

ゲーム開発者向け

---

クイック

スタート

ガイド

## はじめに

---

このドキュメントでは、ASR C/API を使ってゲームに音声認識機能を追加する方法を簡潔に説明します<sup>1</sup>。音声認識技術を使って機能を拡張できるゲームの種類としては、アドベンチャー、ボードゲーム、シミュレーション ゲーム、RPG などが挙げられます。たとえば "X" というアドベンチャー ゲームがあると仮定します。アドベンチャー ゲームにおいて、プレイヤーは、物をとる、人と話す、物を使うなど、さまざまな行動 (アクション) をとります。従来、こうした行動をとるための一般的なインターフェイスとしてメニューを使用し、コントロール デバイスとしてゲームパッドを使用していました。音声認識技術を利用すると、たとえば、メニュー内を移動するのにボタンを押さなくても、アクションを起こすことが可能となります。

このドキュメントでは、ゲームの記述方法についてまでは触れていません。音声認識機能をアプリケーションに組み込むのに必要な関数についてのみ説明します。

## ゲーム

---

### 初期化

音声認識を使用するには、まず API を初期化する必要があります。そのためには、ヘッダー ファイルをインクルードし、`CasrAPIInit` 関数を呼び出す必要があります。

```
#include "ASR1600.h"

ERRORID err;
HAPI hApi;

...

COSCALLEBACKS OsCallBacks = {
    mCBMALLOC,
    mCBFREE,
    mCBREALLOC,
    mCBGETCURTASK,           /* カレントタスクの ID の取得。
                             マルチタスク環境でのみ有用です。 */
    mCBGETCURTHREAD,        /* カレントスレッドの ID の取得。
                             マルチスレッド環境でのみ有用です。 */
    mCBCREATECRITICALSECTION,
    mCBDELETECRITICALSECTION,
    mCBENTERCRITICALSECTION,
    mCBLEAVECRITICALSECTION
};

...

/* API の初期化 */
DWORD dwApiUserData = 0;
err = CasrAPIInit(&OsCallBacks, dwApiUserData, &hApi);
if (err!=ERR_SUCCESS) HaltOnError();
...
```

`OsCallBacks` 構造体には、プラットフォーム固有の処理 (メモリの割り当てなど) に使用する関数のアドレスが含まれます。

---

<sup>1</sup> このドキュメントの説明は、Consumer API 上の Macro API には適用されません。

dwApiUserData パラメータは、カプセル化クラスが存在する場合、そのアドレスを指定します。ここでは、クラスを使用しないため、パラメータの値は0 にしてかまいません。hApi パラメータには、この関数の呼び出しにより作成したインスタンスのハンドルが含まれます。このハンドル は、ほとんどの API 関数の呼び出しに必要なため、グローバル変数にした方が賢明です。

## 言語データ

初期化の次に、言語データをロードします。このアドベンチャー ゲームは、1 つの言語だけで実行するものとします。したがって、API の初期化が終了したら、言語情報を直接ロードし、アクティブにします。

```
...  
  
HDATA hData;  
  
...  
  
char* pData;  
ReadLanguageData(pData);           /* 言語データのロード */  
BOOL IsPermanent = TRUE;  
err = CasrImportData(hApi, (PDATA) pData, IsPermanent, &hData)  
if (err!=ERR_SUCCESS) HaltOnError();  
  
...
```

ReadLanguageData 関数はバッファを割り当て、その中に言語モデル (GDFS 上の.LNG ファイル) を格納します。そして、このバッファのポインタを CasrImportData 関数に渡します。これらの言語モデルは、通常、500 KB 以上のメモリを必要とします。ASR の使用頻度が高い場合には、アプリケーションの開始時に言語モデルをロードすることをお勧めします。

GDFS の言語データはアプリケーションによって RAM にコピーされなければ使えないので、IsPermanent の値は常に TRUE とします。IsPermanent パラメータを FALSE に設定した場合、エンジンは、バッファの内部コピーを作成します。これは、転送速度の遅い ROM から速い RAM にコピーする場合にしか役に立ちません。このパラメータの値を TRUE に設定した場合、言語データのハンドルを含む hData が無効な場合を除いては、バッファを解放してはいけません。

