IUPAC International Chemical Identifier (InChI)
InChI version 1, software version 1.04 (September 2011)

API Reference

Last revision date: September 12, 2011

This document is a part of the release of the IUPAC International Chemical Identifier with InChIKey, version 1, software version 1.04.

CONTENTS

Overview ........................................................................................................................................... 3

InChI API - “classic” – general-purpose ......................................................................................... 3

  GetINCHI ......................................................................................................................................... 4

  FreeINCHI ....................................................................................................................................... 7

  GetINCHIfromINCHI ......................................................................................................................... 7

  GetStructFromINCHI ...................................................................................................................... 8

  FreeStructFromINCHI .................................................................................................................... 9

  Free_inchi_Input ............................................................................................................................ 9

  Get_inchi_Input_FromAuxInfo ..................................................................................................... 10

  CheckINCHI .................................................................................................................................. 11

InChI API - “classic” – standard InChI subset .................................................................................. 12

  GetStdINCHI .............................................................................................................................. 12

  FreeStdINCHI ............................................................................................................................ 13

  GetStructFromStdINCHI ............................................................................................................... 13

  FreeStructFromStdINCHI ........................................................................................................... 13

  Free_std_inchi_Input ..................................................................................................................... 14
<table>
<thead>
<tr>
<th>Function/Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get_std_inchi_Input_FromAuxInfo</td>
<td>14</td>
</tr>
<tr>
<td>InChI API - modularized – general-purpose</td>
<td>15</td>
</tr>
<tr>
<td>INCHIGEN_Create</td>
<td>16</td>
</tr>
<tr>
<td>INCHIGEN_Setup</td>
<td>16</td>
</tr>
<tr>
<td>INCHIGEN_DoNormalization</td>
<td>17</td>
</tr>
<tr>
<td>INCHIGEN_DoCanonicalization</td>
<td>18</td>
</tr>
<tr>
<td>INCHIGEN_DoSerialization</td>
<td>18</td>
</tr>
<tr>
<td>INCHIGEN_Reset</td>
<td>19</td>
</tr>
<tr>
<td>INCHIGEN_Destroy</td>
<td>20</td>
</tr>
<tr>
<td>InChI API - modularized – standard InChI subset</td>
<td>20</td>
</tr>
<tr>
<td>STDINCHIGEN_Create</td>
<td>20</td>
</tr>
<tr>
<td>STDINCHIGEN_Setup</td>
<td>21</td>
</tr>
<tr>
<td>STDINCHIGEN_DoNormalization</td>
<td>22</td>
</tr>
<tr>
<td>STDINCHIGEN_DoCanonicalization</td>
<td>22</td>
</tr>
<tr>
<td>STDINCHIGEN_DoSerialization</td>
<td>22</td>
</tr>
<tr>
<td>STDINCHIGEN_Reset</td>
<td>23</td>
</tr>
<tr>
<td>STDINCHIGEN_Destroy</td>
<td>23</td>
</tr>
<tr>
<td>InChIKey API – general-purpose</td>
<td>24</td>
</tr>
<tr>
<td>GetINCHIKeyFromINCHI</td>
<td>24</td>
</tr>
<tr>
<td>CheckINCHIKey</td>
<td>25</td>
</tr>
<tr>
<td>InChIKey API – standard InChI subset</td>
<td>26</td>
</tr>
<tr>
<td>GetStdINCHIKeyFromStdINCHI</td>
<td>26</td>
</tr>
<tr>
<td>InChI API – miscellaneous</td>
<td>27</td>
</tr>
<tr>
<td>GetStringLength</td>
<td>27</td>
</tr>
<tr>
<td>Examples of InChI API use</td>
<td>27</td>
</tr>
</tbody>
</table>
Overview

The current version of InChI Identifier is 1; the current version of the InChI software is 1.04 (September 2011). Previously released versions 1.01 (2006), 1.02-beta (2007), 1.02-standard (2009), and 1.03 (June 2010) as well as all earlier versions, are now considered obsolete.

By default, InChI software v. 1.04 generates standard InChI. In particular, standard identifier is generated when the software is used without any passed options. If some options are specified, and at least one of them qualifies as related to non-standard InChI, the software produces non-standard InChI/InChIKey.

However, for compatibility with the previous v. 1.02-standard (2009) release, API calls which deal only with standard InChI – for example, GetStdINCHI() - are retained (technically, they provide pre-customized interface to general-purpose API functions).

