InChI Software Version 1.02-Beta

This is a beta release of InChI version 1 (software version 1.02.)

This beta release package includes C source code of InChI library (API) and stand-alone executable, cInChI. Both have new features described in this document. Also, the examples of using the new functionality are provided. Finally, several minor bugs have been fixed since the official 1.01 software release in August 2006. Other components, such as InChI documentation, wInChI.exe Windows executable, validation set, etc. will be included in the final release.

Table of Contents

New software features

1. InChIKey and InChIKey API

The InChIKey is a character signature based on a hash code of the InChI string.

A hash code is a fixed length condensed digital representation of a variable length character string. Providing a hash derived from an InChI string should be helpful in search applications, including Web searching and chemical structure database indexing; also, this hash may serve as a checksum for verifying InChI, for example, after transmission over a network.

InChIKey has four (4) distinct components: a 14-character hash of the basic (Mobile-H) InChI layer (without /p segment accounting for added or removed protons); a 8-character hash of the remaining layers; a 1 character is a flag indicating selected features (e.g. presence of fixed-H layer); a 1 character is a “check” character. The overall length of InChIKey is fixed at 25 characters, including separator:

AAAAAAAAAAAAAA-BBBBBBBBCD

This is significantly shorter than a typical InChI string (for example, the average length of InChI string for Pubchem collection is 146 characters).

InChIKey layout is as follows:

AAAAAAAAAAAAAA
First block (14 letters)
Encodes molecular skeleton (connectivity)
BBBBBBBB
Second block (8 letters)
Encodes proton positions (tautomers), stereochemistry, isotopomers, reconnected layer
C
Flag character
Indicates InChI version, presence of a fixed-H layer, isotopes, and stereochemical information.
D
Check character, obtained from all symbols except delimiters, i.e. from
AAAAAAAAAAAAAABBBBBBBBC

All symbols except the delimiter (a dash, that is, a minus) are uppercase English letters representing a "base-26" encoding.

Hash blocks
The two hash blocks of InChIKey are based on truncated SHA-256 cryptographic hash function (http://en.wikipedia.org/wiki/SHA_hash_functions#SHA-2).

A theoretical – optimistic – estimate of collision resistance (i.e., the minimal size of a database at which a single collision is expected, that is, an event of two hashes of two different InChI strings being equal) is 6.1×10^{9} molecular skeletons × 3.7×10^{5} stereo/protonation/isotopic substitution isomers per each skeleton ´ 2.2×10^{15}.
To exemplify: the probability of a single first block collision in a database of 1 billion compounds is 1.3%. In other words, a single first block collision is expected in 1 out of 75 databases of 10^{9} compounds each. For 10^{8} (100 million) compounds in a database this probability is 0.014%.

Alternative estimate of collision resistance is by a chance of an accidental collision upon adding a new entry to an existing collection. For a collection of 1 billion different InChIKey entries, the estimated probability of an accidental collision of the first layers for a newly added structure is 2.7×10^{-9} % and for both layers is $2.0*10^{-20}$ %.

Check character

InChIKey check character is simply a checksum which may be tested to verify the InChIKey string.

Flag character

Values of InChIKey flag character (A..X) are explained below. In Table 1, the first column of letters means InChI v. 1, 2nd - v.2 and 3rd - post-v.2. Since the current InChI version is 1, only letters A..H are actually used.

Table 1. Interpretation of the InChIKey flag character

isotopic	fixedH	stereo	flag char dependent on InChI version (current is v. 1)		
			v.1	v.2	v.2+
0	0	0	**A**	I	Q
0	0	1	**B**	J	R
0	1	0	**C**	K	S
0	1	1	**D**	L	T
1	0	0	**E**	M	U
1	0	1	**F**	N	V
1	1	0	**G**	O	W
1	1	1	**H**	P	X

Example:

'F' means InChI version 1, structure has isotopic and stereo but not fixed-H layers.

