IUPAC International Chemical Identifier (InChI)

InChI version 1, software version 1.03 (2010)

InChI Source Code Documentation

Stephen E. Hull, John M. Barnard and Daniel G. Thomas

Digital Chemistry Ltd.
30 Kiveton Lane
Todwick
Sheffield S26 1HL

john.barnard@digitalchemistry.co.uk

Prepared by Digital Chemistry Ltd. under contract to the InChI Trust

30 March 2011

InChI TRUST

digital chemistry

Table of Contents

1 Introduction

This document has been prepared as a “guide” to the InChI source code, with the aim of allowing developers who are maintaining, modifying or extending it to understand the basis on which it has been written, the principles underlying its organization and the main algorithms it implements. It is not the aim of this document to reverse engineer the code, nor to provide a complete functional specification for InChI, but rather to provide an insight into how the code is constructed and what algorithms are used.

The document outlines the main control paths within the code and points developers to the areas of code that are relevant to particular functionality. It deliberately does not duplicate the information in the Technical Manual (Stein *et al.,* 2010), but where appropriate makes reference to relevant sections of the Technical Manual, which should therefore be read in conjunction with it. Other relevant documents are also referenced where appropriate, and are listed in Section 23.

No program documentation can ever be considered truly comprehensive, and though we have attempted to make this document as complete as possible within the time constraints imposed on its preparation, some aspects of the code are inevitably discussed in less detail than others. It should be noted that this document has not been written by the original authors of the InChI software, though where possible the original authors have been consulted for clarification on certain points. Due to constraints on their availability, it has not been possible to obtain detailed clarification on some aspects of the code, in particular the use of "balanced network search" for tautomer normalization (see sections 16.8 to 16.18),and this is therefore described in outline only. Assumptions we have made here and elsewhere are noted in the text, though it is possible that some misinterpretations of the code may have crept in, and this should be borne in mind by readers.

1.1 Organization of Source Code Files

The InChI code consists of approximately 112,000 lines of C code containing approximately 970 functions and 130 macros in 37 source files. Generally the individual source files contain related functions. A list of source files, with notes on their contents, is given in Section 22.

1.2 Organization of this Document

Sections 2 to 4 of this document provide a general discussion of aspects of the source code, while Sections 5 to 12 each describe one of the principal high-level operations, identifying the actual functions involved in implementing them. Sections 13 to 20 describe the more important individual functions in some detail, grouped according to the high-level operations they are involved with. Sections 21 to 23 are in the nature of appendices, summarising the code in each source file, and providing a glossary and bibliographic references.

1.3 Conventions Used

Within this document the following conventions are used:

Source code filenames are shown in **bold type**.

Source code function names are shown in *italics*.

Identifiers and labels used in the source code are shown underlined.

2 Overall Control

The InChI code provides functionality that can be divided into several distinct areas. These are broadly:

1. An API that can be called by user written programmes. See the InChI API Reference (IUPAC 2010) for further details.
2. Reading MOL files (Symyx, 2010) into the internal program structures.
3. Production of InChI strings and InChIKey hash codes.
4. Normalising structures.
5. Handling isotopic H and heavy atoms.
6. Handling stereochemistry.
7. Canonicalisation of structures.

A general outline of each of these functions is given in its own section below. Further sections describe individual functions in the code in greater detail.

The exact route taken through the code is determined by command line parameters.

Because the InChI has a defined format and is designed to produce consistent results it is important that no changes are made to the code that cause it to give different results from earlier versions unless there were bugs in the earlier. This is seen as particularly important in structure canonicalisation where any changes to the code should give results that are provably identical to the earlier version. This is perhaps less significant for the other main areas of the system (normalisation/tautomerism, isotope handling, stereochemistry) where there may be possible changes in requirements, interpretation or theoretical advances. Note that even a comprehensive test suite of examples such as outlined in Digital Chemistry (2010a) would not be a guarantee that any changes result in equivalent code, although a test suite may highlight problems caused by changes to the system. Compliance with the test suite is a necessary but not sufficient condition for code equivalence. Any changes should be in accord with the principles set out by Fowler (1999). Some aspects of the work that needs to be undertaken if major changes were to be made to the InChI software have been discussed in Digital Chemistry (2010b).