次のステップでは、エンジンをアクティブにします。CasrOpen 関数を呼び出す必要があります。その後、既にインポートされている言語データをアクティブにします。

```

...

HASR hAsr;

CRECOGCALLBACKS RecogCallBacks = {
    mCBRESULT,          /* 認識結果の通知 */
    mCBSTATE,           /* 状態の通知 */
    mCBTRAIN,           /* ユーザワードトレーニングの通知 */
    mCBABNORM,          /* 異常状態の通知 */
    mCBAGC,             /* アナログゲインの変更要求 */
    mCBASKCURRENTGAIN   /* 現在のゲイン設定値を返す */
};

...

/* 認識エンジンのオープン */
DWORD dwUserData = 0;
err = CasrOpen(hApi, &RecogCallBacks, dwUserData, &hAsr);
if (err!=ERR_SUCCESS) HaltOnError();
/* ここでエンジンは BOOT 状態になる */

/* エンジン上の言語をアクティベート */
err = CasrActivateData(hAsr, hData);
if (err!=ERR_SUCCESS) HaltOnError();
/* ここでエンジンは DATAREDAY 状態になる */

...

```

ここでもまた、関数のアドレスが定義された構造体を渡す必要があります。今回は主に通知関数です。例外は、プラットフォームに固有で、アプリケーション側で実装する必要がある AGC 関数です。この例では、dwUserData は使用しません (したがって値は 0 です)。通常この引数は、エンジンにリンクした構造体のアドレスを渡すのに使用します。コールバック関数は、DWORD 型ユーザー データの両方(dwApiUserData と dwUserData ) を含む USERDATA 構造体を返します。hAsr パラメータは、エンジンに適用される関数で必要となり、エンジンのハンドルが格納されます。このハンドルもグローバル変数にした方がよいでしょう。

## コンテキスト

コンテキストとは、コンパイルされた文法のこと、認識対象のすべての文 (構文 + ボキャブラリ) が含まれます<sup>2</sup>。

これで、実際の作業にとりかかる準備が整いました。次のステップでは、認識対象の単語 (コンテキスト) をどう選択するかを決定します。アドベンチャー ゲームでは、そのゲームの状況により、選択する単語が大きく異なってきます。たとえば、ゲーム中に、ドアの表示を読ませる場面があったとします。この場合、コンテキスト内には "表示を読む" コマンドが含まれている必要があります。次の場面では、表示はありません。したがって、"表示を読む" コマンドは必要ありません。API を使ってアクティブにできるコンテキストは 1 つだけです。各場面に合わせて異なるコンテキストを作成することも可能です。しかし、アドベンチャー ゲームには非常に多くの場面が設定されているため、この方法では解決が非常に複雑になります。解決策としては、同じコンテキスト内にすべてのコマンドを格納し、CasrSetActiveWords API 関数を使って必要なコマンドだけをアクティブにする方法があります。

<sup>2</sup> コンテキストと言語ファイルの詳細については、adt\_ug\_203.pdf を参照してください。

別の解決策として、コマンドごとにコンテキストを作成し、実行時に `CasrMergeContextsAndClasses` 関数を使ってコンテキストをマージする方法があります。しかしこの解決策は、あまりお勧めできません。したがって、以下に示すサンプルでは、単語の有効無効オプションを使用しています。なお、上記の 2 つの解決策を組み合わせでの使用することもできます。つまり、複数のコンテキストを作成して、必要に応じてマージし、必要な単語やコマンドだけをアクティブにすることも可能となります。

```
...  
  
HCONT hCont;  
  
...  
  
char* pCont;  
ReadContextData(pCont)          /* コンテキストのロード */  
  
BOOL IsPermanent = TRUE;  
  
err = CasrImportContext(hApi, (PCONT) pCont, IsPermanent, &hCont);  
if (err!=ERR_SUCCESS) HaltOnError();  
  
...
```

このサンプルは、一見すると、言語モデルのインポートに非常に似ています。

次に、アクティブにする単語を決定します。音声、メイン コントロール "デバイス" として使用しようとしているため、アクティブ化する単語を変更するには、認識結果を待つ必要があります。

```
...  
  
#define WID_EXIT          1  
#define WID_NEWGAME      2  
#define WID_RESUMESAVED  3  
  
CWORDID idWords[3] = { WID_EXIT,          /* "Exit"          */  
                        WID_NEWGAME,      /* "New Game"     */  
                        WID_RESUMESAVED }; /* "Resume Saved" */  
  
err = CasrSetActiveWords(hAsr, idWords, 3);  
if (err!=ERR_SUCCESS) HaltOnError();  
/* ここでエンジンは IDLE 状態になる */  
  
...
```

