Intel® Integrated Performance Primitives for Intel® Architecture Unified Speech Component Interface

Reference Manual

November 2009

Document Number: A314997-005US

World Wide Web: http://developer.intel.com

Version	Version Information	Date
-001	Original issue. Documents the Unified Speech Component Interface (USCI) implementation for the Intel® Integrated Performance Primitives (IPP) 5.2 Samples beta release.	09/2006
-002	Documents the USCI implementation for IPP 5.2 Samples gold release.	01/2007
-003	Documents the USCI implementation for IPP 5.3 Samples release. Changes to the Signal Filter API caused by incorporation of voice activity detection (VAD) algorithms have been documented. Appendix “USC Algorithms Supported by Intel® Integrated Performance Primitives” has been removed.	09/2007
-004	Documents the USCI implementation for IPP 6.0 Samples release.	08/2008
-005	Documents the USCI implementation for IPP 6.1 Samples release.	11/2009

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or by visiting Intel's Web Site.

Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. See http://www.intel.com/products/processor_number for details.

Contents

Overview 1

Unified Speech Component (USC) interface is designed for implementing speech codecs, echo cancellers and other algorithm modules in the C language using Intel® Integrated Performance Primitives (Intel® IPP).

About This Software

The purpose of the USC interface is to provide unified access to an algorithm module, the access being independent of the algorithm internals. USC interface also enables binaries to be easily integrated into existing software applications. Decoupling the interface and the algorithm details enables making development of system components independent of the algorithm implementation.

The USC interface defines a global table of unified functions that are applicable to a USC algorithm. The table can be augmented for future functionality expansions. Each USC algorithm *must* implement USC base functions and *may* implement algorithm-specific functions.

Currently the USC library implements the following types of algorithms:

- Speech codec
- Echo cancellation algorithm, also referred to as echo canceller
- Speech signal filter
- Tone detection and generation algorithm, also referred to as tone detector.

The interface defines base functions and their specializations for particular algorithm types as well as algorithm-specific functions for the algorithms.

Hardware and Software Requirements

The USC interface is based on Intel IPP, which determines hardware and software requirements for the interface. See the Intel IPP Release Notes for the specifics.

USC interface can be used in an application/library written in C or C++.

Platforms Supported

USC interface can be used on the Windows*, Linux*, and Mac OS* X platforms. All the code in this manual, that is, function declarations, type definitions and code examples, is written in the ANSI C style. However, versions of USC objects for different processors or operating systems may, of necessity, vary slightly.

Technical Support

The USC interface is subject to terms and conditions of support services provided for Intel IPP. For the latest information, including product features, white papers, and technical articles, check: http://developer.intel.com/software/products/

Intel also provides a support web site that contains a rich repository of self help information, including getting started tips, known product issues, product errata, license information, user forums, and more (visit http://support.intel.com/support/).

About This Manual

This manual provides basic concepts of the USC interface and describes functions defined in the interface.

Each function in reference sections in the document is introduced by its name, a short description of its purpose, and the calling sequence, or syntax, for each type of a USC algorithm with which the function is used. The following sections are also included:

Description Describes the function operation.

Parameters Specifies each parameter and its purpose.

Return Values Describes values indicating status codes set as a result of the function call.

Audience for This Manual

The manual is intended for C/C++ developers of speech processing algorithms for various operating systems. The audience should have thorough understanding of speech processing and be familiar with Intel IPP.

Manual Organization

The manual contains the following chapters and appendices:

Chapter 1	Overview. Introduces the Unified Speech Components interface, provides information on the manual organization, and explains notational conventions.
Chapter 2	Unified Speech Component Interface Concepts. Explains basic concepts of the USC interface and describes data structures that the interface uses.
Chapter 3	Base API. Provides a reference for USC base functions.
Chapter 4	Codec API. Provides a reference for USC codec functions.
Chapter 5	Echo Canceller API. Provides a reference for USC echo canceller functions.
Chapter 6	Signal Filter API. Provides a reference for USC signal filter functions.
Chapter 7	Tone Detector API. Provides a reference for USC tone detector functions.
Appendix A	Code Examples. Gives usage examples for the USC interface.

The manual also includes a Bibliography and Index.

Notational Conventions

This manual employs the following font conventions:

`UPPERCASE COURIER`	Used in the text for constant identifiers; for example, `USC_BadArgument`.
`lowercase courier mixed with UpperCase courier`	Code examples, function names, and data types; for example, `USC_Options`, `GetOutStreamSize`.
`lowercase courier mixed with UpperCase courier italic`	Parameters in function prototype and parameters description; for example, *`options`*.

Online Version

This manual is available in an electronic format (Portable Document Format, or PDF). To obtain a hard copy of the manual, print the file using the printing capability of the Adobe Acrobat* software, used for the online presentation of the document.

Unified Speech Component Interface Concepts 2

This chapter explains basic concepts of the Unified Speech Component (USC) interface and describes data structures that the interface uses.

USC Algorithm Interface

The interface for each USC algorithm module is defined as a global table of functions, which is exposed by the USC library or object file. The functions must be reentrant. The memory model of the USC interface is designed so that any USC algorithm can be implemented with no memory allocation inside the algorithm.

A USC algorithm is identified by a unique name no longer than 64 symbols. An application must link to a USC algorithm by the function table name (See examples in Appendix A, "Code Examples".)

Each USC algorithm

- *Must* implement mandatory functions, or USC *base functions.*
- *May* implement *algorithm-specific functions*.

Base functions. USC base functions, which each USC algorithm must implement, provide the following functionality. The `NumAlloc` and `MemAlloc` functions let an application know the algorithm memory requirements. The algorithm specifies its memory usage by filling in a memory table with the block sizes required. An application allocates memory blocks and passes the pointers to the `Init` function, which initializes the memory for the new algorithm instance. The `Reinit` function can be used to reinitialize the memory for an existing algorithm instance. Each algorithm instance requires its own memory allocation. The number of algorithm instances an application can create has no other restrictions than the amount of available memory. The `GetInfo` function provides general information on the type of algorithm or the current state of a given algorithm instance. The function `GetInfoSize` returns the size of memory to allocate for the parameter of the `GetInfo` function that holds the information. The `Control` function enables modifying the algorithm modes asynchronously.

Algorithm-specific functions. Functions inherent to algorithms of a particular type augment the base function list for algorithms of this type.

For example,

- A *USC speech codec* augments the base function list with four functions: the Encode and Decode functions perform the main codec operations. The Encode function encodes one single input PCM frame and produces one single bitstream frame of the corresponding length and type. The Decode function performs the operation inverse to the one of Encode. The GetOutStreamSize function is used to get the maximum output buffer size required for a given-size input buffer when encoding or decoding the stream. If a codec supports multiple frame sizes, the SetFrameSize function can be used to set desired frame size.
- A *USC echo canceller* has only one algorithm-specific function: CancelEcho.

Structures and Enumerators

USC interface functions operate on *base data types* and *algorithm-specific types*.

Base Structures and Enumerators

Below are base structures used by the USC interface.

Algorithm types

The `USC_AlgType` enumerator stores information on USC algorithm types:

`USC_Codec`	Codec algorithm
`USC_EC`	Echo cancellation algorithm
`USC_Filter`	Speech signal filter algorithm
`USC_TD`	Tone detection and generation algorithm.
`USC_FilterVad`	Voice activity detection (VAD) algorithm based on Signal Filter API.
`USC_T38`	T38 Fax algorithm.
`USC_MAX_ALG`	Predefined bound of USC algorithms numbering.

PCM types

The `USC_PCMType` structure, for storing parameters of a PCM stream supported by a USC algorithm, is defined as

```
typedef struct {
    int sample_frequency;
    int bitPerSample;
    int nChannels;
}USC_PCMType;
```

where the meaning of the parameters is as follows:

sample_frequency	The sample frequency in Hz: 8000, 16000, etc.
bitPerSample	The number of bits per sample: 8, 16, etc.
nChannels	The number of channels: 1 - mono, 2 - stereo.