Below is a brief description of InChI/InChIKey API functions (for more details on the related data structures/parameters and see inchi_api.h header file in the InChI software source code).

InChI API - “classic” – general-purpose

The API functions for “classic” (v. 1.01-style, non-modularized) are similar to those present in InChI software v. 1.01 and v. 1.02-beta (see, however, the notes below).
GetINCHI

int INCHI_DECL GetINCHI(inchi_Input *inp, inchi_Output *out)

Description

GetINCHI() is the primary function producing InChI.

GetINCHI produces standard InChI if no InChI creation/stereo modification options are
specified. If at least one of the options SUU | SLUUD | RecMet | FixedH | Ket |
15T | SRel | SRac | SUCF is specified, generated InChI will be non-standard one.

Input

Data structure inchi_Input is created by the user.
Its layout is described in inchi_api.h header file in the InChI software source code.

Options supplied to GetINCHI in inchi_Input.szOptions should be preceded by
‘/’ under Windows or ‘-‘ Linux). Valid options are listed below.

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
<th>Default behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Structure perception (compatible with standard InChI)</td>
<td></td>
</tr>
<tr>
<td>NEWPSOFF</td>
<td>Both ends of wedge point to stereocenters</td>
<td>Only narrow end of wedge points to</td>
</tr>
<tr>
<td></td>
<td></td>
<td>stereocenter</td>
</tr>
<tr>
<td>DoNotAddH</td>
<td>All hydrogens in input structure</td>
<td>Add H according to usual</td>
</tr>
</tbody>
</table>
are explicit valences
SNOn Ignore stereo Use absolute stereo

Stereo interpretation (lead to generation of non-standard InChI)
SRel Use relative stereo Use absolute stereo
SRac Use racemic stereo Use absolute stereo
SUCF Use Chiral Flag in MOL/SD file record: if On – use Absolute stereo, Off – Relative
ChiralFlagON Set chiral flag ON -
ChiralFlagOFF Set chiral flag OFF -

InChI creation options (lead to generation of non-standard InChI)
SUU Always indicate unknown/undefined stereo Does not indicate unknown/undefined stereo unless at least one defined stereo is present
SLUUD Stereo labels for “unknown” and “undefined” are different, ‘u’ and ‘?’, resp. (new option) Stereo labels for “unknown” and “undefined” are the same (‘?’)
FixedH Include reconnected metals Do not include results
RecMet Include Fixed H layer Do not include
KET Account for keto-enol tautomerism (experimental; extension to InChI 1) Ignore keto-enol tautomerism
15T Account for 1,5-tautomerism (experimental; extension to InChI 1) Ignore 1,5-tautomerism
Miscellaneous

**AuxNone**
Omit auxiliary information

**Wnumber**
Set time-out per structure in seconds; W0 means unlimited

**OutputSDF**
Output SDfile instead of InChI

**WarnOnEmptyStructure**
Warn and produce empty InChI for empty structure

**SaveOpt**
Save custom InChI creation options (non-standard InChI)

**Output**

Data structure `inchi_Output` is described in `inchi_api.h` header file.

`inchi_Output` does not need to be initialized out to zeroes; see `FreeNCHI()/FreeSTDINCHI()` on how to deallocate it. Strings in `inchi_Output` are allocated and deallocated by InChI.

**Return codes**

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>inchi_Ret_OKAY</code></td>
<td>0</td>
<td>Success; no errors or warnings</td>
</tr>
<tr>
<td><code>inchi_Ret_WARNING</code></td>
<td>1</td>
<td>Success; warning(s) issued</td>
</tr>
<tr>
<td><code>inchi_Ret_ERROR</code></td>
<td>2</td>
<td>Error: no InChI has been created</td>
</tr>
<tr>
<td><code>inchi_Ret_FATAL</code></td>
<td>3</td>
<td>Severe error: no InChI has been created (typically, memory allocation failure)</td>
</tr>
<tr>
<td><code>inchi_Ret_UNKNOWN</code></td>
<td>4</td>
<td>Unknown program error</td>
</tr>
<tr>
<td><code>inchi_Ret_BUSY</code></td>
<td>5</td>
<td>Previous call to InChI has not returned yet</td>
</tr>
<tr>
<td><code>inchi_Ret_EOF</code></td>
<td>-1</td>
<td>no structural data has been provided</td>
</tr>
<tr>
<td><code>inchi_Ret_SKIP</code></td>
<td>-2</td>
<td>not used in InChI library</td>
</tr>
</tbody>
</table>
FreeINCHI

void INCHI_DECL FreeINCHI(inchi_Output *out)

Description

This function should be called to deallocate char* pointers obtained from each GetINCHI call.

