

Game Developers

Quick Start

Guide

Introduction

The purpose of this document is to explain very briefly how to add speech recognition to a game using the ASR1600 C/API¹. There are several game types that could be extended with this technology; such as adventures, board games, simulation games, RPG's... For example, imagine we have an adventure game "X". In this kind of games the player can perform several actions, like picking up objects, talk to persons, use objects and many more. Common interfaces to access the actions are menus, with a gamepad as controlling device. We could for instance allow the user to formulate the action instead of just pushing buttons to navigate through a menu structure.

Note that this document will not tell you how to write games, I will only try to clarify some functions needed to enable speech recognition in your application.

The Game

Initialization

First, to enable the speech recognition we need to initialize the API. To do this, We need to include the header file and call the function `CasrAPIInit`.

```
#include "ASR1600.h"

ERRORID err;
HAPI hApi;

...

COSCALLEBACKS OsCallBacks = {
    mCBMALLOC,
    mCBFREE,
    mCBREALLOC,
    mCBGETCURTASK,                /* Get ID of current task. Only
                                useful in multitasking environments
                                */
    mCBGETCURTHREAD,            /* Get ID of current thread. Only
                                useful in multithreading
                                environments */
    mCBCREATECRITICALSECTION,
    mCBDELETECRITICALSECTION,
    mCBENTERCRITICALSECTION,
    mCBLEAVECRITICALSECTION
};
```

¹ This document does not apply to the Macro API on top of the Consumer API.

```

...

/* Initialize API */
DWORD dwApiUserData = 0;
err = CasrAPIInit(&OsCallbacks, dwApiUserData, &hApi);
if (err!=ERR_SUCCESS) HaltOnError();

...

```

The structure `OsCallbacks` contains addresses of functions for platform specific operations, for example allocation of memory.

The parameter `dwApiUserData` is intended for the address of the encapsulating class if there is one. Since we do not use it here it may be zero. The parameter `hApi` contains a handle to the instance we created by calling this function. It might be wise to make this a global variable, as we are going to need this handle in most API function calls.

Language Data

The next step is to load the language data. Our adventure game will run in one language only, so we could load and activate the language information directly after the initialization of the API.

```

...

HDATA hData;

...

char* pData;
ReadLanguageData(pData);          /* Load language data */
BOOL IsPermanent = TRUE;
err = CasrImportData(hApi, (PDATA) pData, IsPermanent, &hData)
if (err!=ERR_SUCCESS) HaltOnError();

...

```

The function `ReadLanguageData` will allocate the buffer and fill it with the language model (a .LNG file on the GDFS). Afterwards, we pass a pointer to this buffer to the `CasrImportData` function. These language models will typically take over 500KB of memory, so you probably want to load this at the beginning of the application, unless there is a long period where ASR will not be used.

Since an application has to copy the language data from GDFS to RAM, the `IsPermanent` parameter can always be TRUE. **If you set the `IsPermanent` parameter to FALSE**, the engine will make an internal copy of the buffer; this is only useful when copying from slow ROM to faster RAM. **When you set the value of this parameter to TRUE, you should not free the buffer as long as `hData`, containing the handle of the language data, is valid.**

The next step is activating the engine. We need to call `CasrOpen`, and afterwards we activate the language data we have imported earlier.

```

...

HASR hAsr;

CRECOGCALLBACKS RecogCallBacks = {
    mCBRESULT,          /* Result notification */
    mCBSTATE,           /* State notification */
    mCBTRAIN,           /* Userword training notification */
    mCBABNORM,          /* Abnormal Condition notification */
    mCBAGC,             /* Request to change the analog gain */
    mCBASKCURRENTGAIN   /* Returns the current gain setting */
};

...

/* Open a recognition engine */
DWORD dwUserData = 0;
err = CasrOpen(hApi, &RecogCallBacks, dwUserData, &hAsr);
if (err!=ERR_SUCCESS) HaltOnError();
/* The engine is now in BOOT state */

/* Activate the language on the engine */
err = CasrActivateData(hAsr, hData);
if (err!=ERR_SUCCESS) HaltOnError();
/* The engine is now in DATAREADY state */

...