PCM stream

The USC_PCMStream structure, for storing information on a specific PCM stream, is defined as

```
typedef struct {
     char        *pBuffer;
     int          nbytes;
     USC_PCMType  pcmType;
     int          bitrate;
        }USC_PCMStream;
```

where the meaning of the variables is as follows:

pBuffer	Pointer to the PCM stream in memory
nbytes	PCM stream length in bytes
pcmType	PCM type
bitrate	Desirable compression rate in bps.

USC memory types

The USC_MemType enumerator defines the USC memory types:

USC_OBJECT	Persistent memory
USC_BUFFER	Scratch memory
USC_CONST	Memory for tables, constants
USC_MAX_MEM_TYPES	The number of memory types.

USC memory space types

The USC_MemSpaceType enumerator defines the memory space types:

USC_NORMAL	Normal memory space

USC_MEM_CHIP	High speed memory
USC_MAX_SPACE	The number of memory space types

USC memory bank

The USC_MemBank structure, for storing memory buffer parameters, is defined as

```
typedef struct {
     char *pMem;
     int   nbytes;
     int   align;
     USC_MemType memType;
     USC_MemSpaceType memSpaceType;
         }USC_MemBank;
```

where the meaning of the parameters is as follows:

pMem	Pointer to the buffer
nbytes	Buffer size in bytes
align	Number of bytes to align memory to, for example, 16 to align to paragraph; 0 - do not care for alignment
memType	Memory type
memSpaceType	Memory space type, normal for PC

Codec Structures and Enumerators

This section presents structures used exclusively in USC Codec API.

Codec direction types

The USC_Direction enumerator stores information on possible direction types of a codec operation:

USC_ENCODE	Encoder
USC_DECODE	Decoder
USC_DUPLEX	Does both encoding and decoding
USC_MAX_DIRECTION_TYPES	The number of codec direction types.

Codec output modes

The USC_OutputMode enumerator stores information on codec output modes:

USC_OUT_NO_CONTROL	Default PCM mode
USC_OUT_MONO	Mono PCM output
USC_OUT_STEREO	Stereo PCM output
USC_OUT_COMPATIBLE	Compatibility mode
USC_OUT_DELAY	Delayed output
USC_MAX_OUTPUT_MODES	The number of codec output modes

Codec modes

The USC_Modes structure, used for storing information on the modes supported by a codec, is defined as

```
typedef struct {
   int bitrate;
   int truncate;
   int vad;
   int hpf;
   int pf;
   USC_OutputMode outMode;
}USC_Modes;
```

where the meaning of the variables is as follows:

bitrate	Compression rate in bit per second (bps)
truncate	If set to 1, indicates truncation of an input or output PCM to less than 16 bits.
vad	Voice activity detection modes supported by the codec; if set to zero, VAD mode is disabled.
hpf	If set to 1, indicates that high-pass filter is on.
pf	If set to 1, indicates that postfilter is on.
outMode	Codec output mode.

Bit rates

The USC_Rates structure, used as an entry in the table of supported compression rates, is defined as

```
typedef struct {
  int bitrate;
} USC_Rates;
```

where `bitrate` is the compression rate in bit per second (bps).

Codec options

The `USC_Option` structure, for storing codec options, that is, combinations of basic and additional modes of codec operation, is defined as

```
typedef struct {
   USC_Direction  direction;
   int            law;
   int            framesize;
   USC_PCMType    pcmType;
   int            nModes;
   USC_Modes      modes;
}USC_Option;
```

where the meaning of variables is as follows:

`direction` Direction(s) of the codec operation.

`law` For a PCM used as input to encode or output after decode, indicates the law to which the PCM conforms: 1- aLaw, 2 - µlaw, 0 - linear.

`framesize` Size of a codec frame in bytes, that is, the amount of PCM data (in bytes) that can be encoded by the `Encode` function.

`pcmType` PCM type supported by the codec.

`nModes` The number of modes (should be equal to `sizeof(USC_Modes)/sizeof(int)`).

`modes` Codec basic mode.

Codec general information

The `USC_CodecInfo` structure, for storing USC codec general information, is defined as

```
typedef struct {
  const char         *name;
  int                 maxbitsize;
  int                 nPcmTypes;
  USC_PCMType        *pPcmTypesTbl;
  int                 nRates;
  const USC_Rates    *pRateTbl;
  USC_Option          params;
}USC_CodecInfo;
```

The structure encapsulates the entire set of codec-specific parameters:

`name` Codec name, must be not longer than 64 symbols.

`maxbitsize` Longest bitstream frame that can be obtained by encoding of one input PCM frame.

`nPcmTypes` The number of PCM types supported.

`pPcmTypesTbl` Supported PCM types lookup table.

`nRates` The number of different compression rates supported by the codec (number of entries in `pRateTbl`).

`pRateTbl` Table of supported compression rates.

`params` Options supported by the codec type or the current options of an instance.

Bit stream

The `USC_Bitstream` structure, for storing parameters of a bit stream produced by the codec, is defined as

```
typedef struct {
   char        *pBuffer;
   int          nbytes;
   int          frametype;
   int          bitrate;
}USC_Bitstream;
```

where the meaning of the parameters is as follows:

`pBuffer` Pointer to the buffer containing bitstream data.

`nbytes` Bitstream length in bytes.

`frametype` Type of bitstream frame, codec-specific.

`bitrate` Bitstream compression rate in bit per second (bps).

Echo Canceller Structures and Enumerators

This section presents structures used exclusively in USC Echo Canceller (EC) API.

EC algorithm types

The `USC_EC_Algs` enumerator defines EC algorithm types:

`EC_SUBBAND`	Subband.
`EC_FULLBAND`	Full band.
`EC_FASTSUBBAND`	Fast subband.

EC adaptation types

The `USC_AdaptType` enumerator defines EC adaptation types:

`AD_OFF`	No adaptation
`AD_FULLADAPT`	Full adaptation
`AD_LITEADAPT`	Lite adaptation.

EC modes

The `USC_EC_Modes` structure, for storing modes supported by an echo canceller, is defined as

```
typedef struct {
    USC_AdaptType adapt;
    int zeroCoeff;
    int cng;
    int nlp;
    int td;
    int ah;
    int ap;
    int nr;
    int nr_smooth;
    int dcFlag;
}USC_EC_Modes;
```

where the meaning of the variables is as follows:

`adapt`	Type of the adaptation.
`zeroCoeff`	The use of zero filter's coefficient: 0 - disable, 1 - enable.
`cng`	The use of comfort noise generation (CNG): 0 - disable CNG, 1 - enable general-type CNG, 2 - enable SBF algorithm-specific CNG.
`nlp`	The use of non-linear processor (NLP):

0 - disable NLP,

1 - enable general-type NLP,

2 - enable subband fast (SBF) algorithm-specific NLP.

3 - general NLP with initial howling suppressor, SBF only.

td — The use of tone disabler (TD): 0 - disable, 1 - enable.

ah — The use of anti-howling (AH): 0 - disable, 1 - use cosine frequency shifting up for anti-howling.

ap — Order of the affine projection to be used. May have any positive integer value. 1 and 4 are recommended values.

nr — Noise reduction (NR) level: 0 - without NR, 1- Low NR, 2- Medium NR, 3- Normal NR, 4- High NR, default, 5- Auto NR. Basically, the bigger number the higher level of NR.

nr_smooth — NR smoothing mode:

0 - no smoothing filter after NR,

1 - static smoothing,

2 - dynamic smoothing.

dcFlag — Direct current (DC) offset compensation flag:
0 - DC offset is not compensated, 1 - DC offset is compensated.

EC options

The USC_EC_Option structure, for storing EC options, that is, combinations of EC operation modes, is defined as

```
typedef struct {
    USC_EC_Algs algType;
    USC_PCMType pcmType;
    int echotail;
    int framesize;
    int nModes;
    USC_EC_Modes modes;
}USC_EC_Option;
```

where the meanings of the variables is as follows:

algType — Type of the EC algorithm

pcmType — Supported PCM type.

nModes — Number of modes (should be equal to sizeof(USC_EC_Modes)/sizeof(int)).