3 Compilation Options

Unless otherwise stated no specific compilation options have been specified when describing the code functions.

Many options are fixed and defined in the file **mode.h**. Some of these options are fixed by definitions in the code and cannot be overridden at compile time (except by changing the code itself). It seems that compilation options have often been used as a means of recording changes from version to version as the system has developed. An alternative might have been to use a source control system.

Options available are:

Option	**Comment**
ENABLE_ENGINEERING_OPTIONS	Specifies that extra command line parameters are allowed. (Used in several functions in **ichiparm.h**.)
USE_STDINCHI_API	Seems to be set by default to allow non-standard InChI command line parameters.
INCHI_LIBRARY	Defined when compiling as an API.
INCHI_LIB	Seems to be always defined. Used to decide whether to generate messages and for output options.
READ_INCHI_STRING	Definition tied to INCHI_LIB. Comment in code is 'input InChI string and process it', but it is not clear what this does.
INCHI_STANDALONE_EXE	Defined when compiling as a command line executable.
INCHI_MAIN	Defined when compiling as a DLL.
_USRDLL	Defines exported functions for DLL.
INCHI_ALL_CPP	Allow C++ compilation/linkage of functions prototyped in .h files (used in conjunction with definition __cplusplus).
INCHI_ANSI_ONLY	Switch that appears to depend on the compiler being used (Microsoft or non-Microsoft?).
_DEBUG	As specified by the compiler options. Appears to decide whether breakpoint statements can be accessed for convenience when debugging. Often used in conjunction with bRELEASE_VERSION.
bRELEASE_VERSION	Set to 1 for release version; 0 for debug.

CHECK_WIN32_VC_HEAP	Enables heap checking. Normally switched off.
SEPARATE_CANON_CALLS	Used to profile canonicalisation functions.
INCHIKEY_DEBUG	Set to 2 if key debugging is required. Normally hard-coded to zero.
DISPLAY_DEBUG_DATA	Used in conjunction with DISPLAY_DEBUG_DATA_C_POINT for debugging.
DISPLAY_DEBUG_DATA_C_POINT	See DISPLAY_DEBUG_DATA.
NEVER	Protects code that will never be compiled; presumably should never be defined.
__UTIL_H__	Not used.
defined_NEIGH_LIST	Not used.
INCHI_USETIMES	Set when compiling to determine which version of the function *InchiClock* to use.
DO_NOT_TRACE_MEMORY_LEAKS	defined if memory leaks are to be traced.
TRACE_MEMORY_LEAKS	set to 1 if leaks are to be traced.
MY_REPORT_FILE	file associated with tracing memory leaks when not using the Visual C++ debugger.
REPEAT_ALL	looks like the number of repeats required and is presumably for timing tests.
USE_ALLOCA	determines which versions of *qmalloc* and *qfree* should be used. Always defined so that *inchi_malloc* and *inchi_free* are used.
ADD_NON_ANSI_FUNCTIONS	adds language extensions if not provided by build environment. Not normally required.
ADD_CMLPP	allows CML files to be input. May be overridden at compile time. Normally not set. Definitions of USE_CMLPPDLL and MSC_DELAY_LOAD_CMLPPDLL are triggered if ADD_CMLPP is set, but appear to have no bearing on the code.
CML_DEBUG	Debug output for CML.
QUEUE_QINT	Sets which queuing functions to be used in search algorithms.
MOL_PRESENT	Sets which type of MOL file is being read (query, react or CPSS). By default none of these are set.
EXTR_FLAGS	Defines how errors in MOL files are to be reported.

