KORT 2.0.0

by zplane.development

January 20, 2021

Contents

1 Kort Documentation

1.1 Introduction

Kort is zplane's chord recognition SDK. It analyzes the harmonic content of a music signal and outputs a sequence of chords with corresponding time stamps. Kort assumes that the input signal contains music in Western tonality with 12 pitches per octave (C, C#, D,..., B) and in equal temperament. It furthermore assumes that the music contains simultaneously sounding notes that form musical chords consisting of three to four different pitches.
Each chord in the output chord sequence is specified by a root pitch (e.g. C, C#, D, ...) and a chord type (e.g. major, minor, diminished, 7, ...). The output also provides information about the pitch classes contained in a chord (e.g. "C, E and G" in a C major chord). For each chord in the resulting chord sequence, the user can retrieve a list of alternative chords together with corresponding probabilities.
Kort can be used in three different modes: A basic mode that outputs only major and minor chords, and an extended mode that outputs a larger set of chord types containing between three and four pitches, and a custom mode in which users can define their own set of chords.
Chords can be aligned with beat times. Zplane's [aufTAKT] beat tracking SDK seamlessly integrates with Kort, but the algorithm can also be provided with beat information from a different source such as a fixed beat grid. Likewise, the tuning frequency of the input signal can be specified manually if this information is available.
Kort is an offline process which means it requires the complete audio file as input and outputs the chord sequence result only after the file has been processed in its entirety. It is *not* an online or real-time chord recognition system. In extended mode the algorithm is able to process audio files approx. 60 times faster than realtime on a 64-bit 2.3 GHz Dual-Core processor. The mimimum sample rate is 16 kHz. There is no restriction on the number of input channels.

The project contains the required libraries for the operating system Kort was licensed for with the appropriate header files. An example application illustrates the functionality of this SDK.

This document is structured as follows: The first part contains the API documentation of the Kort SDK. The API documentation contains naming conventions and function descriptions of the C++-API. In the second part, a detailed explanation of the example application is provided.

1.1.1 Naming Conventions

The following naming conventions are used throughout this manual: A **frame** denotes the number of audio samples per channel, i.e. 512 stereo frames correspond to 1024 float values (samples).
A **pitch class** describes the name of a pitch independent of the octave it occurred in. In other words, notes with pitches at *C0, C1, C2, etc.* all belong to pitch class *C*.
The **root pitch** refers to the pitch class of the root of the chord, i.e. the root pitch of a

C major chord is pitch class *C*.
A **chord type** is defined as a set of intervals above a root. The chord type *major*, for example, consists of the intervals: *unison, major third, and fifth*.
Chords consists of a *root pitch* and a chord type, e.g. *C major*.
A **chord dictionary** or simply **dictionary** is a set of chord types. The dictionary KortIf::majorMinor for example consists of the chord types *major* and *minor*.

1.2 API Documentation

The analysis consists of two stages: a pre-processing stage in which the audio is analyzed, and a processing stage that carries out the actual chord estimation.
The pre-processing stage is based on the push principle: successive blocks of input audio frames are pushed into the KortIf::PreProcess() function. It finishes by calling KortIf::FinishPreProcess() after the audio file has been entirely pushed into KortIf::PreProcess(). The KortIf::Process() function is called subsequently and results can be obtained by calling KortIf::GetResult().

1.2.1 Memory Allocation

The Kort SDK does not allocate any buffers handled by the calling application. The input buffer as well as the result objects have to be allocated/created by the calling application.

1.2.2 Instance Control Functions

- **static ErrorType KortIf::CreateInstance (KortIf*& pCKortIf, float fSample↩ Rate, int iFileSizeInFrames, int iNumOfChannels)**
  Creates a new instance of Kort. The handle to the new instance is returned in parameter pCKortIf. fSampleRate and iNumChannels denote the input samplerate and number of channels, respectively. The expected length of the input file in frames is provided by the argument iFileSizeInFrames. This values is used to initialize internal buffers required for the chord estimation. It is possible to push more frames into KortIf::PreProcess() than initially specified, but this will cause further memory allocations during KortIf::PreProcess().
  The function returns an error code (see KortIf::ErrorType). A call to this function is mandatory.

- **static ErrorType DestroyInstance (KortIf*& pCKortIf)**
  Destroys the Kort instance provided by pCKortIf. The function returns an error code (see KortIf::ErrorType). A call to this function is mandatory.

1.2.3 Parameter Setting Functions

- **ErrorType KortIf::SetChordDictionaryType (ChordDictionaryType eChord↩ Dict)**
  Sets the chord dictionary Kort will use (see KortIf::ChordDictionaryType). This

function is optional but it needs to be called before KortIf::Process(). The default dictionary is KortIf::ChordDictionaryType::extended.