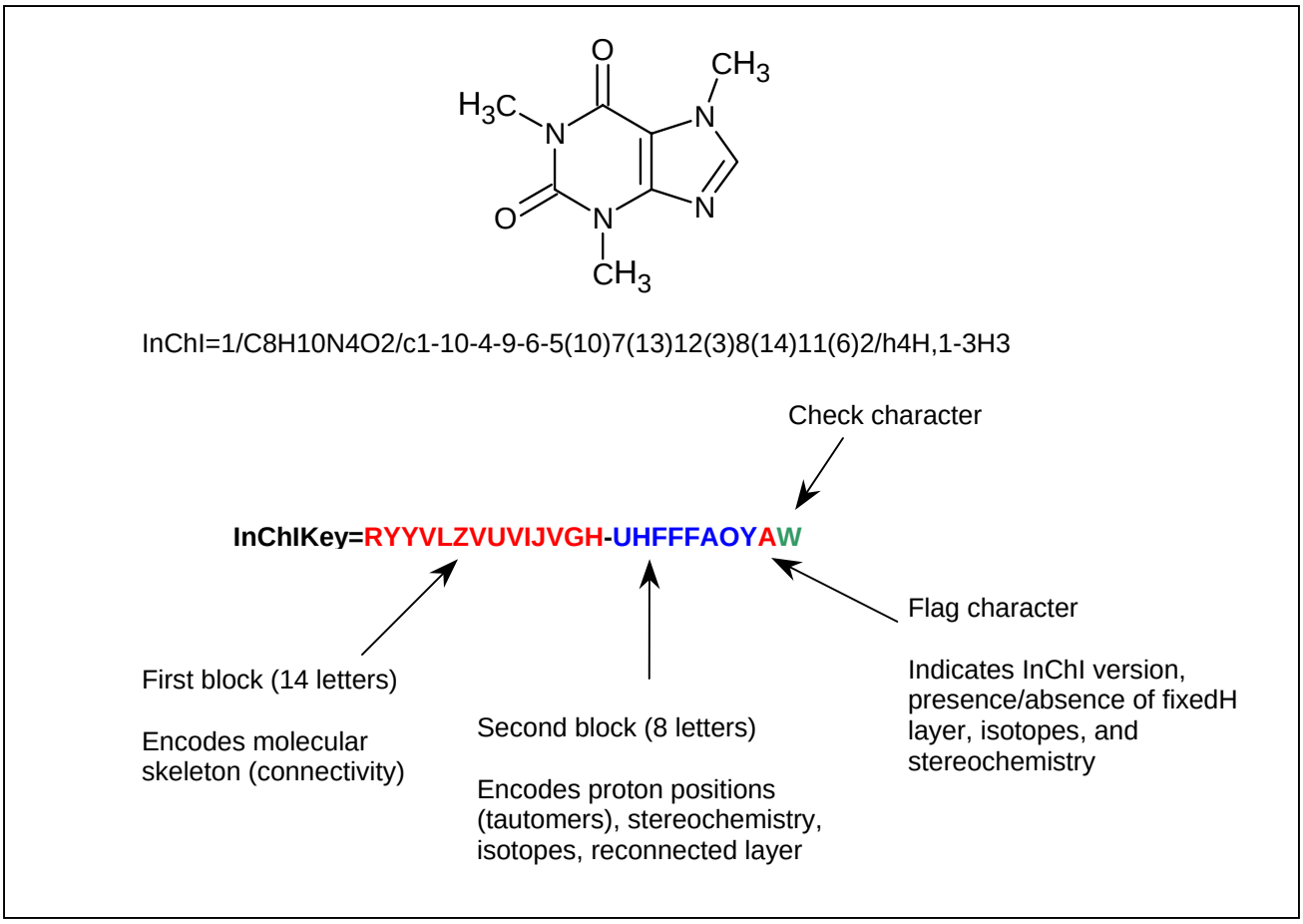

Fig. 1. InChIKey example.