EXTR_MASK	Defines how errors in MOL files are to be reported.
ENTITY_REFS_IN_XML_MESSAGES	Always set on.
RESET_EDGE_FORBIDDEN_MASK	Unknown purpose in BNS code. Set and never changed in files **ichi_bns.c** and **ichirvrs.h**.
FIX_CPOINT_BOND_CAP	Bug fix in file **ichi_bns.c**.
CHECK_TACN	File **ichi_bns.c**. Coment 'prohibits replacing (-) on N with H unless H can be moved to N'. Presumed bug fix.
ALLOW_ONLY_SIMPLE_ALT_PATH	File **ichi_bns.c**. Comment 'allow alt. path to contain same bond 2 times (in opposite directions)'.
BNS_MARK_ONLY_BLOCKS	File **ichi_bns.c**. Comment 'find only blocks, do not search for ring systems'.
KEEP_METAL_EDGE_FLOW	Defined so that the function *ForbidMetalCarbonEdges* is never called.
ALWAYS_SET_STEREO_PARITY	It appears that this was used in testing to set the stereo parity.
MAX_LOCAL_TGNUM	Always set to zero and never used.
FIX_BASE26_ENC_BUG	Appears to be a bug fix that is never used.
COUNT_ALL_NOT_DERIV	Always set on. Looks like a bug fix.
ADD_CAPACITY_RADICAL	Appears as a bug fix in file **ichi_bns.h**.
RI_ERR_ALLOC	Determines how errors are dealt with. Mainly refers to input errors, but is also used in BNS.
ALL_NONMETAL_Z	Used to determine which atoms are allowed as the central atom of a tautomeric system (function *is_Z_atom*). Always set to restrict list to C, N, S, P, Sb, As, Se, Te, Br, Cl, I, and to exclude B, O, Si, Ge, F, At. (Technical Manual Table 6)
ALLOW_METAL_BOND_ZERO	Always set to allow zero bonds to metals.
INIT_METAL_BOND_ZERO	Always set so that metal atoms take their prescribed valence and not zero.
TAUT_OTHER	Seems to be always set. Indicates that tautomeric ring systems should be found.
INCLUDE_NORMALIZATION_ENTRY_POINT	Comment in code is 'disabled extra external calls to InChI algorithm', and it is always disabled.
INCHI_CANON_USE_HASH	Is not set. Affects hashing in file **ichican2.c**.
FIX_ADD_PROTON_FOR_ADP	Never used.
MOVE_CHARGES_FROM_HETEREO_TO_METAL	Never used.

FIX_P_IV_Plus_O_Minus	Fix always used.
INCHI_ZFRAG	Never used. Appears to add dummy atom types for bond types.
OLD_ITEM_DISCOVERY	Never set. Presumably the code protected by the definition is not required any more.
REMOVE_CUT_DERIV	Always set on. Code comment is 'remove disconnected derivatizing agents'.
DISPLAY_ORIG_AT_NUMBERS	Always set to display original atom numbers.
SELF_TEST	Appears to protect unused testing code.

The file **mode.h** defines some options grouped by type and generally with individual comments. Definitions within the groups headed 'bug fixes in v1.00', 'bug fixes in post-v1.00', 'bug fixes in post-v1.02b', 'additions to v1.00', 'Normalization settings', 'stereo', 'added tautomeric structures', 'define canonicalization modes', 'questionable behavior', 'canonicalization settings I', 'canonicalization settings II' and 'torture test' are not further detailed here because they are not likely to be changed for future builds.

Some other definitions are well commented in **mode.h**, and are not further detailed here.

Some options are compiler dependent (e.g. Borland) and are not listed here.

Note that several blocks of code are protected by '#if 0' directives, and so will never be compiled. The code could be somewhat simplified by removing conditional compilations which will never occur given the hard definitions in the code.

4 Command Line Parameters

Unless otherwise stated no specific command line parameters have been specified when describing the code functions.