- **ErrorType SetCustomChordDictionary (const KortDictionaryIf* const p↩ Dictionary)**
  Sets a custom chord dictionary. To initialize the dictionary please refer to the interface KortDictionaryIf.

- **ErrorType KortIf::SetTuningFrequency (float fTuningFrequency)**
  Sets the tuning frequency of the input audio. When called before KortIf::PreProcess(), this will bypass the internal tuning frequency estimation which will lead to overall shorter processing times.

- **ErrorType KortIf::SetBassWeight (float fBassWeight)**
  Sets the weight of the estimated bass pitches against the estimate of the remaining chord notes. The value must be in the range between 0 and 1. A value of 0 means that the chord estimation is done without bass estimation, a value of 1 means that only the bass (and no other pitches) is taken into account for the chord estimation. The default value is 0.35.

1.2.4 Processing Functions

- **ErrorType KortIf::PreProcess (float const *const *const ppfInputBuffer, int iNumberOfFrames)**
  Pre-processing function. ppfInputBuffer is an array of pointers to the audio data. ppfInputBuffer[0] is a pointer to the data of the first channel, ppfInput↩ Buffer[1] points to the data of the second channel etc. iNumberOfFrames specifies the number of frames, i.e. the number of samples in each channel. This function can repeatedly be called with successive chunks of audio until the entire signal has been pushed into Kort. This function will return an error if it is called after KortIf::FinishPreProcess() has been called.

- ***ErrorType KortIf::FinishPreProcess()***
  This function has to be called after all audio frames have been pushed into Kort by KortIf::PreProcess(). It can (and needs to) be called only once and will return an error if called before KortIf::PreProcess() or after KortIf::Process().

- ***ErrorType KortIf::Process(const CaufTAKTResultIf const pBeatResult = nullptr)***
  This function does the actual chord estimation. It can be called without any input arguments, in which case the algorithm will figure out the chord boundaries by itself. If a CaufTAKTResultIf beat info object is provided, Kort will use this information to match the chord boundaries with the provided beat times. This function has to be called after KortIf::FinishPreProcess().

1.2.5 Result Retrieving Function

- ***ErrorType KortIf::GetResult (KortResultIf *pCResult)***
  Returns the chord sequence as a KortResultIf object. The KortResultIf object needs to be instantiated first before it is passed to this function. This function can only be called after KortIf::Process() has been called.

- *ErrorType KortIf::GetTuningFrequency (float& fTuningFrequency)*
  Returns the estimated tuning frequency. This function can only be called after the preprocessing stage.

1.3 Command Line Usage Example

The command line example can be executed by the following command

```
KortCl -i <inputFile> -r <chordResultFile>
```

The complete code can be found in the example source file KortClMain.cpp. In the first step, we declare a KortIf pointer and a KortResultIf pointer and create an instance of the KortIf class:

```
KortIf*            pInstanceHandle = 0;
KortResultIf*      pResult         = 0;
eError = KortIf::CreateInstance (pInstanceHandle,
                                 inputFile.GetSampleRate(),
                                 inputFile.GetFileSize(),
                                 inputFile.GetNumOfChannels());
```

We can select a chord dictionary as follows:

```
pInstanceHandle->SetChordDictionaryType (KortIf::extended);
```

We can set the tuning frequency of the input file in case this information is available. It is advisable to do this immediately after instance creation.

```
// set tuning frequency (optional)
//pInstanceHandle->SetTuningFrequency (fTuningFreq);
```

We then read chunks of data from our input file,

```
    while (bReadNextFrame) {
        iNumFramesRead = inputFile.Read(ppfInput, kBlockSize);
        if (iNumFramesRead < kBlockSize) {
            for (int ch = 0; ch < inputFile.GetNumOfChannels(); ch++)
                memset(&ppfInput[ch][iNumFramesRead], 0, (kBlockSize -
  iNumFramesRead) * sizeof(float));
            bReadNextFrame = false;
        }
```

And push each chunk into our PreProcess() function.

```
// now we can start the preprocessing!
cout << "****************************************" << endl;
cout << "Kort Pre-Processing!" << endl;
while (bReadNextFrame)
{
    // read audio data
    int iNumFramesRead (inputFile.Read (ppfInput, kBlockSize));

    if(iNumFramesRead < kBlockSize)
    {
        for (int ch = 0; ch < inputFile.GetNumOfChannels(); ch++)
            memset(&ppfInput[ch][iNumFramesRead], 0,
  (kBlockSize-iNumFramesRead)*sizeof(float));
        bReadNextFrame = false;
    }
    if (bVerbose)
        CLShowProcessTime(iCurrentFrame++, inputFile.GetSampleRate(),
  kBlockSize);

    // preprocessing
    clStartTime = clock();
    eError = pInstanceHandle->PreProcess (ppfInput, iNumFramesRead);
    }
```