`modes`	EC modes.
`echotail`	Maximal echo tail length supported, in ms.
`framesize`	EC frame size.

EC general information

The `USC_EC_Info` structure, encapsulating the entire set of EC-specific parameters, is defined as

```
typedef struct {
   const char     *name;
   int             nPcmTypes;
   USC_PCMType    *pPcmTypesTbl;
   USC_EC_Option  params;
}USC_EC_Info;
```

where the meaning of the variables is as follows:

`name`	EC name, must be not longer than 64 symbols.
`nPcmTypes`	The number of supported PCM types.
`pPcmTypesTbl`	Supported PCM type lookup table.
`params`	Options supported by the EC type or the current options of an instance.

Signal Filter Structures and Enumerators

This section presents structures used exclusively in USC Signal Filter API.

Frame Type

The `USC_Frame_Type` enumerator defines types of a signal filter frame:

`NODECISION`	Frame cannot be classified.
`ACTIVE`	Frame contains active voice.
`INACTIVE`	Frame does not contain active voice.
`END_OF_STREAM`	Last frame in the stream.

Filter modes

The `USC_FilterModes` structure, for storing modes supported by a USC filter, is defined as

```
typedef struct {
   int vad;
   int reserved2;
   int reserved3;
   int reserved4;
}USC_FilterModes;
```

where the meaning of the variable `vad` is as follows:

`vad`	The number of voice activity detection modes supported by the signal filter algorithm; if set to zero, VAD is not supported by the filter.

All of the other variables are reserved for future use.

Filter options

The `USC_FilterOption` structure for storing filter options, that is, combinations of basic and additional modes of filter operation, is defined as

```
typedef struct {
   int       minframesize;
   int       framesize;
   int       maxframesize;
   USC_PCMType  pcmType;
   USC_FilterModes modes;
}USC_FilterOption;
```

where the meaning of the variables is as follows:

`minframesize`	Minimum filter frame size in bytes.
`framesize`	Optimum filter frame size in bytes.
`maxframesize`	Maximum filter frame size in bytes.
`pcmType`	PCM type supported by the filter.
`modes`	Filter modes.

Note that the `framesize` value must be in the interval `[minframesize, maxframesize]`.

Filter general information

The `USC_FilterInfo` structure, encapsulating the entire set of filter-specific parameters, is defined as

```
typedef struct {
   const char          *name;
   int                 nOptions;
   USC_FilterOption    params[1];
}USC_FilterInfo;
```

where the meaning of the variables is as follows:

name	Filter name, must be not longer than 64 symbols
nOptions	The number of options supported by the filter
params	Array of options supported by the filter. Has length nOptions.

Tone Detector Structures and Enumerators

This section presents structures used exclusively in USC Tone Detector (TD) API.

Supported tones

The enumerator USC_ToneID defines IDs of tones supported by a USC TD:

```
USC_NoTone
USC_Tone_0
USC_Tone_1
USC_Tone_2
USC_Tone_3
USC_Tone_4
USC_Tone_5
USC_Tone_6
USC_Tone_7
USC_Tone_8
USC_Tone_9
USC_Tone_ASTERISK
USC_Tone_HASH
USC_Tone_A
USC_Tone_B
USC_Tone_C
USC_Tone_D
USC_ANS
USC_slashANS
USC_ANSam
USC_slashANSam .
```

In this enumeration, values are assigned according to [RFC2833] (value names originate from respective event names defined in this document).

TD modes

The USC_TD_Modes structure, for storing modes supported by a USC TD, is defined as

```
typedef struct {
   int reserved1;
   int reserved2;
   int reserved3;
   int reserved4;
}USC_TD_Modes;
```

Currently they are reserved for future use.

TD options

The USC_TD_Option structure, for storing USC TD options, that is, combinations of basic and additional modes of TD operation, is defined as

```
typedef struct {
   int                framesize;
   USC_PCMType        pcmType;
   USC_TD_Modes       modes;
}USC_TD_Option;
```

where the meaning of the variables is as follows:

pcmType PCM type supported by the tone detector

modes Tone detector basic mode

framesize A codec frame size in bytes, that is, the amount of PCM data in bytes that can be encoded by the DetectTone function

TD general information

The USC_TD_Info structure, encapsulating the entire set of TD-specific parameters, is defined as

```
typedef struct {
   const char         *name;
   int                nToneIDs;
   const USC_ToneID   *pToneIDsTbl;
   int                nOptions;
```

```
    USC_TD_Option      params[1];
}USC_TD_Info;
```

where the meaning of the variables is as follows:

name	Tone detector name, must be not longer than 64 symbols
nToneIDs	The number of tones that can be detected/generated
pToneIDsTbl	Supported detected/generated tones lookup table
nOptions	The number of options supported by tone detector
params	Array of options supported by the tone detector. Has length nOptions.

Error Reporting

The enumerator USC_Status defines all status codes that USC interface functions may return along with their values (see Table 2-1). Zero value indicates successful completion. Negative values indicate errors. Positive values correspond to warnings.

Table 2-1 USC Functions Return Codes

Status Code	Value	Comment
USC_BadArgument	-10	Wrong input data.
USC_UnsupportedEchoTail	-9	Unsupported echo tail length (for echo cancellers).
USC_NotInitialized	-8	Initialization was not performed prior to operation or initialization cannot be performed.
USC_InvalidHandler	-7	Algorithm instance handle is invalid.
USC_NoOperation	-6	Operation cannot be performed.
USC_UnsupportedPCMType	-5	Unsupported PCM type.
USC_UnsupportedBitRate	-4	Unsupported compression rate.
USC_UnsupportedFrameType	-3	Unsupported frame type.
USC_UnsupportedVADType	-2	Unsupported VAD mode.
USC_BadDataPointer	-1	Invalid data or buffer pointer.
USC_NoError	0	Operation completed successfully.
USC_StateNotChanged	1	Operation had no effect.

Base API 3

This chapter describes Unified Speech Component (USC) base functions. Each base function can be applied to a USC algorithm of any type and may have specific features for codecs, echo cancellers (ECs), filters and tone detectors. Accordingly, the discussion of each function expands upon the general usage as well as focuses on a particular algorithm specifics, if any.

The functions use type definitions supplied in the following header files:

Base type definitions	in `usc_base.h`
Codec-specific type definitions	in `usc.h`
EC-specific type definitions	in `usc_ec.h`
Filter-specific type definitions	in `usc_filter.h`
TD-specific type definitions	in `usc_td.h`

All the structures and enumerators defined in the header files are described in respective subsections of the Structures and Enumerators section.

USC Base Function Table

The USC interface defines a table of functions that are supported by all USC objects. Each USC object is visible only by its USC function table which is used for memory management, creation, and operation of an algorithm instances. The structure of a USC base function table is defined as follows:

```
typedef struct {
     USC_AlgType algType;
     USC_Status (*GetInfoSize)(int *pSize);
     USC_Status (*GetInfo)(USC_Handle handle, void *pInfo);
     USC_Status (*NumAlloc)(const void *memOptions, int *nbanks);
     USC_Status (*MemAlloc)(const void *memOptions, USC_MemBank *pBanks);
USC_Status (*Init)(const void *initOptions, const USC_MemBank
                    *pBanks, USC_Handle *handle );
USC_Status (*Reinit)(const void *reinitParams, USC_Handle handle );
USC_Status (*Control)(const void *controlParams, USC_Handle handle );
              } USC_stdFxns;
```

Algorithm types

In the USC base function table, algType may have values specified by the USC_algType structure definition in the Base Structures and Enumerators section).

USC Base Functions

Table 3-1 lists all USC base functions along with a short description of their operation. All the functions are reentrant.

Table 3-1 USC Base Functions

Function Name	Operation
GetInfoSize	Gets the size of a memory buffer that the GetInfo function requires.
GetInfo	Informs about algorithm features and instance status.
NumAlloc	Gets the number of memory blocks needed for the algorithm instance.
MemAlloc	Gets the sizes of the memory blocks required for the algorithm instance.
Init	Creates an algorithm instance and sets it to the initial state.
Reinit	Sets an algorithm to the initial state.
Control	Changes the algorithm instance modes on fly.