アクティブにするすべての単語 ID の配列を作成しておく必要があります。どんな ID が、コンテキスト内のコマンドに含まれるのかを確認するには、SDK の "showctxinfo.exe" を使用します。

## 認識ループ

エンジンは、現在、認識サイクルの最初のモードである IDLE モードになっています。CasrStart を呼び出して処理を開始し、エンジンを SLEEP モードにする必要があります。音声開始が検出されると、エンジンは自動的に RUN モードになります。音声開始の検出が無効化されている場合、エンジンは、SLEEP モードをスキップして、直接 RUN モードになります。一定時間以上の無音状態が検出されたり、CasrStop が呼び出された場合、エンジンは RECOVER モードとなり、認識結果がある場合にはそれを通知し、IDLE モードに戻ります。

このゲームでは、CasrStop を呼び出してユーザーの応答を待ち、結果が IDLE への状態変更通知が得られたら、音声入力を評価します。

```
...

err = CasrStart(hAsr);
if (err!=ERR_SUCCESS) HaltOnError();
/* ここでエンジンは SLEEP 状態になる */

StartRecordingDevice();

...

unsigned char mCBSTATE(CASRSTATE State, PUSERDATA pDummy)
{
    if (State==CASR_RECOVER) {
        StopRecordingDevice(); /* RECOVER 状態での計算に長い時間が
                                かった場合のバッファオーバーフローを避
                                けるため、録音デバイスを停止する。*/
    }
    return 0;
}

unsigned char mCBRESULT(CASRRESULT* pResult, PUSERDATA pDummy)
{
    EvaluateResult(pResult);
    return 0;
}

void EvaluateResult(CASRRESULT* pResult)
{
    if (!pResult) return;
    if (pResult->iNbr==0) return;
    SENTENCE* pSentence=pResult->pSentences+0;
    /* この例では簡単に、発話の最初の単語だけによって処理を決める。*/
    CWORDID Id=pSentence->pWords[0].pAlternatives[0].idWord;
    if (Id==WID_NEWGAME) {
        /* 必要な処理を実行 */
        ...
        SetNewGameState(ST_NEWGAME); /* ゲームのリセット */
    }
}
```

```

void StateNewGame()      /* ゲームの状況が変わるときに呼ばれる */
{
    /* 新しいコマンドセットをアクティブ化する */
    err = CasrSetActiveWords(hAsr, idWords, dwNbrWords);
    if (err!=ERR_SUCCESS) HaltOnError();

    err = CasrStart(hAsr);
    if (err!=ERR_SUCCESS) HaltOnError();
    /* ここでエンジンは SLEEP 状態になる */
    StartRecordingDevice();
    ...
}

...

```

## エンジンへのフィード

エンジンが SLEEP か RUN モードにある場合、録音された音声サンプルをエンジンに供給する必要があります。ここでよく使用される手法が、ループを 1 秒間に数回呼び出すというもの（例えば垂直回帰ループ）です。新しく録音した音声サンプルをこのループ内で供給することをお勧めします。

```

...

void FeedSamples(char* pSampleBuffer, int nSamples)
{
    err = CasrAcquisition(hAsr, PCM_16_11KHZ, (PVOID)pSampBuffer,
                          nSamples*2);
    if (err!=ERR_SUCCESS) HaltOnError();
    ClearSampleBuffer(pSampBuffer);
}

...

```

レコグナイザにフィードするバイト数 (nSamples) について、いくつか考慮すべき点があります。バッファが非常に小さい場合、ループ時間が長すぎると、オーバーフローが生じる可能性があります。たとえば、1 秒間に 60 回、エンジンにバッファをフィードする場合、少なくとも 368 バイト長のバッファが必要になります。これは、毎秒 16 ビットを 11025 回サンプリングするためです。念のため、バッファのサイズは大き目 (512 バイトなど) に設定して、オーバーフローを回避します。内部では、データは 10 ミリ秒単位で処理されます。つまり、レコグナイザが 110 のサンプルまたは 220 バイトを受け取ったときに、計算が開始されることとなります（それ以下のデータはキューに入れられます）。