Note that some of the command line parameters have no effect unless the compilation option ENABLE_ENGINEERING_OPTIONS is specified. These options appear to be have been included for debugging purposes during code development and are as follows:

ACIDTAUT:	ALT	ANNPLAIN	ANNXML
AUXFULL	AUXINFO:	AUXMAX	AUXMIN
CMP	CMPNONISO	COMPRESS	DCR
DCR	DDSRC	DISCONMETAL:	DISCONMETALCHKVAL:
DISCONSALT:	DoDRV	DoneOnly	DoR2C
DSB	EQU	EQUNONISO	EXACT
FB2OFF	FBOFF	FixRad	FixSp3bugOFF
FNUDOFF	FULL	InChI2InChI	KeepBalanceP
MERGE	MERGESALTTG:	MIN	MOLFILENUMBER
MOVEPOS:	NoADP	NOHDR	NOUUSB
NOUUSC	NoVarH	O:	ONLYEXACT
ONLYFIXEDH	ONLYNONISO	ONLYRECMET	ONLYRECSALT
OP:	OutputANNPLAIN	OutputANNXML	OutputPLAIN
OutputXML	PGO	PLAIN	PW
RECONMETAL:	RSB:	SASXYZOFF	SCT
SDFID	SdfSplit	SplitInChI	SPXYZOFF
TAUT	UNCHARGEDACIDS:	XML	

5 API Functions

The InChI API is described in IUPAC (2010). The functions can be called as described in that document. The following outline C++ code has been used in running the test examples referred to in the current document.

```
// declare input structures
inchi_Input inp;
inp.num_atoms = nAtoms;

// allocate and initialise memory
inp.atom = (struct tagInchiAtom*)malloc(inp.num_atoms * sizeof inchi_Atom);
memset(inp.atom, 0, inp.num_atoms * sizeof inchi_Atom);
inp.num_stereo0D = 0;
inp.stereo0D = NULL;

// set command line options
inp.szOptions = "/FIXEDH";

// set up connection table in structure out
(…)

// generate InChI
inchi_Output out;
int ret = GetINCHI(&inp, &out);

// generate InChI key
char cKey[50];
GetStdINCHIKeyFromStdINCHI(out.szInChI, cKey);

// tidy up memory
free(inp.atom);
FreeINCHI(&out);
```

6 Reading MOL Files

MOL files are read and converted to internal InChI connection tables by the function *MolfileToInchi_Atom*. Function *e_read_sdfile_segment* is used to read the MOL file and put data into an intermediate structure, which is then converted to an InChI inchi_Atom structure by the function *mol_to_inchi_Atom*. The function *mol_to_inchi_Atom_xyz* looks at the MOL file co-ordinates to determine whether the structure is zero-, two- or three-dimensional.

7 Production of InChI

The InChI identifier and auxiliary information (if required) are constructed in the function *FillOutINChI*. The function *SortAndPrintINChI* is used to sort the separate InChI strings for components of the overall structure.

8 Normalising Structures

Section IVb of the InChI Technical Manual discusses the steps that are taken to normalise chemical structures for the purposes of InChI so that in most cases equivalent structures produce the same InChI. Normalisation is divided into several interconnected processes, some of which are deeply intertwined within the code.

The following sections point to where each step of normalisation is carried out.

8.1 Changes to the Structure Drawing (Technical Manual – Step 1)

Function *fix_odd_things* is used to change the input connection table as specified in Technical Manual Table 1. Function *remove_ion_pairs* is used to detect and alter ion pairs as specified in Technical Manual Table 2.

8.2 Disconnection of Salts (Technical Manual – Step 2)

Function *DisconnectSalts* is used to disconnect metal and ammonium salts.

8.3 Disconnection of Metals (Technical Manual – Step 3)

Function *DisconnectMetals* is used to disconnect metals that are not considered to be acid salts.

8.4 Elimination of Radicals (Technical Manual – Step 4)

Bonds are fixed according to the rules of Technical Manual Table 1 in Function *SetForbiddenEdges*.

Radical cancellation as shown in the Technical Manual (Figures 9a and 9b) does not appear in the library. This is presumably explained by the comment in the manual that this is considered to be an extension for MOL files only and is not handled by the main library code, but as specific code for reading MOL files.