GetInfoSize

Gets the size of a memory buffer that the GetInfo function requires.

Syntax

```
USC_Status GetInfoSize(int *pSize);
```

Parameters

pSize	Pointer to the size of output buffer that the `GetInfo` function requires.

Description

The function returns the size of memory buffer the application should allocate for the *pInfo* parameter of the `GetInfo` function.

Return Values

`USC_BadDataPointer`	Indicates an error when the *pSize* pointer is NULL.

GetInfo

Informs about algorithm features and instance status.

Syntax

General syntax:

```
USC_Status GetInfo(USC_Handle handle, void *pInfo);
```

Codec:

```
USC_Status GetInfo(USC_Handle handle, USC_CodecInfo *pInfo);
```

EC:

```
USC_Status GetInfo(USC_Handle handle, USC_EC_Info *pInfo);
```

Filter:

```
USC_Status GetInfo(USC_Handle handle, USC_FilterInfo *pInfo);
```

TD:

```
USC_Status GetInfo(USC_Handle handle, USC_TD_Info *pInfo);
```

Parameters

handle	Pointer to the algorithm instance or NULL if the info for the algorithm type is queried.
pInfo	Pointer to the buffer to fill in with the information structure. The pointer must have size returned by `GetInfoSize`.

Description

If an algorithm instance is supplied, the function fills in the information structure with current status of the algorithm instantiation. The structure depends upon the algorithm type. If *handle* is NULL, the function reports the features and modes supported by any instance of an algorithm of the appropriate type, as described below:

- USC codec:
  - The number of VAD mode variations supported, for example, `USC_Modes.vad`=2 means that if enabled, the algorithm supports two VAD modes.
  - The number and table of compression rates supported.
  - Whether postfilter (`USC_Modes.pf`=1), high-pass filter (`USC_Modes.hpf`=1) or truncation modes are supported.
  - Maximal PCM frame length supported.
  - Directions supported: 0 – only encode, 1 – only decode, 2 – both.
  - Maximal bitstream frame size supported.
- USC EC:
  - Default algorithm type (`EC_SUBBAND`).
  - Default adaptation type (`AD_FULLADAPT`).
  - Default echo tail length (16).
  - Whether tone disabler (`USC_EC_Modes.td`=1), zero filter's coefficient (`USC_EC_Modes.zeroCoeff`=1), NLP (`USC_EC_Modes.nlp`=1), or antihowling (`USC_EC_Modes.ah`=1) are supported.
  - Maximal PCM frame length supported.

- USC signal filter:
  - — The number of supported filter options.
  - — For each option,
    - Maximal supported length of a PCM frame
    - Type of a PCM stream.
- USC TD:
  - — The number of supported TD options
  - — The number and table of tones supported.
  - — For each option,
    - Maximal supported length of a PCM frame
    - PCM stream type.

Return Values

`USC_BadDataPointer` Indicates an error when the *`pInfo`* pointer is NULL.

See Also

Base Structures and Enumerators.

Codec Structures and Enumerators.

Echo Canceller Structures and Enumerators.

Signal Filter Structures and Enumerators.

Tone Detector Structures and Enumerators.

NumAlloc

Gets the number of memory blocks needed for the algorithm instance.

Syntax

General syntax:

```
USC_Status NumAlloc(const void *memOptions, int *nbanks);
```

Codec:

```
USC_Status NumAlloc(const USC_Option *options, int *nbanks);
```

EC:

```
USC_Status NumAlloc(const USC_EC_Option *options, int *nbanks);
```

Filter:

```
USC_Status NumAlloc(const USC_FilterOption *options, int *nbanks);
```

TD:

```
USC_Status NumAlloc(const USC_TD_Option *options, int *nbanks);
```

Parameters

memOptions	Pointer to the input structure of options for the new instance to be created.
nbanks	Pointer to the output number of memory blocks required for the new instance.

Description

You must create the options structure in your application prior to the function call. The function returns the number of memory blocks that an algorithm instance might require for proper operation in the configuration determined by the supplied options.

Return Values

`USC_BadDataPointer`	Indicates an error when the *memOptions* or *nbanks* pointer is NULL.

See Also

Base Structures and Enumerators.

Codec Structures and Enumerators.

Echo Canceller Structures and Enumerators.

Signal Filter Structures and Enumerators.

Tone Detector Structures and Enumerators.

MemAlloc

Gets the sizes of the memory blocks required for the algorithm instance.

Syntax

General syntax:

```
USC_Status MemAlloc(const void *memOptions, USC_MemBank *pBanks);
```

Codec:

```
USC_Status MemAlloc(const USC_Option *options, USC_MemBank *pBanks);
```

EC:

```
USC_Status MemAlloc(const USC_EC_Option *options, USC_MemBank *pBanks);
```

Filter:

```
USC_Status MemAlloc(const USC_FilterOption *options, USC_MemBank *pBanks);
```

TD:

```
USC_Status MemAlloc(const USC_TD_Option *options, USC_MemBank *pBanks);
```

Parameters

memOptions, options	Pointer to the input structure of options for the new instance to be created. Must be the same as used in `NumAlloc`.
pBanks	Pointer to the output array having the `USC_MemBank` structure of length *nbanks*, determined by `NumAlloc`.

Description

The function fills in each element of the **pBanks* array with the size (in bytes) of a memory block required for the instance to be created.

Return Values

`USC_BadDataPointer`	Indicates an error when the *memOptions*(*options*) or *pBanks* pointer is NULL.

See Also

Base Structures and Enumerators.

Codec Structures and Enumerators.

Echo Canceller Structures and Enumerators.

Signal Filter Structures and Enumerators.

Tone Detector Structures and Enumerators.

Init

Creates an algorithm instance and sets it to the initial state.

Syntax

General syntax:

```
USC_Status Init(const void *initOptions, const USC_MemBank *pBanks, USC_Handle
    *handle);
```

Codec:

```
USC_Status Init(const USC_Option *options, const USC_MemBank *pBanks,
    USC_Handle *handle);
```

EC:

```
USC_Status Init(const USC_EC_Option *options, const USC_MemBank *pBanks,
    USC_Handle *handle);
```

Filter:

```
USC_Status Init(const USC_FilterOption *options, const USC_MemBank *pBanks,
    USC_Handle *handle);
```

TD:

```
USC_Status Init(const USC_TD_Option *options, const USC_MemBank *pBanks,
    USC_Handle *handle);
```

Parameters

initOptions, options Pointer to the input structure of algorithm-specific initialization options for the new instance to be created. Must be the same as used by the NumAlloc and MemAlloc functions.

pBanks Pointer to the input array of USC_MemBank structures that contain the pointers to the memory blocks allocated by the application according to the sizes reported by the MemAlloc function.

handle Pointer to the output codec instance. Has a USC object pointer type.

Description

You must create input structures in your application prior to the function call. Each memory block in _*pBanks_ must be allocated properly, that is, receive nbytes bytes of memory, and pMem pointers set. The function creates an instance of the algorithm of the appropriate type and returns its handle. The memory passed via the _*pBank_ array is for the exclusive use of the new algorithm instance and the function should not be used to reinitialize the memory.

Call the GetInfo function after a call Init for exact parameter values of the algorithm instance initialized.

Return Values

USC_BadDataPointer Indicates an error when any of the pointers *initOptions*(*options)*, *pBanks* or *handle* is NULL.

See Also

Base Structures and Enumerators.

Codec Structures and Enumerators.

Echo Canceller Structures and Enumerators.

Signal Filter Structures and Enumerators.

Tone Detector Structures and Enumerators.

Reinit

Sets an algorithm to the initial state.