InChIKey API functions

Two InChI API functions have been added to support InChIKey (see inchi_api.h header file).

int GetINCHIKeyFromINCHI(const char* szINCHISource, char* szINCHIKey)

Calculate InChIKey from InChI string.
Input: szINCHISource – source null-terminated InChI string
Output: szINCHIKey - InChIKey string, null-terminated
Returns: 0 => Success, any other value is an error code (see inchi_api.h)
The user-supplied buffer szINCHIKey should be at least 26 bytes long.
NB: the only attempt to check if the input szINCHISource represents a valid InChI identifier is: the string should start with "InChI=1/" followed by at least one of a..zA..Z0..9 or '/' characters.

int CheckINCHIKey(const char* szInchiKey)

Check if the string represents a valid InChIKey.

Checks for both a proper letter layout and whether the check-character matches the InChIKey.
Input: szInchiKey – InChIKey string
Returns: 0 => valid key, any other value is an error code (see inchi_api.h).

Q&A on InChIKey

Question 1: Can InChI be restored/decrypted from its InChIKey?
Answer:
No. To find the InChI that generated an InChIKey, you need a cross-reference or lookup table. The situation is similar to that with the Internet DNS lookup which resolves host name to the IP address.

In general, it is expected that InChI lookup service will be provided by Internet tools. Both the InChI and InChIKey are readable and searchable by web search engines (such as Google, Microsoft, Yahoo, Ask, and so on). It is expected that InChI lookup tables will be populated by web search/crawl programs. Naturally, for stand-alone databases a lookup service may be added by developers/maintainers.

Note also that you may extract some information about the chemical structure and the InChI by examining a flag character of InChIKey.

Question 2: What is the collision resistance of the InChIKey? Has it been tested on real-life examples?
Answer:
InChIKey hash consists, internally, of 102 bits: 65 for the first block (molecular skeleton, or connectivity) and 37 for the second one (stereo/protonation/isotopic substitution isomers).

This suggests that a theoretical – optimistic – estimate of the collision-resistance (corresponds to 50% probability of a single collision) is 6.1 billion molecular skeletons × × 3.7×10^{5} stereo/protonation/isotopic substitution isomers per each skeleton ´ 2.2×10^{15}.

Alternatively, for a collection of 1 billion different InChIKey entries, the estimated probability of an accidental collision of the first layers for a newly added structure is 2.7×10^{-9} % and for both layers is 2.0×10^{-20} %.

The InChIKey was tested on databases of InChI strings created out of real and generated structures, derived from:

- *Zinc* (~4×10^{6} entries, real structures, http://zinc.docking.org/),
- *PubChem* (~10×10^{6}, real, http://pubchem.ncbi.nlm.nih.gov/),
- *GDB* (~26×10^{6}, generated; courtesy of Prof. J.-L. Reymond, University of Berne, http://dcbwww.unibe.ch/groups/reymond/, private communication),
- *FP42* (~42×10^{6}, custom-generated),
- *OVERALL* (~77×10^{6}, all of the above merged, duplicates excluded; real+generated).

No hash collisions were observed in any of these databases.

However, this does not imply that the collisions are impossible. In fact, due to the very essence of hash functions, collisions are unavoidable in sufficiently large collections. Some recent estimates of chemical space size for small molecules are in excess of 10^{60}, and for proteins it is 10^{390} [Nature, 2004, **432**(7019), Insight, pp. 823-865, http://www.nature.com/nature/insights/7019.html and refs. therein]. T. Fink et al. in "Virtual Exploration of the Small-Molecule Chemical Universe below 160 Daltons", [Angew. Chem. Int. Ed. 2005, **44**(10), pp. 1504-1508] quote an estimate of 10^{18}-10^{200}.

Also, the estimates of hash collision probabilities given above are for an ideal hash and may not be valid in practice because of unknown yet properties of SHA-2 hash.

Question 3: Why do you use only 26 letters of the alphabet and not 26 letters plus 10 digits? Why don't you use a hexadecimal notation?

Answer:
– This is just a representation issue. In fact, the same hash may be represented by letters, digits, letters and digits and even with bare 0s and 1s (as it actually is represented internally, in computer memory).