GetINCHIfromINCHI

int INCHI_DECL GetINCHIfromINCHI(inchi_InputINCHI *inpInChI,
                           inchi_Output *out)

Description

GetINCHIfromINCHI does same as -InChI2InChI option: converts InChI into InChI for validation purposes. It may also be used to filter out specific layers. For instance, SNon would remove stereochemical layer. Omitting FixedH and/or RecMet would remove Fixed-H or Reconnected layers. Option InChI2InChI is not needed.

Notes: options are supplied in inpInChI,szOptions should be preceded by ‘/’ under Windows or ‘-‘ under Linux; there is no explicit tool to conversion from/to standard InChI

Input

inchi_InputINCHI is created by the user.

Output

Strings in inchi_Output are allocated and deallocated by InChI. inchi_Output does not need to be initalized out to zeroes; see FreeINCHI() on how to deallocate it.
**Return codes**

Same as for GetINCHI.

**GetStructFromINCHI**

```c
int INCHI DECL GetStructFromINCHI(inchi_InputINCHI *inpInChI,
inchi_OutputStruct *outStruct)
```

**Description**

This function creates structure from InChI string.

Option Inchi2Struct is not needed for GetStructFromINCHI.

**Input**

Data structure inchi_Inputinchi_InputINCHI is created by the user.

For the description, see header file inchi_api.h.

**Output**

For the description of inchi_OutputStruct, see header file inchi_api.h. Pointers in inchi_OutputStruct are allocated and deallocated by InChI. inchi_OutputStruct does not need to be initialized out to zeroes; see FreeStructFromINCHI() on how to deallocate it.

**Return codes**

The same as for GetINCHI.
**FreeStructFromINCHI**

void INCHI_DECL FreeStructFromINCHI( inchi_OutputStruct *out )

*Description*

Should be called to deallocate pointers obtained from each GetStructFromINCHI.

**Free_inchi_Input**

void INCHI_DECL Free_inchi_Input( inchi_Input *pInp )

*Description*

To deallocate and write zeroes into the changed members of pInchiInp->pInp call
Free_inchi_Input( inchi_Input *pInp ).
Get_inchi_Input_FromAuxInfo

int INCHI_DECL Get_inchi_Input_FromAuxInfo(
char *szInchiAuxInfo, int bDoNotAddH, int bDiffUnkUndfStereo, InchiInpData *pInchiInp)

Description

This function creates input data structure for InChI out of auxiliary information string. Note the parameter bDiffUnkUndfStereo (if not 0, use different labels for unknown and undefined stereo) appeared in the software v. 1.03.

Input

szInchiAuxInfo
contains ASCII Z string of InChI output for a single structure or only the AuxInfo line
bDoNotAddH
if 0 then InChI will be allowed to add implicit H
bDiffUnkUndfStereo
if not 0, use different labels for unknown and undefined stereo
pInchiInp
should have a valid pointer pInchiInp->pInp to an empty (all members = 0) inchi_Input structure

Output

The following members of pInp may be filled during the call: atom, num_atoms, stereo0D, num_stereo0D

Return codes

Same as for GetINCHI.
**CheckINCHI**

int INCHI_DECL CheckINCHI(const char *szINCHI, const int strict)

**Description**

Check if the string represents valid InChI/standard InChI.

**Input**

Input:
- szINCHI: source InChI
- strict: if 0, just briefly check for proper layout (prefix, version, etc.). The result may not be strict.
  If not 0, try to perform InChI2InChI conversion and returns success if a resulting InChI string exactly match source. Be cautious: the result may be too strict, i.e. the 'false alarm', due to imperfectness of conversion.