Syntax

General syntax:

```
USC_Status Reinit(const void *reinitParams, USC_Handle handle);
```

Codec:

```
USC_Status Reinit(USC_Modes *modes, USC_Handle handle);
```

EC:

```
USC_Status Reinit(USC_EC_Modes *modes, USC_Handle handle);
```

Filter:

```
USC_Status Reinit(USC_FilterModes *modes, USC_Handle handle);
```

TD:

```
USC_Status Reinit(USC_TD_Modes *modes, USC_Handle handle);
```

Parameters

reinitParams, modes	Pointer to the structure containing the algorithm initialization options, that is, modes to be set.
handle	Instance handle. Has a USC object pointer type.

Description

The function initializes the codec instance and sets the requested modes. The instance continues occupying the same memory and its instance handle is not changed.

Return Values

`USC_BadDataPointer`	Indicates an error when the *reinitParams(modes)* or *handle* pointer is NULL.

See Also

Base Structures and Enumerators.

Codec Structures and Enumerators.

Echo Canceller Structures and Enumerators.

Signal Filter Structures and Enumerators.

Tone Detector Structures and Enumerators.

Control

Changes the algorithm instance modes on fly.

Syntax

General syntax:

```
USC_Status Control(const void *controlParams, USC_Handle handle);
```

Codec:

```
USC_Status Control(USC_Modes *modes, USC_Handle handle);
```

EC:

```
USC_Status Control(USC_EC_Modes *modes, USC_Handle handle);
```

Filter:

```
USC_Status Control(USC_FilterModes *modes, USC_Handle handle);
```

TD:

```
USC_Status Control(USC_TD_Modes *modes, USC_Handle handle);
```

Parameters

controlParams, modes	Pointer to the structure holding algorithm-specific control options, that is, modes to be set.
handle	Pointer to the input algorithm instance. Has a USC object pointer type.

Description

The function sets the algorithm instance to the requested modes. Once the algorithm modes are initialized by the `Init` (or `Reinit`) function, you can change them using the `Control` function.

Return Values

`USC_BadDataPointer`	Indicates an error when the *controlParams(modes)* or *handle* pointer is NULL.

See Also

Base Structures and Enumerators.

Codec Structures and Enumerators.

Echo Canceller Structures and Enumerators.

Signal Filter Structures and Enumerators.

Tone Detector Structures and Enumerators.

4

Codec API

This chapter describes the API of Unified Speech Component (USC) codecs. USC codec functions use base type definitions, supplied in the `usc_base.h` header file, as well as codec-specific type definitions, supplied in `usc.h`. All the structures and enumerators defined in the header files are described in respective subsections of the Structures and Enumerators section. All the USC codec functions are reentrant.

Codec Function Table

The USC codec function table augments the USC base function table (see the “USC Base Function Table” section) with codec-specific functions:

```
typedef struct {
    USC_stdFxns std;
    USC_Status (*Encode)(USC_Handle handle, USC_PCMStream *in,
      USC_Bitstream *out);
    USC_Status (*Decode)(USC_Handle handle, USC_Bitstream *in,
      USC_PCMStream *out);
    USC_Status (*GetOutStreamSize)(const USC_Option *options, int
      bitrate, int nbytesSrc, int *nbytesDst);
    USC_Status (*SetFrameSize)(const USC_Option *options, USC_Handle
      handle, int frameSize);
}USC_Fxns;
```

Codec Base Functions

The base (standard) functions of a USC codec algorithm are listed below along with their short description and calling syntax. For the detailed descriptions of the USC base (standard) functions, refer to Chapter 2, "Unified Speech Component Interface Concepts", where the discussion focuses on the codec-specific features of a function, if any.

GetInfoSize

Gets the size of a memory buffer that the `GetInfo` function requires.

```
USC_Status GetInfoSize(int *pSize);
```

GetInfo

Informs about algorithm features and instance status.

```
USC_Status GetInfo(USC_Handle handle, USC_CodecInfo *pInfo);
```

NumAlloc

Gets the number of memory blocks needed for the algorithm instance.

```
USC_Status NumAlloc(const USC_Option *options, int *nbanks);
```

MemAlloc

Gets the sizes of the memory blocks required for the algorithm instance.

```
USC_Status MemAlloc(const USC_Option *options, USC_MemBank *pBanks);
```

Init

Creates an algorithm instance and sets it to the initial state.

```
USC_Status Init(const USC_Option *options, const USC_MemBank *pBanks,
    USC_Handle *handle);
```

Reinit

Sets an algorithm to the initial state.

```
USC_Status Reinit(const USC_Modes *modes, USC_Handle handle);
```

Control

Changes the algorithm instance modes on fly.

```
USC_Status Control(const USC_Modes *modes, USC_Handle handle);
```

Codec-specific Functions

Table 4-1 lists USC codec-specific functions.

Table 4-1 USC Codec-Specific Functions

Function Name	Operation
`Encode`	Compresses an input PCM audio.
`Decode`	Decompresses a bit stream.
`GetOutStreamSize`	Reports the maximum size of the output stream.
`SetFrameSize`	Sets the PCM audio frame size.

Encode

Compresses an input PCM audio.

Syntax

```
USC_Status Encode(USC_Handle handle, USC_PCMStream *in, USC_Bitstream *out);
```

Parameters

handle	Algorithm instance handle. Has a USC object pointer type.
in	Pointer to the input PCM audio stream.
out	Pointer to the output bit stream.

Description

The function compresses one frame of the input PCM audio and forms the output compressed bitstream frame at the requested compression rate. The function sets *in*->`nbytes` to the frame size in bytes, that is the number of bytes actually encoded. The *out*->`nbytes` value is set to the length of the output compressed bitstream frame in bytes. *out*->`frametype` is set to a codec-dependent bitstream frame type. *out*->`bitrate` is set to the compression rate of the compressed bitstream.

Return Values

`USC_BadDataPointer`	Indicates an error when any of the pointers *in*, *out*, or *handle* is NULL.

See Also

Base Structures and Enumerators

Codec Structures and Enumerators.

Decode

Decompresses a bit stream.

Syntax

```
USC_Status Decode(USC_Handle handle, USC_Bitstream *in, USC_PCMStream *out);
```

Parameters

handle	Algorithm instance handle. Has a USC object pointer type.
in	Pointer to the input bit stream.
out	Pointer to the output PCM audio stream.

Description

The function decompresses one input bitstream frame and forms the output PCM audio frame. The function sets *in*->nbytes to the bitstream frame size in bytes, that is the number of bytes actually decoded. The *out*->nbytes is set to the length of the output decompressed PCM audio frame in bytes. If the input data parameter *in* is zero or *in*->pBuffer is zero, then a lost frame is to be decoded. If Packet Loss Concealment (PLC) is not supported by the codec, the status USC_NoOperation is returned and no output data produced. It is the responsibility of your application to remedy losses for a codec without PLC support.

Return Values

USC_BadDataPointer	Indicates an error when the *out*, *handle* pointer is NULL.
USC_NoOperation	Indicates an error in case mentioned in the Description section above.

See Also

Base Structures and Enumerators

Codec Structures and Enumerators.

GetOutStreamSize

Reports the maximum size of the output stream.

Syntax

```
USC_Status GetOutStreamSize(const USC_Option *options, int bitrate, int
    nbytesSrc, int *nbytesDst);
```

Parameters

options	Pointer to the options.
bitrate	Compression rate in bps
nbytesSrc	Input buffer size in bytes.
nbytesDst	Output buffer size in bytes.

Description

The function reports the size of the longest output buffer in *nbytesDst* for an input buffer size of *nbytesSrc*. The size can be for encoding or decoding of the stream. If *options*->direction = USC_ENCODE then the input stream is considered to be a PCM stream and *nbytesDst* specifies the encoded buffer size. Otherwise, when options->direction = USC_DECODE, the input stream is considered to be a bit stream and the longest output PCM stream size is returned in *nbytesDst*. In case of USC_DECODE, if options->modes.vad=0, the size, written to *nbytesDst*, is computed assuming that the input bitstream has no SID frames.

Return Values

USC_BadDataPointer	Indicates an error when the *options* or *nbytesDst* pointer is NULL.
USC_BadArgument	Indicates an error when *nbytesSrc* or *bitrate* is less or equal to 0.