After the entire file has been read and pushed into Kort, we call FinishPre↩ Process() once to terminate the preprocessing stage

```
eError = pInstanceHandle->FinishPreProcess();
```

The bass weight can be set by calling SetBassWeight()

```
//     set bass weight (optional)
//    pInstanceHandle->SetBassWeight (0.73f);
```

We then call Kort's process function:

```
eError = pInstanceHandle->Process ();
```

If beat information is available we could provide Kort with a CaufTAKTResultIf∗

```
//eError = pInstanceHandle->Process (pCBeatResult); // beat results can
   optionally be provided
```

To obtain the resulting chord sequence, we create an instance of KortResultIf,

```
// create results instance
if (KortResultIf::CreateInstance (pResult) == false)
```

and call Kort's GetResult() function

```
// get chord results
bool bChordSmoothing = false;
pInstanceHandle->GetResult (pResult, bChordSmoothing);
```

We can access the individual chords of the resulting chord sequence (e.g. in order to print them on the command line) as well as a list of alternative chords:

```
cout << "Printing Results!" << endl;
float fTuningFreq;
pInstanceHandle->GetTuningFrequency (fTuningFreq);
cout << "Tuning frequency: " << fTuningFreq << endl;
for (int i = 0; i < pResult->GetNumEntries (); i++)
{
    const KortResultIf::ChordSequenceElement element = pResult->GetEntry
(i);
    cout << element.GetChord().GetName() << "  ";
    // get alternative chords for the current entry
    //std::vector<KortResultIf::ChordSequenceElement> alternativeChords =
pResult->GetAlternativeChords (i);
}
cout << endl;
```

Finally we destroy the KortIf instance and eventually the KortResultIf instance

```
KortResultIf::DestroyInstance (pCResult);
KortIf::DestroyInstance (pInstanceHandle);
```

The above code snippets demonstrated the basic functionality of the Kort library.

1.4 Support

Support for the source code is - within the limits of the agreement - available from:

zplane.development GmbH & Co KG
grunewaldstr. 83
d-10823 berlin
germany

fon: +49.30.854 09 15.0
fax: +49.30.854 09 15.5

@: info@zplane.de

2 Namespace Index

2.1 Namespace List

Here is a list of all namespaces with brief descriptions:

3 Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

4 File Index

4.1 File List

Here is a list of all files with brief descriptions:

5 Namespace Documentation

5.1 KortGlobals Namespace Reference

Global defines and class interfaces.

Classes

- class Chord

  Class representing a musical chord.
- class Key

  Class representing a musical key.

Enumerations

- enum PitchClass {
  noPitch = -1, C, CSharp, D,
  DSharp, E, F, FSharp,
  G, GSharp, A, ASharp,
  B }
- enum ChordType {
  noChord, major, minor, diminished,
  augmented, majorSeventh, minorSeventh, seventh,
  diminishedSeventh, halfDiminishedSeventh, majorSixth, minorSixth,
  numChordTypes }
- enum KeyType { majorKey, minorKey }

Variables

- static const int numPitchClasses = 12

5.1.1 Enumeration Type Documentation

PitchClass `enum KortGlobals::PitchClass`

Pitch classes

Enumerator

noPitch	
C	
CSharp	
D	
DSharp	
E	
F	
FSharp	
G	
GSharp	
A	
ASharp	
B	

ChordType `enum KortGlobals::ChordType`

Chord types

Enumerator

noChord	
major	
minor	
diminished	
augmented	
majorSeventh	
minorSeventh	
seventh	
diminishedSeventh	
halfDiminishedSeventh	
majorSixth	
minorSixth	
numChordTypes	

KeyType `enum KortGlobals::KeyType`

Key types

Enumerator

majorKey	
minorKey	

5.1.2 Variable Documentation

numPitchClasses `const int KortGlobals::numPitchClasses = 12 [static]`

6 Class Documentation

6.1 KortGlobals::Chord Class Reference

Class representing a musical chord.

`#include <KortGlobalsIf.h>`

Collaboration diagram for KortGlobals::Chord:

Public Member Functions

- Chord ()
- Chord (PitchClass rootPitch, ChordType chordType)
- Chord (const Chord &other)
- Chord (Impl *pImpl)
- ~Chord ()
- Chord & operator= (const Chord &rhs)
- const bool operator== (const Chord &rhs) const
- const bool operator!= (const Chord &rhs) const
- PitchClass GetRootPitch () const
- ChordType GetChordType () const
- std::vector< PitchClass > GetPitches () const
- std::string GetName () const

6.1.1 Constructor & Destructor Documentation

Chord() [1/4] `KortGlobals::Chord::Chord ( )`

Constructs an empty chord object.