```

Again we need to pass a structure with addresses of functions. In this case the functions are mainly notification functions, with the exception of the AGC functions because these are again platform specific and need to be implemented in the application. In this example we will not use `dwUserData` (hence the 0 value). Typically this argument is used to pass an address of a structure linked to an engine.

The callbacks will return the structure USERDATA containing both user data DWORDS (`dwApiUserData` and `dwUserData`).

The parameter `hAsr` is filled with a handle to the engine and will be needed in functions that apply to the engine. Again it might be useful to make this a global variable.

The Context

A context is a compiled grammar which represents all sentences (syntax + vocabulary) that can be recognized².

Now we are ready for the real work. The next step is to decide how we are going to select the words we want to recognize (the context). In an adventure game this is highly dependent on the situation in the game. For instance, at one point in the game there is a door sign that can be read, so we will need to have a command like “read sign” in the context. In the next scene there might be no sign, so the command “read sign” would be completely obsolete. Since the API only allows one context to be active, we could make different contexts for each situation. But adventure games contain a high number of these situations, so this might be a far too complex solution. An

² Other documents describe in more detail what contexts and language files are (see `adt_ug_203.pdf`)

alternative would be to put all commands in the same context, and activate only the needed ones using the `CasrSetActiveWords` API function. A second alternative is to make a context for each command and merge them at run time using `CasrMergeContextsAndClasses`. This is less interesting in this situation, and I will use the word enabling/disabling option in this sample. Of course, there is also the possibility to combine the latter two, by making a few contexts, merge them as needed and activate the needed words or commands.

```
...
HCONT hCont;
...
char* pCont;
ReadContextData(pCont)          /* Load Context */

BOOL IsPermanent = TRUE;

err = CasrImportContext(hApi, (PCONT) pCont, IsPermanent, &hCont);
if (err!=ERR_SUCCESS) HaltOnError();
...
```

As you can see this is very similar to the import of the language model.

Now it is time to decide which words we are going to activate. Since we are going to use speech as our main control “device”, we have to wait for a result to change the activated words.

```
...
#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();
/* The engine is now in IDLE state */
...
```

You need to build an array of all the word ID's you would like to activate. To find out what ID belongs to a command in your context, you can use `showctxinfo.exe`, which is provided with the SDK.

The Recognition Loop

Now the engine is in IDLE mode, which is the first state of the recognition cycle. We have to start the process by calling `CasrStart`, to put the engine in the SLEEP state. When begin of speech is detected the engine will go into RUN mode automatically. When begin of speech is disabled, the engine will go into RUN mode directly, skipping the SLEEP state. When a trailing silence is detected, or when `CasrStop` is called, the engine will go into the RECOVER state, notify the result if there is one, and go back into IDLE mode.

In our game this means that if we want to wait for the response of the user, we call `CasrStart`, wait for the result or IDLE state change notification and evaluate the input.

```
...

err = CasrStart(hAsr);
if (err!=ERR_SUCCESS) HaltOnError();
/* The engine is now in SLEEP state */

StartRecordingDevice();

...

unsigned char mCBSTATE(CASRSTATE State, PUSERDATA pDummy)
{
    if (State==CASR_RECOVER) {
        StopRecordingDevice(); /* Stop the recording device in
                               the recover mode, to avoid buffer
                               overflow on long recover
                               calculations */
    }
    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;
    /* For this simple example, we will parse based on first word
       of the utterance */
    CWORDID Id=pSentence->pWords[0].pAlternatives[0].idWord;
    if (Id==WID_NEWGAME) {
        /* do stuff */
        ...
        SetNewGameState(ST_NEWGAME); /* Game will reset */
    }
}
```

```

void StateNewGame()      /* Called when game is making the transition */
{
    /* Activate the new set of commands */
    err = CasrSetActiveWords(hAsr, idWords, dwNbrWords);
    if (err!=ERR_SUCCESS) HaltOnError();

    err = CasrStart(hAsr);
    if (err!=ERR_SUCCESS) HaltOnError();
    /* The engine is now in SLEEP state */
    StartRecordingDevice();
    ...
}
...