See Also

Codec Structures and Enumerators.

SetFrameSize

Sets the PCM audio frame size.

Syntax

```
USC_Status SetFrameSize(const USC_Option *options, USC_Handle handle, int
   frameSize);
```

Parameters

options	Pointer to the options structure.
handle	Algorithm instance handle. Has a USC object pointer type.
frameSize	Desired input frame size in bytes.

Description

The function changes the codec frame size. Along with setting a new frame size, some codecs may change the bit rate. In this case, you can get a new bit rate using the `GetInfo` function.

In any case, call the `GetInfo` function after `SetFrameSize`, as the former will return the final, precise, value of the newly set frame size.

Return Values

`USC_BadDataPointer`	Indicates an error when the *options* or *handle* pointer is NULL
`USC_BadArgument`	Indicates an error when *frameSize* is less or equal to 0.
`USC_NoOperation`	Indicates that no frame size changes were made.

See Also

Codec Structures and Enumerators.

Echo Canceller API 5

This chapter describes the API of Unified Speech Component (USC) echo cancellers (EC). USC EC functions use base type definitions, supplied in the `usc_base.h` header file, as well as EC-specific type definitions, supplied in `usc_ec.h`. All the structures and enumerators defined in the header files are described in respective subsections of the Structures and Enumerators section. All the USC EC functions are reentrant.

EC Function Table

The USC EC function table augments the USC base function table (see the "USC Base Function Table" section) with only one EC-specific function:

```
typedef struct {
    USC_stdFxns std;
    USC_Status (*CancelEcho)(USC_Handle handle, short *pSin,
     short *pRin, short *pSout);
}USC_EC_Fxns;
```

EC Base Functions

The base (standard) functions of a USC EC algorithm are listed below along with their short description and calling syntax. For the detailed descriptions of the USC base (standard) functions, refer to Chapter 3, "Base API", where the discussion focuses on the EC-specific features of a function, if any.

GetInfoSize

Gets the size of a memory buffer that the `GetInfo` function requires.

```
USC_Status GetInfoSize(int *pSize);
```

GetInfo

Informs about algorithm features and instance status.

```
USC_Status GetInfo(USC_Handle handle, USC_EC_Info *pInfo);
```

NumAlloc

Gets the number of memory blocks needed for the algorithm instance.

```
USC_Status NumAlloc(const USC_EC_Option *options, int *nbanks);
```

MemAlloc

Gets the sizes of the memory blocks required for the algorithm instance.

```
USC_Status MemAlloc(const USC_EC_Option *options, USC_MemBank *pBanks);
```

Init

Creates an algorithm instance and sets it to the initial state.

```
USC_Status Init(const USC_EC_Option *options, const USC_MemBank *pBanks,
   USC_Handle *handle);
```

Reinit

Sets an algorithm to the initial state.

```
USC_Status Reinit(const USC_EC_Modes *modes, USC_Handle handle);
```

Control

Changes the algorithm instance modes on fly.

```
USC_Status Control(const USC_EC_Modes *modes, USC_Handle handle);
```

EC-specific Functions

Table 5-1 lists all USC EC-specific functions along with their short descriptions.

Table 5-1 USC EC-specific Functions

Function Name	Operation
CancelEcho	Performs echo cancellation of one input PCM audio frame.

CancelEcho

Performs echo cancellation in one input PCM audio frame.

Syntax

```
USC_Status CancelEcho(USC_Handle handle, short *pSin, short *pRin, short
    *pSout);
```

Parameters

handle	Algorithm instance handle. Has a USC object pointer type.
pSin	Pointer to send-in PCM data (input).
pRin	Pointer to receive-in PCM data (input).
pSout	Pointer to send-out PCM data (output).

Description

The function takes send-in PCM audio data, containing speech with echo, and receive-in data, containing echo, and removes echo from send-in. Result of the echo cancellation is stored in send-out.

Return Values

`USC_BadDataPointer`	Indicates an error when any of the pointers *pSin*, *pRin* or *pSout* is NULL.

See Also

Echo Canceller Structures and Enumerators.

Signal Filter API

6

This chapter describes the API of Unified Speech Component (USC) signal filters. USC signal filter functions use base type definitions, supplied in the `usc_base.h` header file, as well as filter-specific type definitions, supplied in `usc_filter.h`. All the structures and enumerators defined in the header files are described in respective subsections of the Structures and Enumerators section. All the USC signal filter functions are reentrant.

Filter Function Table

The USC signal filter function table augments the USC base function table (see the "USC Base Function Table" section) with two filter-specific functions:

```
typedef struct {
    USC_stdFxns std;
    USC_Status (*SetDlyLine)(USC_Handle handle, Ipp8s *pDlyLine);
    USC_Status (*Filter)(USC_Handle handle, USC_PCMStream *pIn,
      USC_PCMStream *pOut, USC_FrameType *pDecision);
}USC_Filter_Fxns;
```

Filter Base Functions

The base (standard) functions of a USC signal filter algorithm are listed below along with their short description and calling syntax. For the detailed descriptions of the USC base (standard) functions, refer to Chapter 3, "Base API", where the discussion focuses on the filter-specific features of a function, if any.

GetInfoSize

Gets the size of a memory buffer that the `GetInfo` function requires.

```
USC_Status GetInfoSize(int *pSize);
```

GetInfo

Informs about algorithm features and instance status.

```
USC_Status GetInfo(USC_Handle handle, USC_FilterInfo *pInfo);
```

NumAlloc

Gets the number of memory blocks needed for the algorithm instance.

```
USC_Status NumAlloc(const USC_FilterOption *options, int *nbanks);
```

MemAlloc

Gets the sizes of the memory blocks required for the algorithm instance.

```
USC_Status MemAlloc(const USC_FilterOption *options, USC_MemBank *pBanks);
```

Init

Creates an algorithm instance and sets it to the initial state.

```
USC_Status Init(const USC_FilterOption *options, const USC_MemBank *pBanks,
    USC_Handle *handle);
```

Reinit

Sets an algorithm to the initial state.

```
USC_Status Reinit(const USC_FilterModes *modes, USC_Handle handle);
```

Control

Changes the algorithm instance modes on fly.

```
USC_Status Control(const USC_FilterModes *modes, USC_Handle handle);
```

Filter-specific Functions

Table 6-1 lists all USC filter-specific functions along with their short descriptions.

Table 6-1 USC filter-specific Functions

Function Name	Operation
`SetDlyLine`	Copies the pointed vector to the internal delay line.
`Filter`	Filters input PCM data.

SetDlyLine

Copies the pointed vector to the internal delay line.

Syntax

```
USC_Status  SetDlyLine (USC_Handle handle, Ipp8s *pDlyLine);
```

Parameters

handle	Algorithm instance handle. Has a USC object pointer type.
pDlyLine	Pointer to the input delay line, that is a vector of type `signed char` and length equal to the algorithm frame size.

Description

The function copies the input vector to the internal delay line. The function requires the input data length to be not less than the algorithm frame size. A call to this function may be skipped. In case you do not call the function, zero delay line will be used.

Return Values

`USC_BadDataPointer`	Indicates an error when the `pDlyLine` pointer is NULL.
`USC_InvalidHandler`	Indicates an error when the handle is NULL.
`USC_NoOperation`	Indicates that the operation cannot be performed because the algorithm does not use internal delay line.

See Also

Signal Filter Structures and Enumerators.

Filter

Filters input PCM data.

Syntax

```
USC_Status  Filter (USC_Handle handle, USC_PCMStream *pIn, USC_PCMStream *pOut,
    USC_FrameType *pDecision);
```

Parameters

handle	Algorithm instance handle. Has a USC object pointer type.
pIn	Pointer to the input PCM audio stream.
pOut	Pointer to the output filtered PCM audio stream.
pDecision	Pointer to the output frame type.

Description

The function filters the input PCM data. The function requires that *pOut* has size sufficient to store the filtered signal frame.