Chord() [2/4]
```
KortGlobals::Chord::Chord (
        PitchClass rootPitch,
        ChordType chordType )
```

Constructs a chord with a given rootPitch and chordType.

Chord() [3/4]
```
KortGlobals::Chord::Chord (
        const Chord & other )
```

Constructs a copy of the provided chord.

Chord() [4/4]
```
KortGlobals::Chord::Chord (
        Impl * pImpl )
```

Constructs a chord given an existing implementation. This constructor is only for internal use.

~Chord() `KortGlobals::Chord::~Chord ( )`

Destroys a chord object.

6.1.2 Member Function Documentation

operator=()
```
Chord& KortGlobals::Chord::operator= (
        const Chord & rhs )
```

Assignment operator.

operator==() `const bool KortGlobals::Chord::operator== ( const Chord & rhs ) const`

Equality operator.

operator"!=() `const bool KortGlobals::Chord::operator!= ( const Chord & rhs ) const`

Inequality operator.

GetRootPitch() `PitchClass KortGlobals::Chord::GetRootPitch ( ) const`

Returns the pitch class of the root of the chord.

GetChordType() `ChordType KortGlobals::Chord::GetChordType ( ) const`

Returns the chord type of the chord.

GetPitches() `std::vector<PitchClass> KortGlobals::Chord::GetPitches ( ) const`

Returns the pitch classes of the chord.

GetName() `std::string KortGlobals::Chord::GetName ( ) const`

Returns the name string of the chord in the format <rootPitch>:<chordType>.

The documentation for this class was generated from the following file:

- KortGlobalsIf.h

6.2 KortResultIf::ChordSequenceElement Class Reference

Class representing an element of a resulting chord sequence.

`#include <KortResultIf.h>`

Collaboration diagram for KortResultIf::ChordSequenceElement:

Public Member Functions

- ChordSequenceElement (const ChordSequenceElement &)
- ChordSequenceElement (Impl *pImpl)
- ~ChordSequenceElement ()
- ChordSequenceElement & operator= (const ChordSequenceElement &rhs)
- float GetStartTimeInS () const
- const KortGlobals::Chord GetChord () const
- float GetProbability () const
- bool IsValid () const

6.2.1 Constructor & Destructor Documentation

ChordSequenceElement() [1/2] `KortResultIf::ChordSequenceElement::ChordSequenceElement ( const ChordSequenceElement & )`

Copy constructor. For internal use only.

ChordSequenceElement() [2/2] `KortResultIf::ChordSequenceElement::ChordSequenceElement ( Impl * pImpl )`

Constructor for a given implementation. For internal use only.

~ChordSequenceElement() `KortResultIf::ChordSequenceElement::~ChordSequenceElement ( )`

Destructor.

6.2.2 Member Function Documentation

operator=() `ChordSequenceElement& KortResultIf::ChordSequenceElement::operator= ( const ChordSequenceElement & rhs )`

Assignment operator.

GetStartTimeInS() `float KortResultIf::ChordSequenceElement::GetStartTimeInS ( ) const`

Returns the start time of the chord in seconds.

GetChord() `const KortGlobals::Chord KortResultIf::ChordSequenceElement::GetChord ( ) const`

Returns the chord of this sequence element.

GetProbability() `float KortResultIf::ChordSequenceElement::GetProbability ( ) const`

Returns the probability of this sequence element.

IsValid() `bool KortResultIf::ChordSequenceElement::IsValid ( ) const`

Checks if this sequence element is valid.

The documentation for this class was generated from the following file:

- KortResultIf.h

6.3 KortGlobals::Key Class Reference

Class representing a musical key.

`#include <KortGlobalsIf.h>`

Collaboration diagram for KortGlobals::Key:

Public Member Functions

- Key ()
- Key (PitchClass rootPitch, KeyType keyType)
- Key (const Key &other)
- Key (Impl *pImpl)
- ~Key ()

- Key & operator= (const Key &rhs)
- const bool operator== (const Key &rhs) const
- const bool operator!= (const Key &rhs) const
- PitchClass GetRootPitch () const
- KeyType GetKeyType () const
- std::string GetName () const

6.3.1 Constructor & Destructor Documentation

Key() [1/4] `KortGlobals::Key::Key ( )`

Constructs an empty key object.

Key() [2/4]
```
KortGlobals::Key::Key (
        PitchClass rootPitch,
        KeyType keyType )
```
Constructs a key with a given rootPitch and keyType.