**Return codes**

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>INCHI_VALID_STANDARD</td>
<td>0</td>
<td>InChI is valid and standard</td>
</tr>
<tr>
<td>INCHI_VALID_NON_STANDARD</td>
<td>-1</td>
<td>InChI is valid and non-standard</td>
</tr>
<tr>
<td>INCHI_INVALID_PREFIX</td>
<td>1</td>
<td>InChI has invalid prefix</td>
</tr>
<tr>
<td>INCHI_INVALID_VERSION</td>
<td>2</td>
<td>InChI has invalid version number (not equal to 1)</td>
</tr>
<tr>
<td>INCHI_INVALID_LAYOUT</td>
<td>3</td>
<td>InChI has invalid layout</td>
</tr>
<tr>
<td>INCHI_FAIL_I2I</td>
<td>4</td>
<td>Checking InChI thru InChI2InChI is either failed or produced the result which does not match source InChI string</td>
</tr>
</tbody>
</table>
InChI API - “classic” – standard InChI subset

Described below are “standard” counterparts of general-purpose functions; these “standard” API calls are retained for compatibility reasons.

**GetStdINCHI**

```c
int INCHI_DECL GetStdINCHI(inchi_Input *inp, inchi_Output *out)
```

*Description*

This is a “standard” counterpart of GetINCHI() which may produce only the standard InChI.

*Input*

The same as for GetINCHI except that perception/creation options supplied in inchi_Input.szOptions may be only:

NEWPSOFF  DoNotAddH  SNon

Other possible options are:

AuxNone  Wnumber  OutputSDF  WarnOnEmptyStructure

*Output*

The same as for GetINCHI except for that only standard InChI is produced.

*Return codes*

The same as for GetINCHI.
FreeStdINCHI

void INCHIDECL FreeStdINCHI(inchi_Output*out)

Description

This is a “standard” counterpart of FreeINCHI which should be called to deallocate char* pointers obtained from each GetStdINCHI call.

GetStructFromStdINCHI

int INCHIDECL GetStructFromStdINCHI(inchi_InputINCHI*inpInChi,
                        inchi_OutputStruct*outStruct)

Description

This is a “standard” counterpart of GetStructFromINCHI.

Input

The same as for GetStructFromINCHI.

Output

The same as for GetStructFromINCHI.

Return codes

The same as for GetStructFromINCHI.

FreeStructFromStdINCHI

void INCHIDECL FreeStructFromStdINCHI(inchi_OutputStruct*out)
**Description**

Should be called to deallocate pointers obtained from each GetStructFromINCHI.

**Free_std_inchi_Input**

```c
void INCHI_DECL Free_std_inchi_Input( inchi_Input *pInp )
```

**Description**

This is a “standard” counterpart of Free_inchi_Input

**Get_std_inchi_Input_FromAuxInfo**

```c
int INCHI_DECL Get_std_inchi_Input_FromAuxInfo(char *szInchiAuxInfo,
                                                  int bDoNotAddH,
                                                  InchiInpData *pInchiInp )
```

**Description**

This is a “standard” counterpart of Get_std_inchi_Input_FromAuxInfo.
InChI API - modularized – general-purpose

The main purpose of modularized interface of InChI library is to modularize the process of InChI generation by separating normalization, canonicalization, and serialization stages. Using these API functions allows, in particular, checking intermediate normalization results before performing further steps and getting diagnostics messages from each stage independently. The functions use exactly the same `inchi_input` and `inchi_output` data structures as “classic” InChI API functions do. However, a new data structure, `INCHIGEN_DATA`, has been added to expose the normalization results (see `inchi_api.h` header file).

A typical process of InChI generation with this API calls is as follows.

1) Get handle of a new InChI generator object:
   
   ```c
   HGen = INCHIGEN_Create();
   ```

2) read a molecular structure and use it to initialize the generator:
   
   ```c
   result = INCHIGEN_Setup(HGen, pGenData, pInp);
   ```

3) normalize the structure:
   
   ```c
   result = INCHIGEN_DoNormalization(HGen, pGenData);
   ```
   optionally, look at the results;

4) obtain canonical numberings:
   
   ```c
   result = INCHIGEN_DoCanonicalization(HGen, pGenData);
   ```

5) serialize, i.e. produce InChI string:
   
   ```c
   retcode=INCHIGEN_DoSerialization(HGen, GenData, pResults);
   ```

6) reset the InChI generator
   
   ```c
   INCHIGEN_Reset(HGen, pGenData, pResults);
   ```
   and go to step 2 to read next structure, or

7) Finally destroy the generator object and free standard InChI library memories:
   
   ```c
   INCHIGEN_Destroy(HGen);
   ```
**INCHIGEN_Create**

INCHIGEN_HANDLE INCHI_DECL INCHIGEN_Create(void)

**Description**

InChI Generator: create generator.

Once the generator is created, it may be used repeatedly for processing the new structures. Before repetitive use, the pair of calls INCHIGEN_Reset / INCHIGEN_Setup should occur.