However, representation issues may appear critical for applications like publishing or Web search. In particular, search engines may tend to break the text "on the border" between letters and non-letters, trying to detect "words" since the words of human languages do not contain digits or punctuation marks.

Though the exact behavior may vary from one Web search engine to another and from context to context (and even change with time for the same Web search engine), it is more robust to have nothing but letters in the InChIKey. Using only letters increases chances that a search engine would consider InChIKey as a single "word" (or phrase) and would index it as such.

Also, the robust approach includes use of only upper-case letters as most if not all search engines may not differentiate between upper- and lower-case letters. For example, [at the moment] Google reports the same number of 40,600,000 results for quoted strings "hash", "HASH" and "hAsh".

Naturally, this means that we somewhat sacrifice hash length for the sake of robustness (the more symbols are in the "alphabet", the shorter is the hash representation).

Question 4: How the check character is calculated?
Answer:
– The check character is obtained from all the hash characters and the flag character, dash excluded. The algorithm of check character calculation is explained below.

Each character c_i, i=1–23, in the hash string $c_1c_2..c_{14}$-$c_{15}..c_{22}c_{23}$ is considered as a base-26 representation of a number k_i (letters A-Z are mapped on numbers 0-25), thus forming vector $K = (k_1, k_2\ldots, k_{23})$. Base-26 check-number k_{24} is calculated from a dot product of K and the weight vector W modulo 26, so that

$(k_1, k_2\ldots, k_{23})\cdot(w_1, w_2\ldots, w_{23}) = k_{24} \pmod{26}$

After that k_{24} is mapped on the corresponding letter to obtain the check-character c_{24}.

The weight vector W is as follows: (1, 3, 5, 7, 9, 11, 15, 17, 19, 21, 23, 25, 1, 3, 5, 7, 9, 11, 15, 21, 23, 25, 1). Weights are chosen to be co-prime with 26 (that is, they have no common divisor except 1). This ensures that the check-character detects any single-letter error in the source string (see, e.g. *J.A. Gallian, "Error detection methods", ACM Computing Surveys, 1996, vol. 28, pp. 504-517*). It also detects most of transposition errors (like $ab \rightarrow ba$ or $abc \rightarrow cba$).

The InChI library API provides function **CheckINCHIKey,** which determines whether the check character in the InChIKey string matches the string itself.

Question 5: Why did you use a strong and time-consuming cryptographic SHA-2 256-bit hash function since it anyway is truncated to obtain the InChIKey? Is it just a wasting of the processor time?

Answer:

– Our tests revealed that computing the InChIKey is quite affordable in terms of CPU time. Hashing the entire InChI set generated from PubChem required less than 5 min on a machine with Pentium Dual Core 2.8 GHz CPU. Additionally, using cryptographic (i.e., producing – hopefully – a strongly randomized output) hash function increases the chances that collision resistance will be as close to the theoretical limit as possible. Note that the truncation of the hash is directly allowed by the SHA-2 description (http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf).

2. Newly added interface to the InChI generation process

Seven new API functions were added to InChI software version 1.01 interface to InChI library. These functions are described in inchi_api.h header file.

The main purpose was 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.

New InChI generation API functions

INCHIGEN_HANDLE INCHIGEN_Create(void)
InChI Generator: create generator
Returns: a handle of the 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).

int INCHIGEN_Setup(INCHIGEN_HANDLE HGen, INCHIGEN_DATA *pGenData, inchi_Input * pInp)
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.

int INCHIGEN_DoNormalization(INCHIGEN_HANDLE HGen, INCHIGEN_DATA * pGenData)
InChI Generator: perform structure normalization
Note: INCHIGEN_DATA object explicitly exposes the intermediate normalization data, see below.

int INCHIGEN_DoCanonicalization(INCHIGEN_HANDLE HGen, INCHIGEN_DATA * pGenData)
InChI Generator: perform structure canonicalization

int INCHIGEN_DoSerialization(INCHIGEN_HANDLE HGen, INCHIGEN_DATA * pGenData, inchi_Output * pResults)
InChI Generator: perform InChI serialization

void INCHIGEN_Reset(INCHIGEN_HANDLE HGen, INCHIGEN_DATA * pGenData, inchi_Output * pResults)
InChI Generator: reset (use before calling INCHIGEN_Setup(...) to start processing the next structure and before calling INCHIGEN_Destroy(...))

void INCHIGEN_Destroy(INCHIGEN_HANDLE HGen)
InChI Generator: destroy the generator object HGen. Important: make sure INCHIGEN_Reset(...) is called before calling INCHIGEN_Destroy(...).