Key() [3/4]
```
KortGlobals::Key::Key (
        const Key & other )
```
Constructs a copy of the provided key.

Key() [4/4]
```
KortGlobals::Key::Key (
        Impl * pImpl )
```
Constructs a key given an existing implementation. This constructor is only for internal use.

~Key() `KortGlobals::Key::~Key ( )`

Destroys a key object.

6.3.2 Member Function Documentation

operator=()
```
Key& KortGlobals::Key::operator= (
        const Key & rhs )
```
Assignment operator.

operator==()
```
const bool KortGlobals::Key::operator== (
        const Key & rhs ) const
```
Equality operator.

operator"!=()
```
const bool KortGlobals::Key::operator!= (
        const Key & rhs ) const
```
Inequality operator.

GetRootPitch() `PitchClass KortGlobals::Key::GetRootPitch ( ) const`
Returns the pitch class of the root of the key.

GetKeyType() `KeyType KortGlobals::Key::GetKeyType ( ) const`
Returns the key type of the key.

GetName() `std::string KortGlobals::Key::GetName ( ) const`
Returns the pitch classes of the key. Returns the name string of the key in the format <rootPitch>:<keyType>.
The documentation for this class was generated from the following file:

- KortGlobalsIf.h

6.4 KortDictionaryIf Class Reference

Interface class for a chord dictionary.
`#include <KortDictionaryIf.h>`
Collaboration diagram for KortDictionaryIf:

Public Types

- enum ErrorType {
  noError, memError, likelihoodOutOfRangeError, invalidChordError,
  chordNotInDictionaryError, numErrors }

Public Member Functions

- virtual ErrorType AddChord (const KortGlobals::Chord &chord, float fLikelihood)=0
- virtual int GetNumChords () const =0
- virtual const KortGlobals::Chord GetChord (int iIndex) const =0
- virtual float GetLikelihood (int iIndex) const =0

Static Public Member Functions

- static ErrorType CreateInstance (KortDictionaryIf *&pKortDictionary)
- static void DestroyInstance (KortDictionaryIf *&pKortDictionary)

Protected Member Functions

- virtual ~KortDictionaryIf ()

6.4.1 Member Enumeration Documentation

Enumerator

ErrorType `enum KortDictionaryIf::ErrorType`

Error codes

Enumerator

noError	no error occurred
memError	memory allocation failed
likelihoodOutOfRangeError	likelihood argument is invalid
invalidChordError	combination of rootPitch and chordType is invalid
chordNotInDictionaryError	chord is not part of the dicionary
numErrors	

6.4.2 Constructor & Destructor Documentation

~KortDictionaryIf() `virtual KortDictionaryIf::~KortDictionaryIf ( ) [inline], [protected], [virtual]`

6.4.3 Member Function Documentation

CreateInstance() `static ErrorType KortDictionaryIf::CreateInstance ( KortDictionaryIf *& pKortDictionary ) [static]`

Creates an instance of the Kort dictionary class.

Parameters

pKortDictionary	reference to Kort dictionary instance pointer

DestroyInstance() `static void KortDictionaryIf::DestroyInstance ( KortDictionaryIf *& pKortDictionary ) [static]`

Destroys an instance of the Kort dictionary class.

Parameters

pKortDictionary	reference to Kort dictionary instance pointer

AddChord() `virtual ErrorType KortDictionaryIf::AddChord ( const KortGlobals::Chord & chord, float fLikelihood ) [pure virtual]`

Adds a single chord to the chord dictionary.

Parameters

chord	Chord to add
fLikelihood	Likelihood of the chord given in the range [0, ..., 1]

GetNumChords() `virtual int KortDictionaryIf::GetNumChords ( ) const [pure virtual]`

Returns the number of chords in the dictionary

GetChord() `virtual const KortGlobals::Chord KortDictionaryIf::GetChord ( int iIndex ) const [pure virtual]`

Returns the chord from the dictionary at a specific sequence index.

Parameters

iIndex	Index of the chord in the dictionary. It must be in the range [0, ..., GetNumChords()-1]

GetLikelihood() `virtual float KortDictionaryIf::GetLikelihood ( int iIndex ) const [pure virtual]`

Returns the likelihood of a chord from the dictionary at a specific index.

Parameters

iIndex	Index of the chord in the dictionary. It must be in the range [0, ..., GetNumChords()-1]

The documentation for this class was generated from the following file:

- KortDictionaryIf.h

6.5 KortIf Class Reference

Interface class for the Kort SDK.