**Returns**

The handle of InChI generator object or NULL on failure.

Note: the handle is used just to refer to the internal InChI library object, whose structure is invisible to the user (unless the user chooses to browse the InChI source code). This internal object is initialized and modified through the subsequent calls to INCHIGEN API functions.

**INCHIGEN_Setup**

```c
int INCHI_DECL INCHIGEN_Setup(INCHIGEN_HANDLE HGen,
                               INCHIGEN_DATA * pGenData,
                               inchi_Input * pInp)
```

**Description**

InChI Generator: initialization stage (storing a specific structure in the generator object).

Note: INCHIGEN_DATA object contains intermediate data visible to the user, in particular, the string accumulating diagnostic messages from all the steps.
**Input**

INCHIGEN_HANDLE HGen is one obtained through INCHIGEN_Create call.

INCHIGEN_DATA * pGenData is created by the caller. It need not to be initialized.

Data structure inchi_Input * pInp is the same as for GetINCHI.

**Return codes**

The same as for GetINCHI.

**INCHIGEN_DoNormalization**

```c
int INCHI_DECL INCHIGEN_DoNormalization(INCHIGEN_HANDLE HGen, INCHIGEN_DATA * pGenData)
```

**Description**

InChI Generator: perform structure normalization.

Should be called after INCHIGEN_Setup.

Note: INCHIGEN_DATA object explicitly exposes the intermediate normalization data, see inchi_api.h.

**Input**

INCHIGEN_HANDLE HGen and INCHIGEN_DATA *pGenData as they are after calling INCHIGEN_Setup.

**Return codes**

The same as for GetINCHI.
**INCHIGEN_DoCanonicalization**

```c
int INCHI_DECL INCHIGEN_DoCanonicalization(INCHIGEN_HANDLE HGen,
                                           INCHIGEN_DATA * pGenData)
```

**Description**

InChI Generator: perform structure canonicalization.

Should be called after **INCHIGEN_DoNormalization**.

**Input**

INCHIGEN_HANDLE HGen and INCHIGEN_DATA *pGenData as they are after calling
INCHIGEN_DoNormalization.

**Return codes**

The same as for **GetINCHI**.

**INCHIGEN_DoSerialization**

```c
int INCHI_DECL INCHIGEN_DoSerialization(INCHIGEN_HANDLE HGen,
                                          INCHIGEN_DATA * pGenData,
                                          inchi_Output * pResults)
```

**Description**

InChI Generator: perform InChI serialization.

Should be called after **INCHIGEN_DoCanonicalization**.
Input

INCHIGEN_HANDLE HGen and INCHIGEN_DATA *pGenData as they are after calling INCHIGEN_DoCanonicalization.

Return codes

The same as for GetINCHI.

INCHIGEN_Reset

void INCHI_DECL INCHIGEN_Reset(INCHIGEN_HANDLE HGen,
    INCHIGEN_DATA * pGenData,
    inchi_Output * pResults)

Description

InChI Generator: reset (use before calling INCHIGEN_Setup(…) to start processing the next structure and before calling INCHIGEN_Destroy(…))

Input

INCHIGEN_HANDLE HGen and INCHIGEN_DATA *pGenData as they are after calling INCHIGEN_DoSerialization.

Return codes

The same as for GetINCHI.
**INCHIGEN_Destroy**

```c
void INCHI_DECL INCHIGEN_Destroy(INCHIGEN_HANDLE HGen)
```

**Description**

Destroys the generator object and frees associated InChI library memories.

Important: make sure `INCHIGEN_Reset(...)` is called before calling `INCHIGEN_Destroy(...)`.

**Input**

The handle of InChI generator object.

**InChI API - modularized – standard InChI subset**

Described below are “standard” counterparts of general-purpose functions; these “standard” API calls are retained for compatibility reasons.

**STDINCHIGEN_Create**

```c
INCHIGEN_HANDLE INCHI_DECL STDINCHIGEN_Create(void)
```

**Description**

Standard InChI Generator: create generator.

This is a “standard” counterpart of `INCHIGEN_Create`. 


**Returns**

The handle of standard InChI generator object or NULL on failure. Note: the handle serves to access the internal object, whose structure is invisible to the user (unless the user chooses to browse the InChI library source code which is open).