The functions use exactly the same inchi_Input and inchi_Output data structures as other 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:
   HGen = INCHIGEN_Create();

2. read a molecular structure and use it to initialize the generator:
   result = INCHIGEN_Setup(HGen, pGenData, pInp);
3. normalize the structure:
   result = INCHIGEN_DoNormalization(HGen, pGenData);
   optionally, look at the results;
4. obtain canonical numberings:
   result = INCHIGEN_DoCanonicalization(HGen, pGenData);
5. serialize, i.e. produce InChI string:
   retcode = INCHIGEN_DoSerialization(HGen, pGenData, pResults);
6. reset the InChI generator
   INCHIGEN_Reset(HGen, pGenData, pResults)
   and go to step 2 to read next structure,
   or

7. Finally destroy the generator object and free InChI library memories:
   INCHIGEN_Destroy(HGen);

Examples of InChIKey API use

Besides the InChI software library software itself (located in INCHI_API/INCHI_DLL folder), which contains InChI generating code (libinchi.so or libinchi.dll library source code), this package contains examples of using previously available and newly added InChI software library functionality.

Folder ***INCHI_API/INCHI_MAIN*** contains ANSI C testing application source code to call InChI library libinchi.dll under Microsoft Windows or libinchi.so under Linux or Unix.

Defining CREATE_INCHI_STEP_BY_STEP in e_mode.h makes the program use the new (1.02) interface to InChI generation process. This is the default option. Commenting out the line containing this *#define* makes the program use software version 1.01 interface to InChI generation process. However, both options provide examples of using the new, software version 1.02, 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 inchi_api.h; their use is exemplified in e_ichimain_a.c file. Note that including the intermediate (normalization) data in the output may produce a very long log file.

Folder ***INCHI_API/vc6_INCHI_DLL*** contains a MS Visual C++ 6.0 project to build dynamically linked library libinchi.dll under Windows.

Folder ***INCHI_API/ vc6_INCHI_MAIN*** contains a MS Visual C++ 6.0 project to build both dynamically linked library libinchi.dll and the testing application InChI_MAIN.exe under Windows (both library and executable are placed into subfolders Release or Debug of vc6_INCHI_DLL folder).

Folder ***INCHI_API/gcc_so_makefile*** contains a gcc makefile for INCHI_MAIN + INCHI_DLL code to create InChI library as a shared object (Linux) or dll (Windows) dynamically linked to the main program.

Folder ***INCHI_API/gcc_makefile*** contains a gcc makefile for INCHI_MAIN + INCHI_DLL code to create a single statically linked executable under Win32 or Linux, or other operating systems.

Folder ***INCHI_API/INCHI_ADDKEY*** contains ANSI C application source code that calls InChIKey part of the library libinchi (to execute INCHI_ADDKEY.exe the libinchi.dll is needed.)

The program copies the input file to the output file inserting InChIKey string as a separate line after each line containing a single InChI string. All input strings are copied to the output file unchanged (or, optionally, not copied).

Note: a line is assumed to contain an InChI string if it meets the following requirements:

- the line starts with "InChI=1/" string;
- this string is followed by at least one of a..zA..Z0..9 or '/' characters.

Everything between "InChI=1/" and the end of line is considered an InChI string. Note: this program is provided only for demo purpose.

Folder ***INCHI_API/vc6_INCHI_ADDKEY*** contains a MS Visual C++ 6.0 project to build INCHI_ADDKEY.exe executable (Windows).