```

Feeding the engine

When the engine is in SLEEP or RUN mode, we have to supply recorded samples to the engine. Most game programmers use some kind of loop which is called a few times per second (like the vertical retrace loop). A good way is to supply newly recorded samples in this loop.

```

...

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

...

```

There are a few considerations to make about the amount of bytes you want to feed to the recognizer (nSamples). If the buffer is too small, an overflow might occur when the loop time is long. For example, if you feed the buffer to the engine 60 times per second, the buffer needs to be at least 368 bytes long because we sample 16bits 11025 times per second. Just to be on the safe side, always make the buffer a bit bigger (e.g. 512 bytes) to avoid overflow. Internally data is processed per 10 milliseconds. This means that when the recognizer receives 110 samples or 220 bytes it will start calculating (less data just gets queued). The ASR engine itself is not realtime at all. This means that the speed at which data is fed is totally irrelevant. This means that a buffer underrun cannot occur. By buffer underrun we mean that preparing the buffer takes a longer time than the processing by the engine. Buffer overrun is the other way round, the buffer is prepared, but the engine cannot process the data fast enough. If you feed all the available data per vertical retrace, this cannot occur (providing your WSSIP buffer is big enough).

Closing the Engine

Finally, when the recognition engine is not needed anymore, calling a set of functions can close it.

```
...

// Deactivate language and context data
err = CasrActivateData(hAsr, NULL);

if (err!=ERR_SUCCESS) HaltOnError();

// Close the context
err = CasrCloseContext(hCont);

if (err!=ERR_SUCCESS) HaltOnError();

// Close the language data
err = CasrCloseData(hData);

if (err!=ERR_SUCCESS) HaltOnError();

// Close the recognition engine
err = CasrClose(hAsr);

if (err!=ERR_SUCCESS) HaltOnError();

// Close API and free all resources
err = CasrAPIClose(hApi);

if (err!=ERR_SUCCESS) HaltOnError();

...
```

This is about all it takes. Note that the example code simplistically shows usage of the most important C/API functions. Integrating this into a real game will require some more structured code. This depends a lot on how the existing game code is like (C, C++ event handling). An important remark is that all callback functions must be defined, some of them may be dummy functions (doing not much more but always returning success) e.g. the multithreading callbacks will probably never be needed. For a more complete explanation of all functions refer to *the ASR1600/C Low Level Specification for Sega's DreamCast platform*.

File Types

- **Lng: Language model file**

These data files contain the language models. Per language (Japanese and American English) 2 files are provided. You have 8 bits and 4 bits compressed models. This is the data that needs to be given to the engine with the `CasrImportData()` function.

- **Ctx: Binary context file**

This file has an L&H proprietary binary format. It contains compiled grammar information and phonetic information about the context. These files contain the data needed to be given to the `CasrImportContext()` function. The .ctx data works with the 8bits and 4bits language data.

- **Bnf: Context grammar specification file**

These files have an L&H specific 'Bachus Naur Form' documented in `adt_ug_203.pdf`. Bnf's should be converted to a compiled engine useable format with the `LexTool` or `AsrBatchTool`. The `LexTool` is a part of the Development tools. `AsrBatchTool` is a commandline tool provided with the Sega specific development tools.

- **Wcl: Word & class string information file**

File that contains string information for the classes and words in a context. This file is not useable on the DreamCast platform and needs to be converted to a .wrp and .cls first.

- **Wrp: Word string information file**

Contains information about the character strings of the context words. The ASR1600/C V2 does not use strings. The engine works with ID's (to reduce memory and complexity). This means that the .ctx files do not contain the string information that is present in the .bnf. These .wrp files can be used to associate the original strings to the id's returned by the engine. The `ctxfuncs.c` source provided in this release shows how to use this fileformat. These files can be generated with the `convwcl` tool of the `\tools` directory.

- **Cls: Class string information file**

Contains information about the character strings of the context classes. The same remarks as on .wrp files apply.