Return Values

USC_BadDataPointer	Indicates an error when any of the pointers *pOut*, *pIn*, *pIn*->pBuffer or *pOut*->pBuffer is NULL.
USC_InvalidHandler	Indicates an error when the handle is NULL.

See Also

Base Structures and Enumerators

Signal Filter Structures and Enumerators.

Tone Detector API

7

This chapter describes the API of Unified Speech Component (USC) tone detectors (TDs). USC tone detection and generation functions use base type definitions, supplied in the `usc_base.h` header file, as well as TD-specific type definitions, supplied in `usc_TD.h`. All the structures and enumerators defined in the header files are described in respective subsections of the Structures and Enumerators section. All the USC TD functions are reentrant.

TD Function Table

The USC TD function table augments the USC base function table (see the "USC Base Function Table" section) with two TD-specific functions:

```
typedef struct {
    USC_stdFxns std;
    USC_Status (*DetectTone)(USC_Handle handle, USC_PCMStream *pIn,
      USC_ToneID *pDetectedToneID);
    USC_Status (*GenerateTone)(USC_Handle handle, USC_ToneID ToneID, int
      volume, int durationMS, USC_PCMStream *pOut);
}USC_TD_Fxns;
```

TD Base Functions

The base (standard) functions of a USC TD algorithm are listed below along with their short description and calling syntax. For the detailed descriptions of the USC base (standard) functions, refer to Chapter 3, "Base API", where the discussion focuses on the TD-specific features of a function, if any.

GetInfoSize

Gets the size of a memory buffer that the `GetInfo` function requires.

```
USC_Status GetInfoSize(int *pSize);
```

GetInfo

Informs about algorithm features and instance status.

```
USC_Status GetInfo(USC_Handle handle, USC_TD_Info *pInfo);
```

NumAlloc

Gets the number of memory blocks needed for the algorithm instance.

```
USC_Status NumAlloc(const USC_TD_Option *options, int *nbanks);
```

MemAlloc

Gets the sizes of the memory blocks required for the algorithm instance.

```
USC_Status MemAlloc(const USC_TD_Option *options, USC_MemBank *pBanks);
```

Init

Creates an algorithm instance and sets it to the initial state.

```
USC_Status Init(const USC_TD_Option *options, const USC_MemBank *pBanks,
    USC_Handle *handle);
```

Reinit

Sets an algorithm to the initial state.

```
USC_Status Reinit(const USC_TD_Modes *modes, USC_Handle handle);
```

Control

Changes the algorithm instance modes on fly.

```
USC_Status Control(const USC_TD_Modes *modes, USC_Handle handle);
```

TD-specific Functions

Table 7-1 lists all USC TD-specific functions along with their short descriptions.

Table 7-1 USC TD-specific Functions

Function Name	Operation
DetectTone	Detects supported tones in a given input PCM audio.
GenerateTone	Generates a tone of a given ID and volume.

DetectTone

Detects supported tones in a given input PCM audio.

Syntax

```
USC_Status DetectTone (USC_Handle handle, USC_PCMStream *pIn, USC_ToneID
    *pDetectedToneID);
```

Parameters

handle	Algorithm instance handle. Has a USC object pointer type.
pIn	Pointer to the input PCM audio stream.
pDetectedToneID	Pointer to the output tone ID.

Description

The function detects supported tones in a given input PCM audio and forms the output tone ID. If no tone was detected in the input PCM stream, the function returns `USC_NoTones` value in **pDetectedToneID*. Tone detection occurs when necessary signal length is reached. The function detects one tone in a call. To detect all tones in the input PCM audio, keep calling the function while tones are detected.

Return Values

`USC_BadDataPointer`	Indicates an error whenany of the pointers *pIn*, *pDetectedToneID*, or *handle* is NULL.

See Also

Base Structures and Enumerators

Tone Detector Structures and Enumerators.

GenerateTone

Generates a tone of a given ID and volume.

Syntax

```
USC_Status GenerateTone (USC_Handle handle, USC_ToneID ToneID, int volume, int
    durationMS, USC_PCMStream *pOut);
```

Parameters

handle Algorithm instance handle. Has a USC object pointer type.

ToneID Tone ID to generate.

volume The power level of the tone after dropping the sign. A positive value measured in dBm0 and varying from 0 to 63.

durationMS Length of tone in milliseconds.

pOut Pointer to the output PCM audio stream with generated tone.

Description

The function generates tone of a given tone ID and volume and forms the output PCM audio frame. The function requires *pOut* to have size sufficient to store *durationMS* milliseconds of generated signal. If tone is not supported, then the status `USC_NoOperation` is returned and no output data produced.

NOTE. It is the responsibility of the application to pass to the function the length sufficient for the tone to be valid.

Return Values

`USC_BadDataPointer` Indicates an error when the *pOut* or *handle* pointer is NULL.

`USC_BadDataRange` Indicates an error when *volume* or *durationMS* parameters are out of range.

`USC_NoOperation` See the Description section above.

See Also

Base Structures and Enumerators

Tone Detector Structures and Enumerators.

Code Examples

A

This chapter gives Unified Speech Component interface usage examples.

The C code below demonstrates how a raw PCM file can be compressed and decompressed by a USC codec.

Example A-1 Processing a Raw PCM with USC Codec

```
/*
    Some declarations and definitions
*/
#include "usc.h"
#include <stdlib.h>
#define PCMDATA_LEN 10000 /* length of the input PCM data */
static int lenSrc = PCMDATA_LEN;
static unsigned char foo[PCMDATA_LEN]={0}; /* foo: supposed to be filled
with input PCM data */
static unsigned char fooUncompressed[PCMDATA_LEN]; /* uncompressed PCM
after encoding and decoding in a chain */
static char* outputBuffer = NULL;
extern USC_Fxns USC_G729I_Fxns;     /* For example, linking to the G729I
codec */
USC_Fxns *USC_Gxxx_Fnxs = &USC_G729I_Fxns;
```

Example A-1 Processing a Raw PCM with USC Codec

```
/*
    main program: foo data encoding & decoding
*/
int main(int argc, char *argv[]){
    /* Local variables */

    USC_CodecInfo *pInfo;
    int infoSize;
    int i, nbanksEnc,nbanksDec;
    USC_MemBank* pBanksEnc;
    USC_MemBank* pBanksDec;
    USC_Handle hUSCEncoder;
    USC_Handle hUSCDecoder;
    int lenDst, bytesRemaining;
    char* tmpInputBuff=NULL;
    char* tmpOutputBuff=NULL;
    char* tmpOutputPCMBuff=NULL;

    /* Get the size of the Gxxx codec status structure */
    if(USC_NoError != USC_Gxxx_Fnxs->std.GetInfoSize(&infoSize)) exit(1);

    pInfo = (USC_CodecInfo *)malloc(infoSize);

    /* Get the Gxxx codec status */
    if(USC_NoError != USC_Gxxx_Fnxs->std.GetInfo((USC_Handle)NULL,
pInfo)) exit(1);
    /*
        creation of an encoder instance
    */
    pInfo->params.direction = USC_ENCODE;            /* Direction: encode */
    pInfo->params.modes.vad = 0;           /* Suppress silence compression */
```

Example A-1 Processing a Raw PCM with USC Codec

```
    pInfo->params.law = 0;                    /* Linear PCM input */
    pInfo->params.modes.bitrate =
    pInfo->pRateTbl[pInfo->nRates-1].bitrate;  /*For example, set the
    highest bitrate*/

    /* Determine how many memory blocks the encoder needs  */
    if(USC_NoError != USC_Gxxx_Fnxs->std.NumAlloc(&pInfo->params,
&nbanksEnc)) exit(2);



    /* allocate memory for the memory bank table */
    pBanksEnc = (USC_MemBank*)malloc(sizeof(USC_MemBank)*nbanksEnc);



    /* Determine the size of each block  */
    if(USC_NoError != USC_Gxxx_Fnxs->std.MemAlloc(&pInfo->params,
pBanksEnc)) exit(3);



    /* allocate memory for each block */
    for(i=0; i<nbanksEnc;i++){
        pBanksEnc[i].pMem = (unsigned char*)malloc(pBanksEnc[i].nbytes);
    }



    /* Create the encoder instance */
    if(USC_NoError != USC_Gxxx_Fnxs->std.Init(&pInfo->params, pBanksEnc,
&hUSCEncoder)) exit(4);

    /* Get the status of the encoder instance  */
    if(USC_NoError != USC_Gxxx_Fnxs->std.GetInfo(hUSCEncoder, pInfo)) exit(1);
```