8.5 Removal of Protons from Charged Heteroatoms (Technical Manual – Step 5.1)

Function *SimpleRemoveHplusNPO* is used to remove the protons followed by *HardRemoveHplusNP* to remove the protons where there are alternating bonds. The example of “hard” proton removal shown in Figure 10 in the Technical Manual is dealt with by the function *RemoveNPProtonsAndAcidCharges*. The functions *SimpleRemoveAcidicProtons* and *HardRemoveAcidicProtons* perform similar functions for protons on acids.

8.6 Removal of Protons from Neutral Heteroatoms (Technical Manual – Step 5.2)

Removal of protons is done by the function *SimpleRemoveAcidicProtons*.

8.7 Addition of Protons to Reduce Negative Charge (Technical Manual – Step 5.3)

Functions *SimpleAddAcidicProtons* and *HardAddAcidicProtons* are used to add protons.

8.8 Detection of Tautomerism (Technical Manual – Step 6.1)

The process of tautomer detection makes use of algorithms for Balanced Network Searches (BNS) described by Kocay & Stone (1993) and Kocay & Stone (1995) (as referenced in the code).

The algorithm in the function *BalancedNetworkSearch* is run iteratively from the function *RunBalancedNetworkSearch* to detect alternate paths and, if any are detected, to mark the bonds in those paths as alternate. The information stored in the BNS structure is used to check that any potential tautomeric structure detected at a later stage is actually tautomeric.

Function *MarkTautomerGroups* controls the detection of tautomers. (See description of *MarkTautomerGroups* below for description of how each type of tautomer is handled.) The method is to find a possible tautomeric group, check that it is actually contains the correct alternating path and then set the required bonds to alternating with the function *SetTautomericBonds*. The functions used to check the alternating path all use the BNS to do the checking.

During the checking of alternating paths the program may add fictitious nodes to the BNS in order to check for alternating bond paths (Tchekhovskoi, 2011). This is done within the function *bSetBnsToCheckAltPath* using *bAddNewVertex*, after which the function *RunBalancedNetworkSearch* is called again to check on any new alternate paths.

8.9 Detection of Movable Positive Charges (Technical Manual – Step 6.2)

Function *HardRemoveHplusNP* is used to move positive charges and then the methods used in Step 6.1 are applied to detect tautomerism. *HardRemoveHplusNP* also uses the BNS to detect alternating paths along which charges can be moved. Note that the Technical Manual Figure 15 fictitious structures are stored in the BNS, not in the connection table.

8.10 Additional Normalisation (Technical Manual – Step 6.3)

Function *MarkChargeGroups* is used to delocalise charges for Step 6.3 case (1).

Function *MergeSaltTautGroups* is used to delocalise H atoms and negative charges for Step 6.3 case (5).

9 Handling Isotopic H and Heavy Atom Isotopes

As described in the Technical Manual Section IVc, heavy atom isotopes are handled as an atom property, and are used to colour atoms at the appropriate canonicalisation levels (4, 6 and 8). The function *SetAtomAndBondProperties* sets up num_H and the num_iso_H array for each atom where there are H atoms attached to the heavy atom. num_H stores the total number of H atoms attached to the heavy atom and num_iso_H stores the counts for the three isotopes of hydrogen. The function *remove_terminal_HDT* is used to remove H atoms included as specific atoms in the CT and add them to the H atom counts for the heavy atom. Bridging H atoms are not removed by this function.

10 Handling Stereochemistry

Stereochemical information can be made available to InChI in one of two ways. Either directly specified in the MOL file (so-called 0D parity – see the file **inchi_api.h** for code comments) or derived from the atom co-ordinates. 0D parities are used if no co-ordinates are specified.

The input co-ordinates supplied for the atoms are used to decide on stereochemistry. If no co-ordinates are supplied, stereochemistry cannot be decided for bond-centred stereochemistry. Up/down bonds are only used to decide atom-centred stereochemistry (including allenes).