Folder ***INCHI_API/gcc_makefile_inchi_addkey*** contains a gcc makefile for inchi_addkey (Linux) or inchi_addkey.exe (Windows) executable.

Folder ***INCHI_API/RunInChI_Py*** contains a sample program illustrating how the InChI library (Windows DLL/Linux .so) functions may be called from within Python. It 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 readme.txt files in the corresponding directories and in source codes.

3. Stand-alone executable (cInChI) with InChIKey support

Besides the InChI software library software, the package includes stand-alone executable cInChI (located in cInChI folder) and its source C code. This program differs from previously available cInChI v. 1.01 by added support of InChIKey generation. To generate InChIKey, a new command-line option is added: “Key” (prefixed by “/” under Windows and by “-“ under Linux).

Folder ***cInChI/common*** contains cInChI ANSI C code which is in common with the InChI library. Folder ***cInChI/main*** contains the rest of source code.

Folder ***cInChI/vc6_project*** contains a MS Visual C++ 6.0 project to build cInChI.exe executable (Windows).

Folder ***cInChI/gcc_makefile*** contains a gcc makefile for building executable under Win32 or Linux, or other operating systems.

Bugs fixed

The main purpose of the fixes is to withstand malicious attempts to attack a Web server by providing a specially designed InChI string input to InChI binaries.

The files contain fixes to bugs discovered by W. D. Ihlenfeldt (4/11/2007), A. Dalke (until 6/30/2007), Anonymous (03/29/2007, req. 1690823), and a few other minor fixes. For descriptions of the bugs see
http://sourceforge.net/mailarchive/forum.php?thread_name=028b01c77cac%24c721f8e0%248801a8c0%40xempc3&forum_name=inchi-discuss and
http://sourceforge.net/tracker/?atid=741489&group_id=136669&func=browse

The following preprocessor directive activates most of the fixes

#define FIX_DALKE_BUGS 1

included in mode.h header files. A few other fixes have been applied unconditionally.

Some of the features implemented in these fixes are:
in reading InChI string (inchi2inchi and invhi2struct modes):
- restriction on the number of atoms in a component (not more than 1024)
- restriction on the number of components (not more than 1024)
- restriction on max. component charge (not more than +255 and not less than -255)

in converting InChI to InChI (inchi2inchi mode):
- As an improvement, InChI formula layer reconstruction instead of copying was added in case of inchi2inchi option.

In converting InChI to structure (inchi2struct mode):
- Fixed false detection of “Stereo centers/allenes: Falsely inverted”

Q&A on Bugs Fixed

Question 1: Do these changes affect output for valid input?
Answer:
– The fix to the bug discovered by W. D. Ihlenfeldt may affect structure to InChI conversion output in case of the normalization described in InChI Technical Manual, p.13, the last row, 3, in "2-3. Non-terminal fragments - 3 atoms" table, as mentioned in http://sourceforge.net/mailarchive/forum.php?thread_name=028b01c77cac%24c721f8e0%248801a8c0%40xempc3&forum_name=inchi-discuss)
This may affect coordination compounds with substructure C=N-C where N has an additional bond to a metal atom (so that the total formal valence of N is greater than 3).The reason for the possible output change is the nature of the bug: in case when

the bug does not cause a crash it might produce results different from the cited description on page 13 of the InChI Technical Manual. The fix should produce results compliant to the InChI Technical Manual.
– Changes to the reading InChI string code may lead to a rejection of an input InChI; in important cases the v1.01 would crash on such an input. However, one may create an InChI that could be processed by the current v1.01 and refused by this fixed version. An example may contain, for example, a charge=+256 in the /q layer.

Question 2: Do these changes affect output for invalid input?
Answer:
– Changes to the reading InChI string code may cause a rejection of certain invalid InChI that would be accepted by v.1.01.
In case of inchi2inchi mode (read InChI, rearrange/resort the components, output InChI without canonicalization and normalization) a changed output may happen in case of an invalid input InChI; this may only help to detect an invalid InChI.