`#include <KortIf.h>`

Collaboration diagram for KortIf:

Public Types

- enum ErrorType { noError, memError, invalidFunctionParamError, invalidFunctionCallError, notInitializedError, notPreProcessedError, notProcessedError, invalidDataChunk, numErrors }
- enum VersionType { major, minor, patch, revision }
- enum ChordDictionaryType { majorMinor, extended, custom }
- enum DataChunkCheckpoint { afterPreProcessing, afterTuningCorrection }

Public Member Functions

- virtual ErrorType SetTuningFrequency (float fTuningFrequency)=0
- virtual ErrorType SetKey (KortGlobals::Key key, float fStrictness=0.038)=0
- virtual ErrorType SetBassWeight (float fBassWeight)=0
- virtual ErrorType SetChordDictionaryType (ChordDictionaryType eChordDict)=0
- virtual ChordDictionaryType GetChordDictionaryType () const =0
- virtual ErrorType SetCustomChordDictionary (const KortDictionaryIf *const p↩ Dictionary)=0
- virtual ErrorType PreProcess (float const *const *const ppfInputBuffer, int i↩ NumberOfFrames)=0
- virtual ErrorType FinishPreProcess ()=0
- virtual ErrorType Process (const CaufTAKTResultIf *const pBeatResult=0, bool bHighChordChangeSensitivity=false)=0
- virtual ErrorType GetResult (KortResultIf *pCResult, bool bChordSmoothing=false) const =0
- virtual ErrorType GetTuningFrequency (float &fTuningFrequency)=0
- virtual ErrorType GetDataChunk (void *pPreAllocatedDataChunk, DataChunkCheckpoint checkpoint=afterTuningCorrection)=0
- virtual size_t GetDataChunkSizeInBytes (DataChunkCheckpoint checkpoint=afterTuningCorrection)=0
- virtual ErrorType SetDataChunk (void *pDataChunk, int iDataChunkSizeIn↩ Bytes)=0

Static Public Member Functions

- static const int GetVersion (const VersionType eVersionIdx)
- static const char * GetBuildDate ()
- static ErrorType CreateInstance (KortIf *&pKortIf, float fSampleRate, int iFile↩ SizeInFrames, int iNumOfChannels)
- static void DestroyInstance (KortIf *&pKortIf)

Protected Member Functions

- virtual ~KortIf ()

6.5.1 Member Enumeration Documentation

ErrorType `enum KortIf::ErrorType`

Error codes

Enumerator

noError	no error occurred
memError	memory allocation failed
invalidFunctionParamError	one or more function parameters are not valid
invalidFunctionCallError	function call not allowed at this stage
notInitializedError	instance has not been initialized yet
notPreProcessedError	instance has not been preprocessed yet
notProcessedError	instance has not been processed yet
invalidDataChunk	data chunk was invalid
numErrors	

VersionType `enum KortIf::VersionType`
Version number

Enumerator

major	major version number
minor	minor version number
patch	patch version number
revision	revision number

ChordDictionaryType `enum KortIf::ChordDictionaryType`
Chord dictionary

Enumerator

majorMinor	only major and minor chords
extended	dictionary containing triads and tetrads
custom	custom dictionary, return type only when custom dictionary is set

DataChunkCheckpoint `enum KortIf::DataChunkCheckpoint`
Chord data chunk checkpoints

Enumerator

afterPreProcessing	save preprocessed audio and tuning frequency
afterTuningCorrection	save computed pre-estimation, not possible to set tuning frequency

6.5.2 Constructor & Destructor Documentation

~KortIf() `virtual KortIf::~KortIf ( ) [inline], [protected], [virtual]`

6.5.3 Member Function Documentation

GetVersion() `static const int KortIf::GetVersion (`
`const VersionType eVersionIdx ) [static]`

Returns major version, minor version, patch and build number of this Kort version.

Parameters

eVersionIdx	requested version number part

Returns

version number part

GetBuildDate() `static const char* KortIf::GetBuildDate ( ) [static]`

Returns the build date string.

CreateInstance() `static ErrorType KortIf::CreateInstance (`
`KortIf *& pKortIf,`
`float fSampleRate,`
`int iFileSizeInFrames,`
`int iNumOfChannels ) [static]`

Creates an instance of Kort.

Parameters

pKortIf	reference to Kort instance pointer
fSampleRate	sample rate of input audio
iFileSizeInFrames	length of input audio in frames
iNumOfChannels	number of input audio channels

Returns

an error code, or 0 if no error occurred

DestroyInstance() `static void KortIf::DestroyInstance (`
`KortIf *& pKortIf ) [static]`

Destroys an instance of Kort.

Parameters

p↩ KortIf	reference to Kort instance pointer

SetTuningFrequency() `virtual ErrorType KortIf::SetTuningFrequency ( float fTuningFrequency ) [pure virtual]`

Sets the tuning frequency.

This function is optional. When called before the preprocessing stage it bypasses Kort's internal tuning frequency estimation which can speed up the computation.