Function *set_stereo_parity* is called to assign stereo information to atoms and bonds from the co-ordinates.

The code treats allenes and cumulenes by analogy with atom stereocentres and bond stereochemistry. Chains containing an even number of cumulated double bonds are considered as allenes and chains containing an odd number of cumulated double bonds are considered as cumulenes.

The function *set_stereo_bonds_parity* detects allenes and cumulenes (Technical Manual Figure 26) and checks that the atoms involved are suitable for bond stereochemistry according to the rules in the Technical Manual Figure 21 – 23 and Table 7. The function also deduces the parity for double bonds (and cumulenes).

Function *bCanInpAtomBeAStereoCenter* detects whether a central atom is a potential stereocentre according to the rules in the Technical Manual Table 8, applies the rules for correct 2D drawing given in Technical Manual Tables 9 and 10, and assigns the parity of the centre if possible. The function *set_stereo_atom_parity* is used to calculate the parity for either 2D or 3D structures. In 3D structures the given z co-ordinates are used; in 2D structures the up/down nature of the bonds is used to calculate the z co-ordinates. A stereo centre may have three or four neighbours; for centres with three neighbours a dummy fourth neighbour is created to represent H (or a lone pair). The function *Get2DtetrahedralAmbiguity* is used to decide on the rules in Tables 9 and 10. The code also checks that the atoms surrounding the stereo centre have co-ordinates that are sufficiently distinct to allow meaningful determination of stereochemistry using function *triple_prod_and_min_abs_sine2* which also determines whether the parity is odd or even.

Stereochemistry is assigned only tentatively at this point: it may turn out that neighbouring atoms to the potential stereo centre or bond are in fact equivalent.

After the canonicalisation step the function *UnmarkNonStereo* is called to decide whether the atom or bonds marked as stereocentres really do have stereochemistry. Atoms are considered as equivalent if they have the same canonical rank.

In function *Canon_INChI3* a test is carried out by inverting all the stereocentres (using function *InvertStereo*) and testing using function *CopyLinearCTStereoToINChIStereo* to see if the structure has overall stereochemistry and needs information in the '/m' section of InChI (Technical Manual Figure 25).

11 Canonicalisation of Structures

Function *GetBaseCanonRanking* is used to control structure canonicalisation according to the flowchart shown in the Technical Manual Figure 30 which shows how the various layers of InChI are related. Each InChI layer corresponds to a stage in the canonicalisation process.

The basic process is to assign colours to each atom according to the information known at each layer and then use that information to partition or re-partition the structure. The function *CanonGraph* implements the canonicalisation algorithm of McKay (1981), although probably in a modified form. The naming of variables in the code closely follows that of McKay (pages 68 to 72), although there are some changes.

For a general overview of canonicalisation processes see Augeri (2008); for a more specific overview of how canonicalisation works in InChI see Apodaca (2006).

Each stage in the canonicalisation process follows a similar path (although much modified in most stages). The first step is to use the function *SetInitialRanks2*, which sets atom ranks (colours) according to the information available at the current layer about each atom. The function *DifferentiateRanks2* is then called to refine the current partitions according to the available colouring information. Finally the function *CanonGraph* is called to refine canonical ordering according to the McKay algorithm.

The various canonicalisation stages are only called if there is information in the structure that will affect the partitioning. For instance if there is no isotope information specified in the structure, the isotope stage is not called.

The following table shows the canonicalisation stages and the connection tables input and output at each stage.