ASR エンジン自体はリアル タイムでは動作しません。したがって、データがフィードされるスピードは問題とはならず、バッファ アンダーランが発生することはあり得ません。バッファ アンダーランが発生するのは、エンジンの処理速度にバッファの準備が追いつかない場合です。

バッファ オーバーランは、アンダーランとは逆の現象です。バッファは準備できているのに、エンジンが十分なスピードでデータを処理できない状態を指します。垂直回帰ごとに、利用可能なすべてのデータをフィードするようにすれば、オーバーランは発生しません (WSSIP バッファの容量が十分な場合)。

## エンジンの終了

認識エンジンが必要でなくなった場合、エンジンを終了する関数セットを呼び出します。

```
...

// 言語データとコンテキストデータの非アクティブ化
err = CasrActivateData(hAsr, NULL);

if (err!=ERR_SUCCESS) HaltOnError();

// コンテキストのクローズ
err = CasrCloseContext(hCont);

if (err!=ERR_SUCCESS) HaltOnError();

// 言語データのクローズ
err = CasrCloseData(hData);

if (err!=ERR_SUCCESS) HaltOnError();

// 認識エンジンのクローズ
err = CasrClose(hAsr);

if (err!=ERR_SUCCESS) HaltOnError();

// API のクローズと全リソースの開放
err = CasrAPIClose(hApi);

if (err!=ERR_SUCCESS) HaltOnError();

...
```

これらの関数セットは、すべて必要です。例として挙げたコードでは、最も重要な C/API 関数を単純化して使用していることに注意してください。このコードを実際のゲームに使用する場合には、もっと構造化されたコードにする必要があります。既存ゲームのコード (C、C++ イベント処理) により、どのように構造化するかは異なってきます。重要なのは、すべてのコールバック関数を定義すること、コールバック関数の一部はダミー関数 (ほとんど何もせず、常に成功を返す) でもよいということです。例えば、マルチスレッド用コールバックが必要になることは、まずありません。すべての関数の詳細については、『ドリームキャスト プラットフォーム用 ASR1600/C V2 ローレベル API 仕様書』を参照してください。



## ファイルの種類

---

- **Lng : 言語モデル ファイル**

このデータ ファイルには言語モデルが含まれます。日本語用とアメリカ英語用に、2 つのファイルが用意されています。8 ビットと 4 ビットに圧縮されたモデルがあります。CasrImportData() 関数を使って、このファイル内のデータをエンジンにインポートする必要があります。

- **Ctx : バイナリ コンテキストファイル**

このファイルのデータは、L&H 独自のバイナリ形式です。このファイルには、コンテキストに関する、コンパイルされた文法情報と音声情報が格納されています。このファイルには、CasrImportContext() 関数に渡されるデータが含まれます。ctx ファイルのデータは、8 ビットと 4 ビットの言語データと一緒に使用されます。

- **Bnf : コンテキスト文法仕様ファイル**

このファイルのデータは、L&H 独自の「バックナウアー形式」(Bachus Naur Form)です (adt\_ug\_203.pdf を参照)。Bnf は、LexTool や AsrBatchTool を使ってコンパイルし、エンジンに使用できる形式に変換する必要があります。LexTool は開発ツールの一部です。AsrBatchTool は、セガ社指定の開発ツールで提供されているコマンドライン ツールです。

- **Wcl : 単語およびクラスの文字列情報ファイル**

このファイルには、コンテキスト内のクラスや単語の文字列情報が含まれています。このファイルは、ドリームキャストプラットフォームでは使用できないため、まず.wrd と.cls に変換する必要があります。

- **Wrd : 単語文字列情報ファイル**

このファイルには、コンテキスト内の単語の文字列に関する情報が含まれます。ASR1600/C V2 では文字列は使用しません。エンジンは、メモリ使用量を減らし、工程を簡素化するために ID を使用します。このため、.ctx ファイルには、.bnf ファイルに存在する文字列情報が含まれません。この.wrd ファイルは、エンジンから返された ID に元の文字列を関連付けるのに使用されます。このリリースで提供されている ctxfuncs.c ソースにおいて、このファイル形式の使用方法が説明されています。このファイルは、¥tools ディレクトリの convwcl ツールを使って生成することができます。

- **Cls : クラス文字列情報ファイル**

このファイルには、コンテキスト内におけるクラスの文字列に関する情報が含まれます。この情報以外は、.wrd ファイルの説明と同じです。