SetKey() `virtual ErrorType KortIf::SetKey ( KortGlobals::Key key, float fStrictness = 0.038 ) [pure virtual]`

Sets the key.

Whether to prefer chords that are in detected key. This function is optional. The value "fStrictness" must be in range between 0 and 1.

Parameters

key	detected key of song
fStrictness	determines how much preference is given to chords that belong to the same key.

SetBassWeight() `virtual ErrorType KortIf::SetBassWeight ( float fBassWeight ) [pure virtual]`

Sets the bass weight.

The bass weight controls the influence of the estimated bass pitches against the estimate of the remaining chord notes. The value must be in the range between 0 and 1. A value of 0 means that the chord estimation is done without bass estimation, a value of 1 means that only the bass (and no other pitches) is taken into account for the chord estimation.

This function is optional. The default value for the bass weight is 0.35.

SetChordDictionaryType() `virtual ErrorType KortIf::SetChordDictionaryType ( ChordDictionaryType eChordDict ) [pure virtual]`

Set the chord dictionary type.

The chord dictionary determines the types of chords that Kort can detect. The default value is KortIf::ChordDictionaryType::extended. KortIf::ChordDictionaryType↩ ::custom cannot be set; a custom dictionary has to be set by KortIf::SetCustomChordDictionary. The chord dictionary type needs to be set before calling KortIf::Process().

GetChordDictionaryType() `virtual ChordDictionaryType KortIf::GetChordDictionary↩ Type ( ) const [pure virtual]`

Returns the current chord dictionary type.

SetCustomChordDictionary() `virtual ErrorType KortIf::SetCustomChordDictionary ( const KortDictionaryIf *const pDictionary ) [pure virtual]`

Sets a custom chord dictionary.

The dictionary needs to be set before calling KortIf::Process().

PreProcess()
```
virtual ErrorType KortIf::PreProcess (
float const *const *const ppfInputBuffer,
int iNumberOfFrames ) [pure virtual]
```

Preprocesses a block of audio.

This function can be called multiple times in order to provide successive chunks of the input audio signal.

Parameters

ppfInputBuffer	pointer to the input data chunk. ppfInputBuffer[i] points to the i-th audio channel.
iNumberOfFrames	The number of audio samples in each audio channel of the provided input data chunk.

FinishPreProcess() `virtual ErrorType KortIf::FinishPreProcess ( ) [pure virtual]`

Terminates the preprocessing stage.

This function should only be called *once* after the last input frames have been provided by the PreProcess function. A call to this function is required before proceeding to the Process() function.

Process()
```
virtual ErrorType KortIf::Process (
const CaufTAKTResultIf *const pBeatResult = 0,
bool bHighChordChangeSensitivity = false ) [pure virtual]
```

Performs the chord estimation.

Performs the chord estimation. This function can only be called when the preprocessing stage has finished or a previously analyzed data chunk has been set.

Parameters

pBeatResult	pointer to beat information (optional). default is a null pointer, which means that no beat information is available.
bHighChordChangeSensitivity	increases sensitivity for chord changes (optional).

GetResult() `virtual ErrorType KortIf::GetResult (`
`KortResultIf * pCResult,`
`bool bChordSmoothing = false ) const [pure virtual]`

Returns resulting chord sequence.

Parameters

bChordSmoothing	whether to combine adjacent chords that have the same basic triad (optional).

GetTuningFrequency() `virtual ErrorType KortIf::GetTuningFrequency (`
`float & fTuningFrequency ) [pure virtual]`

Returns the tuning frequency.

GetDataChunk() `virtual ErrorType KortIf::GetDataChunk (`
`void * pPreAllocatedDataChunk,`
`DataChunkCheckpoint checkpoint = afterTuningCorrection ) [pure virtual]`

Returns internal analysis data after preprocessing in an pre-allocated memory chunk.

This can be used to save analysis data for later processing. The memory handling has to be done by the calling application. The size of pre-allocated memory can be retrieved by GetDataChunkSizeInBytes()

GetDataChunkSizeInBytes() `virtual size_t KortIf::GetDataChunkSizeInBytes (`
`DataChunkCheckpoint checkpoint = afterTuningCorrection ) [pure virtual]`

Returns the length in bytes to be pre allocated in order to properly call GetData↩ Chunk.

SetDataChunk() `virtual ErrorType KortIf::SetDataChunk (`
`void * pDataChunk,`
`int iDataChunkSizeInBytes ) [pure virtual]`

Set a saved data chunk in order to recover a previous pre-analysis state.