Example A-1 Processing a Raw PCM with USC Codec

```
    /* Determine the size of the maximum output bitstream */
    if(USC_NoError != 
USC_Gxxx_Fnxs->GetOutStreamSize(&pInfo->params,pInfo->params.modes.bitra
te, lenSrc, &lenDst)) exit(5);



    /* allocate the output bitstream buffer */
    outputBuffer = (char*)malloc(lenDst);



    /*
        Creation of the decoder instance
    */
    pInfo->params.direction = USC_DECODE;       /* Direction: decode */



    /* Determine how many memory blocks the decoder needs */
    if(USC_NoError != USC_Gxxx_Fnxs->std.NumAlloc(&pInfo->params,
&nbanksDec)) exit(2);



    /* allocate memory for the memory bank table */
    pBanksDec = (USC_MemBank*)malloc(sizeof(USC_MemBank)*nbanksDec);



    /* Determine the size of each block */
    if(USC_NoError != USC_Gxxx_Fnxs->std.MemAlloc(&pInfo->params,
pBanksDec)) exit(3);
```

Example A-1 Processing a Raw PCM with USC Codec

```
    /* allocate memory for each block */
    for(i=0; i<nbanksDec;i++){
        pBanksDec[i].pMem = (unsigned char*)malloc(pBanksDec[i].nbytes);
    }
    /* Create the decoder instance */
    if(USC_NoError != USC_Gxxx_Fnxs->std.Init(&pInfo->params, pBanksDec,
&hUSCDecoder)) exit(4);

    /*
        Ready to encode and decode.

        Now set initial data pointers.
    */
    bytesRemaining = lenSrc;
    tmpInputBuff = foo;
    tmpOutputBuff = outputBuffer;
    tmpOutputPCMBuff = fooUncompressed;



    /* Main encoding & decoding loop */

    while(bytesRemaining >= pInfo->params.framesize) {
        USC_PCMStream in, outpcm;
        USC_Bitstream out;



        /* Set input stream parameters */
        in.bitrate = pInfo->params.modes.bitrate;
        in.nbytes = bytesRemaining;
        in.pBuffer = tmpInputBuff;
        in.pcmType.bitPerSample = pInfo->params.pcmType.bitPerSample;
        in.pcmType.nChannels = pInfo->params.pcmType.nChannels;
```

Example A-1 Processing a Raw PCM with USC Codec

```
            in.pcmType.sample_frequency = pInfo->params.pcmType.sample_frequency;


            /* Set the output buffer */
            out.pBuffer = tmpOutputBuff;
            outpcm.pBuffer = tmpOutputPCMBuff;


            /* Encode a frame  */
           if(USC_NoError != USC_Gxxx_Fnxs->Encode (hUSCEncoder, &in, &out))
    exit(6);

            /* Decode a frame */

            if(USC_NoError != USC_Gxxx_Fnxs->Decode (hUSCDecoder, &out,
    &outpcm)) exit(6);


            /* Move to the next frame */
            tmpInputBuff += in.nbytes;
            tmpOutputBuff += out.nbytes;
            tmpOutputPCMBuff += outpcm.nbytes;


            /* calculate the size of the remaining PCM data */
            bytesRemaining -= in.nbytes;
        }
```

Example A-1 Processing a Raw PCM with USC Codec

```
    /*
        Release memory of the encoder instance
    */
    for(i=0; i<nbanksEnc;i++){
    free(pBanksEnc[i].pMem);
    }
    free(pBanksEnc);
    /*
        Release memory of the decoder instance
    */
    for(i=0; i<nbanksDec;i++){
        free(pBanksDec[i].pMem);
    }
    free(pBanksDec);
    free(pInfo);
}
```

The C code below demonstrates how to use `Reinit` and `Control` functions.

Example A-2 Using ReInit and Control Functions

```
/*
Some declarations and definitions
*/
#include "usc.h"
#include <stdlib.h>

#define PCMDATA_LEN 10000 /* length of the input PCM data  */
static int lenSrc = PCMDATA_LEN;
```

Example A-2 Using ReInit and Control Functions

```
extern USC_Fxns USC_G729I_Fxns; /* For example, linking to the G729I 
codec */
USC_Fxns *USC_Gxxx_Fnxs = &USC_G729I_Fxns;

/*
main program: foo codec initialization and re-initialization
*/
int main(int argc, char *argv[]){
   /* Local variables */
   USC_CodecInfo *pInfo;
   int infoSize;
   int i, nbanksEnc;
   USC_MemBank* pBanksEnc;
   USC_Handle hUSCEncoder;

   /* Get the size of the Gxxx codec status structure */
   if(USC_NoError != USC_Gxxx_Fnxs->std.GetInfoSize(&infoSize)) exit(1);

   pInfo = (USC_CodecInfo *)malloc(infoSize);

   /* Get the Gxxx codec status */
   if(USC_NoError != USC_Gxxx_Fnxs->std.GetInfo((USC_Handle)NULL,pInfo)) exit(2);
```

Example A-2 Using ReInit and Control Functions

```
/*
creation of the encoder instance
*/
pInfo->params.direction = 0; /* Direction: encode */
pInfo->params.modes.vad = 0; /* Suppress silence compression */
pInfo->params.law = 0; /* Linear PCM input */
pInfo->params.modes.bitrate = pInfo->pRateTbl[pInfo->nRates-1].bitrate; /*For
example, set the highest bitrate*/
/* determine how many memory blocks the encoder needs  */
if(USC_NoError !=
USC_Gxxx_Fnxs->std.NumAlloc(&pInfo->params,&nbanksEnc)) exit(3);
/* allocate memory for the memory bank table */
pBanksEnc = (USC_MemBank*)malloc(sizeof(USC_MemBank)*nbanksEnc);

/* Determine the size of each block */
if(USC_NoError !=
USC_Gxxx_Fnxs->std.MemAlloc(&pInfo->params,pBanksEnc)) exit(4);
/* allocate memory for each block */
for(i=0; i<nbanksEnc;i++){
   pBanksEnc[i].pMem = (unsigned char*)malloc(pBanksEnc[i].nbytes);
}
/* Create an encoder instance */
if(USC_NoError != USC_Gxxx_Fnxs->std.Init(&pInfo->params,
pBanksEnc,&hUSCEncoder)) exit(5);
```

Example A-2 Using ReInit and Control Functions

```
        /*Some processing*/
        /*Now we change bitrate on the fly*/
        pInfo->params.modes.bitrate = pInfo->pRateTbl[0].bitrate; /* For example, set
        the lowest bitrate*/
        if(USC_NoError != USC_Gxxx_Fnxs->std.Control(&pInfo->params.modes,
        hUSCEncoder)) exit(6);
        /*Some processing with the new bitrate*/
        /*Now we re-initialize the encoder instance*/
        if(USC_NoError !=
        USC_Gxxx_Fnxs->std.Reinit(&pInfo->params.modes,hUSCEncoder)) exit(7);
        /*
        Release memory of the encoder instance
        */
        for(i=0; i<nbanksEnc;i++){
           free(pBanksEnc[i].pMem);
        }
        free(pBanksEnc);
        free(pInfo);
     }
```

Bibliography

This bibliography provides a list of reference books and other sources of additional information that might be helpful to the application programmer using the Intel® Integrated Performance Primitives Unified Speech Component interface.

[RFC2833] H. Schulzrinne, S. Petrack. *RTP Payload for DTMF Digits, Telephony Tones and Telephony Signals*. May 2000.
Available from http://www.ietf.org/rfc/rfc2833.txt?number=2833.

Index