**STDINCHIGEN Setup**

```c
int INCHI_DECL STDINCHIGEN_Setup(INCHIGEN_HANDLE HGen,
                                   INCHIGEN_DATA * pGenData,
                                   inchi_Input * pInp)
```

**Description**

Standard InChI Generator: initialization stage (storing a specific structure in the generator object).

This is a “standard” counterpart of INCHIGEN_Setup.

Note: INCHIGEN_DATA object contains intermediate data visible to the user, in particular, the string accumulating diagnostic messages from all the steps.

**Input**

INCHIGEN_HANDLE HGen is one obtained through INCHIGEN_Create call.

INCHIGEN_DATA * pGenData is created by the caller.

Data structure inchi_Input * pInp is the same as for GetINCHI.

**Return codes**

The same as for GetStdINCHI.
STDINCHIGEN_DoNormalization

int INCHI_DECL STDINCHIGEN_DoNormalization(INCHIGEN_HANDLE HGen,
                                           INCHIGEN_DATA * pGenData)

Description

Standard InChI Generator: perform structure normalization.

The entry is “standard” counterpart of INCHIGEN_DoNormalization.

STDINCHIGEN_DoCanonicalization

int INCHI_DECL STDINCHIGEN_DoCanonicalization(INCHIGEN_HANDLE HGen,
                                                 INCHIGEN_DATA * pGenData)

Description

Standard InChI Generator: perform structure canonicalization.

The entry is “standard” counterpart of INCHIGEN_DoCanonicalization.

STDINCHIGEN_DoSerialization

int INCHI_DECL STDINCHIGEN_DoSerialization(
                                           INCHIGENHANDLE HGen,
                                           INCHIGENDATA * GenData,
                                           inchi_Output * pResults)

Description

Standard InChI Generator: perform InChI serialization.
The entry is “standard” counterpart of INCHIGEN_DoSerialization.

**STDINCHIGEN_Reset**

```c
void INCHI_DECL STDINCHIGEN_Reset(INCHIGEN_HANDLE HGen,
    INCHIGEN_DATA * pGenData,
    inchi_Output * pResults);
```

**Description**

Standard InChI Generator: reset (use before calling STDINCHIGEN_Setup(…) to start processing the next structure and before calling STDINCHIGEN_Destroy(…))

The entry is “standard” counterpart of INCHIGEN_Reset.

**STDINCHIGEN_Destroy**

```c
INCHI_API void INCHI_DECL STDINCHIGEN_Destroy(INCHIGEN_HANDLE HGen)
```

**Description**

Destroys the standard InChI generator object and frees associated InChI library memories.

This is a “standard” counterpart of INCHIGEN_Destroy.

Important: make sure STDINCHIGEN_Reset(…) is called before calling STDINCHIGEN_Destroy(…).
InChIKey API – general-purpose

GetINCHIKeyFromINCHI

int INCHI_DECL GetINCHIKeyFromINCHI(const char* szINCHISource,
const int xtra1, const int xtra2,
char* szINCHIKey,
char* szXtra1, char* szXtra2);

Description

Calculate InChIKey from InChI string.

Input

szINCHISource – source null-terminated InChI string.
xtra1 = 1 calculate hash extension (up to 256 bits; 1st block)
xtra2 = 1 calculate hash extension (up to 256 bits; 2nd block)

Output

szINCHIKey - InChIKey string, null-terminated. The user-supplied buffer szINCHIKey should be at least 28 bytes long.
szXtra1 - hash extension (up to 256 bits; 1st block) string. Caller should allocate space for 64 characters + trailing NULL.
szXtra2 - hash extension (up to 256 bits; 2nd block) string. Caller should allocate space for 64 characters + trailing NULL.

Return codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>INCHIKEY_OK</td>
<td>0</td>
<td>Success; no errors or warnings</td>
</tr>
<tr>
<td>INCHIKEY_UNKNOWN_ERROR</td>
<td>1</td>
<td>Unknown program error</td>
</tr>
</tbody>
</table>
INCHIKEY_EMPTY_INPUT         2  Source string is empty
INCHIKEY_INVALID_INCHI_PREFIX 3  Invalid InChI prefix or invalid version (mot 1)
INCHIKEY_NOT_ENOUGH_MEMORY   4  Not enough memory
INCHIKEY_INVALID_INCHI       20 Source InChI has invalid layout
INCHIKEY_INVALID_STD_INCHI   21 Source standard InChI has invalid layout

**CheckINCHIKey**

```c
int INCHI_DECL CheckINCHIKey(const char *szINCHIKey)
```

**Description**

Check if the string represents valid InChIKey.