Parameters

pDataChunk	pointer to data chunk
iDataChunkSizeInBytes	size of the data chunk

Returns

noError if data recovery was performed without error, otherwise invalidData↩ Chunk

The documentation for this class was generated from the following file:

- KortIf.h

6.6 KortResultIf Class Reference

Interface class for the resulting chord sequence.

`#include <KortResultIf.h>`

Collaboration diagram for KortResultIf:

Classes

- class ChordSequenceElement

  Class representing an element of a resulting chord sequence.

Public Member Functions

- virtual int GetNumEntries () const =0
- virtual const ChordSequenceElement GetEntry (int iIndex) const =0
- virtual std::vector< ChordSequenceElement > GetAlternativeChords (int iEntryIndex) const =0

Static Public Member Functions

- static bool CreateInstance (KortResultIf *&pKortResultIf)
- static void DestroyInstance (KortResultIf *&pKortResultIf)

Protected Member Functions

- virtual ~KortResultIf ()

6.6.1 Constructor & Destructor Documentation

~KortResultIf() `virtual KortResultIf::~KortResultIf ( ) [inline], [protected], [virtual]`

6.6.2 Member Function Documentation

CreateInstance() `static bool KortResultIf::CreateInstance ( KortResultIf *& pKortResultIf ) [static]`

Creates an instance of the Kort result class

Parameters

pKortResultIf	Reference to Kort result instance pointer

Returns

true when the instance could be created, otherwise false

DestroyInstance() `static void KortResultIf::DestroyInstance ( KortResultIf *& pKortResultIf ) [static]`

Destroys a KortResultIf instance

Parameters

pKort↩ ResultIf	Reference to Kort result instance pointer

GetNumEntries() `virtual int KortResultIf::GetNumEntries ( ) const [pure virtual]`

Returns the number of entries in the chord sequence of the Kort result instance

GetEntry() `virtual const ChordSequenceElement KortResultIf::GetEntry ( int iIndex ) const [pure virtual]`

Returns a chord sequence element at a given sequence index

GetAlternativeChords() `virtual std::vector<ChordSequenceElement> KortResult↩ If::GetAlternativeChords ( int iEntryIndex ) const [pure virtual]`

Returns a list of alternative chords for a given chord sequence element.

Parameters

iEntryIndex	Sequence index of the target chord sequence element

Returns

Vector of ChordSequenceElement objects in decreasing order of their probability.

The documentation for this class was generated from the following file:

- KortResultIf.h

7 File Documentation

7.1 docugen.txt File Reference

7.2 KortDictionaryIf.h File Reference

interface of the KortDictionaryIf class.

```
#include "KortGlobalsIf.h"
```

Include dependency graph for KortDictionaryIf.h:

7.3 KortGlobalsIf.h File Reference

interface of the KortGlobals namespace.

```
   #include <string>
#include <vector>
```

Include dependency graph for KortGlobalsIf.h: This graph shows which files directly or indirectly include this file:

Classes

- class KortGlobals::Chord

  Class representing a musical chord.
- class KortGlobals::Key

  Class representing a musical key.

Namespaces

- KortGlobals

  Global defines and class interfaces.

Enumerations

- enum KortGlobals::PitchClass { KortGlobals::noPitch = -1, KortGlobals::C, KortGlobals::CSharp, KortGlobals::D, KortGlobals::DSharp, KortGlobals::E, KortGlobals::F, KortGlobals::FSharp, KortGlobals::G, KortGlobals::GSharp, KortGlobals::A, KortGlobals::ASharp, KortGlobals::B }
- enum KortGlobals::ChordType { KortGlobals::noChord, KortGlobals::major, KortGlobals::minor, KortGlobals::diminished, KortGlobals::augmented, KortGlobals::majorSeventh, KortGlobals::minorSeventh, KortGlobals::seventh, KortGlobals::diminishedSeventh, KortGlobals::halfDiminishedSeventh, KortGlobals::majorSixth, KortGlobals::minorSixth, KortGlobals::numChordTypes }
- enum KortGlobals::KeyType { KortGlobals::majorKey, KortGlobals::minorKey }

Variables

- static const int KortGlobals::numPitchClasses = 12

7.3.1 Detailed Description

:

7.4 KortIf.h File Reference

interface of the KortIf class.

```
   #include "KortGlobalsIf.h"
```

Include dependency graph for KortIf.h:

Classes

- class KortIf

  Interface class for the Kort SDK.

7.4.1 Detailed Description

:

7.5 KortResultIf.h File Reference

interface of the KortResultIf class.

```
#include "KortGlobalsIf.h"
```

Include dependency graph for KortResultIf.h:

Classes

- class KortResultIf

  Interface class for the resulting chord sequence.
- class KortResultIf::ChordSequenceElement

  Class representing an element of a resulting chord sequence.

7.5.1 Detailed Description

: