult (...);
                        break;
                }
        }

L_ERROR:

        <Stop Recording>

        CSpotter_Release(…);
        // Share memory ShareMem can be used by other functions after CSpotter_Release(…).

        <Play Prompt using memory ShareMem.lpsPlayBuffer >

}
```

CSpotter_SetSpeakerModel

Purpose

Set model of speaker verification in recognizer.

Prototype

INT CSpotter_SetSpeakerModel(HANDLE hCSpotter, BYTE *lpbySpeakerModel);

Parameters

hCSpotter(IN): Handle of the trainer.

lpbySpeakerModel(IN): The pointer of training model buffer in **DATA FALSH**.

Return value

CSPOTTER_SUCCESS for success or negative error code otherwise.

Remarks

Speaker model must be alone with command model (SI/SD) and developers can switch the recognizer to SV enable mode by calling this function. When SV enable mode, developers can get speaker verification result by calling *CSpotter_GetSpeakerResult(…)* after a recognition result was found.

If lpbySpeakerModel is NULL, the recognizer will switch to general command recognition.

Cspotter_SetSpeakerIdentLevel

Purpose

Set rejection level of speaker verification.

Prototype

INT CSpotter_SetSpeakerIdentLevel(HANDLE hCSpotter, INT nIdentLevel);

Parameters

hCSpotter(IN): Handle of the trainer.

nIdentLevel(IN): Rejection level. The range is [-100, 100], higher rejection level will make the engine more "picky" to return a result.

Return value

CSPOTTER_SUCCESS for success or negative error code otherwise.

Remarks

For more information, please see remarks for *CSpotter_SetRejectionLevel(…)*.

CSpotter_GetSpeakerResult

Purpose

Get the speaker verification result from the recognizer.

Prototype

INT CSpotter_*GetSpeakerResult*(HANDLE hCSpotter, INT *pnScore);

Parameters

hCSpotter (IN): Handle of the recognizer.

pnScore (OUT): The reliability score of the current recognition result for speaker.

Return value

Return the zero-based command ID when success or negative error code otherwise.

Remarks

For more information, please see remarks for *CSpotter_GetResult(…)* and *CSpotter_GetResultScore(…)*.

7. CSpotter SDK Error Code Table

Error Symbol	Error Code
CSPOTTER_ERR_IllegalHandle	-2001
CSPOTTER_ERR_IllegalParam	-2002
CSPOTTER_ERR_LeaveNoMemory	-2003
CSPOTTER_ERR_LoadDLLFailed	-2004
CSPOTTER_ERR_LoadModelFailed	-2005
CSPOTTER_ERR_GetFunctionFailed	-2006
CSPOTTER_ERR_ParseEINFailed	-2007
CSPOTTER_ERR_OpenFileFailed	-2008
CSPOTTER_ERR_NeedMoreSample	-2009
CSPOTTER_ERR_Timeout	-2010
CSPOTTER_ERR_InitWTFFailed	-2011
CSPOTTER_ERR_AddSampleFailed	-2012
CSPOTTER_ERR_BuildUserCommandFailed	-2013
CSPOTTER_ERR_MergeUserCommandFailed	-2014
CSPOTTER_ERR_IllegalUserCommandFile	-2015
CSPOTTER_ERR_IllegalWaveFile	-2016
CSPOTTER_ERR_BuildCommandFailed	-2017
CSPOTTER_ERR_InitFixNRFailed	-2018
CSPOTTER_ERR_EXCEED_NR_BUFFER_SIZE	-2019
CSPOTTER_ERR_Rejected	-2020
CSPOTTER_ERR_NoVoiceDetect	-2021
CSPOTTER_ERR_Expired	-2100

Error Symbol	Error Code
CSPOTTER_ERR_LicenseFailed	-2200
CSPOTTER_ERR_CreateModelFailed	-2500
CSPOTTER_ERR_WriteFailed	-2501
CSPOTTER_ERR_IDExists	-2502
CSPOTTER_ERR_NotEnoughStorage	-2503
CSPOTTER_ERR_TrainSpeakerModelFirst	-2504

8. CSpotter Supported Languages

Chinese (Taiwan)	Chinese (China)	English (US)
English (UK)	English (Australia)	English (Worldwide)
Japanese	Arabic	Bahasa
Cantonese	Czech	Danish
Dutch	Finnish	French
German	Greek	Hindi
Hungarian	Italian	Korean
Norwegian	Polish	Portuguese (Europe)
Portuguese (Brazil)	Russian	Slovak
Spanish (Europe)	Spanish (Latin American)	Swedish
Thai	Turkish	Ukrainian
Vietnam		