**Input**

szINCHIKey - source InChIKey string

**Return codes**

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>INCHIKEY_VALID_STANDARD</td>
<td>0</td>
<td>InChIKey is valid and standard</td>
</tr>
<tr>
<td></td>
<td>-1</td>
<td>InChIKey is valid and non-standard</td>
</tr>
<tr>
<td>INCHIKEY_VALID_NON_STANDARD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INCHIKEY_INVALID_LENGTH</td>
<td>1</td>
<td>InChIKey has invalid length</td>
</tr>
<tr>
<td>INCHIKEY_INVALID_LAYOUT</td>
<td>2</td>
<td>InChIKey has invalid layout</td>
</tr>
<tr>
<td>INCHIKEY_INVALID_VERSION</td>
<td>3</td>
<td>InChIKey has invalid version number (not equal to 1)</td>
</tr>
</tbody>
</table>
**InChIKey API – standard InChI subset**

Described below is “standard” counterpart of general-purpose function; this “standard” API call is retained for compatibility reasons.

**GetStdINCHIKeyFromStdINCHI**

```c
int INCHI_DECL GetStdINCHIKeyFromStdINCHI(
    const char* szINCHISource,
    char* szINCHIKey);
```

**Description**

Calculate standard InChIKey from standard InChI string.

"Standard" counterpart of GetINCHIKeyFromINCHI.

For compatibility with v. 1.02-standard, no extra hash calculation is allowed. To calculate extra hash(es), use GetINCHIKeyFromINCHI with stdInChI as input.

**Input**

szINCHISource – source null-terminated InChI string.

**Output**

szINCHIKey - InChIKey string, null-terminated. The user-supplied buffer szINCHIKey should be at least 28 bytes long.

**Return codes**

The same as for GetINCHIKeyFromINCHI.
InChI API – miscellaneous

**GetStringLength**

```c
int INCHI_DECL GetStringLength( char *p )
```

**Description**

Returns string length.

**Examples of InChI API use**

The distribution package of InChI software v. 1.04 contains the two examples of API usage.

1. The first one is C calling program located in `inchi_main/` subfolder of `INCHI-1-API/INCHI_API/` folder. This program calls InChI library `libinchi.dll` under Microsoft Windows or `libinchi.so` under Linux or Unix (note that the program is just a sample which is not supposed to be used for the production).

   Defining `CREATE_INCHI_STEP_BY_STEP` in `e_mode.h` makes the program use the modularized interface to InChI generation process. This is the default option. Commenting out the line containing this `#define` makes the program use “classic” (“GetINCHI”; software version 1.01-style) interface. The both options provide examples of using interface to the InChIKey part of the library.

   If the testing application is compiled with `CREATE_INCHI_STEP_BY_STEP` option, an additional defining of `OUTPUT_NORMALIZATION_DATA` in `e_mode.h` makes the program output the intermediate (normalization) data into the log file. The related data
structures are described in header file \texttt{inchi\_api.h}; their use is exemplified in \texttt{e\_ichimain\_a.c} file. Note that including the intermediate (normalization) data in the output may produce a very long log file.

Folder \texttt{INCHI-1-API/INCHI\_API/vc9/inchi\_dll/} contains a MS Visual C++ 2008 project to build dynamically linked library \texttt{libinchi.dll} under Windows.

Folder \texttt{INCHI-1-API/INCHI\_API/vc9/inchi\_main/} contains a MS Visual C++ 2008 project to build both dynamically linked library \texttt{libinchi.dll} and the testing application \texttt{InChI\_MAIN.exe} under Windows (both library and executable are placed into subfolders \texttt{Release} or \texttt{Debug} of \texttt{vc6\_INCHI\_DLL} folder).

Folder \texttt{INCHI-1-API/INCHI\_API/gcc\_so\_makefile} contains a gcc makefile for creating \texttt{InChI} library as a Linux shared object dynamically linked to the main program.

2. The second example illustrates how the \texttt{InChI} library (Windows DLL/Linux .so) functions may be accessed from within Python. Source code of a sample program is in the folder \texttt{INCHI-1-API/INCHI\_API/python\_sample}. The program has a simple Mol/SDfile reader and produces InChI strings and, optionally, generates InChIKey codes.

More details on these testing applications may be found in \texttt{readme.txt} files in the corresponding directories and in source codes.