Stage	Stage Functionality	Input CT	Output CT
1	Initial graph with colours based on atom type. Always required.	None	Ct_NoH
2	Only required if Stage 1 produces non-equivalent atoms with the same rank	None	Ct_NoH
3	Equivalent atoms differ by fixed H count.	Ct_NoH	Ct_NoTautH
4	Equivalent atoms differ by isotope (fixed H or heavy atom) in a non-tautomeric structure.	Ct_NoTautH	Ct_NoTautHIso
5	Tautomeric structure.	Ct_NoTautH	Ct_Base
6	Tautomeric structure with isotopes.	Ct_Base	Ct_BaseIso
7	Only used if /FIXEDH is given as a command line	Ct_NoTautH	Ct_FixH

	option.		
8	Only used if /FIXEDH is given as a command line option.	Ct_FixH	Ct_FixHIso

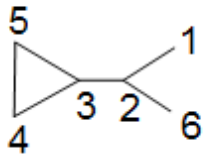
The stage numbers indicate how the function *CanonGraph* is called for that stage. The call at each stage takes the form *CanonGraph0<n>*. In fact these calls are always to the same function, the final digit is simply to help in testing and debugging.

Note that stereochemistry is not dealt with as part of the general canonicalisation process but by a separate process called from *Canon_INChI3* (see Section 10 above).

11.1 Connection Tables in Canonicalisation (Level 1)

The structure ConTable is used to store connection tables during the canonicalisation step. Function *CtPartFill* is used to populate the connection table. Functions *CtPartCompare* and *CtFullCompare* are used to compare the connection tables.

The graph of the connection table is stored in the linear array Ctbl within the ConTable structure. As an example of how the Ctbl array is used we take isopropylcyclopropane. The array contains one entry for each atom and one entry for each bond. Since the structure has 6 atoms and 6 bonds the array has 12 entries.



The following table shows how the structure is stored using an arbitrary numbering scheme.

Atom	Bonds to Atoms	Bonds to Earlier Atoms
1	2	-
2	1, 3, 6	1
3	2, 4, 5	2
4	3, 5	3
5	3, 4	3, 4
6	2	2

The information stored for each atom is atom number followed by bonds to lower numbered atoms sorted in ascending order. In this case we have:

1, 2 1, 3 2, 4 3, 5 3 4, 6 2

giving an array populated as follows:

1 2 1 3 2 4 3 5 3 4 6 2

This is unambiguous because the first time an atom is mentioned it signals the start of the connections for that atom.

The algorithm is looking for a canonical connection table from all the possible numberings of the atoms in the structure.

As an example of how a canonical description of the structure is obtained, we take methylcyclopropane as an example and list all 24 of the atom numbering permutations.

01: 1 2 1 3 1 2 4 1	13: 1 2 1 3 1 2 4 3
02: 1 2 1 3 1 4 1 2	14: 1 2 3 1 2 4 1 3
03: 1 2 1 3 1 2 4 1 = 01	15: 1 2 1 3 1 2 4 3 = 13
04: 1 2 1 3 1 4 1 3	16: 1 2 3 1 2 4 2 3
05: 1 2 1 3 1 4 1 2 = 02	17: 1 2 3 1 2 4 1 3 = 14
06: 1 2 1 3 1 4 1 3 = 04	18: 1 2 3 1 2 4 2 3 = 16
07: 1 2 1 3 2 4 1 2	19: 1 2 1 3 4 1 2 3
08: 1 2 1 3 1 2 4 2	20: 1 2 3 1 4 1 2 3
09: 1 2 1 3 2 4 2 3	21: 1 2 1 3 4 1 2 3 = 19
10: 1 2 1 3 1 2 4 2 = 08	22: 1 2 3 2 4 1 2 3
11: 1 2 1 3 2 4 2 3 = 09	23: 1 2 3 1 4 1 2 3 = 20
12: 1 2 1 3 2 4 1 2 = 07	24: 1 2 3 2 4 1 2 3 = 22

Duplications are indicated to the right of the list. The duplicates occur because the unsubstituted atoms in the ring are equivalent and their numbers can be interchanged. The structure produces InChI:

InChI=1S/C4H8/c1-4-2-3-4/h4H,2-3H2,1H3

that is equivalent to (22) in the list.

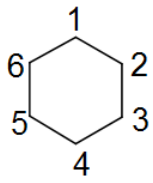
The InChI algorithm puts atoms of the same type in order of increasing valency. *CanonGraph* uses the function *PartitionColorVertex* to sort atoms by their neighbour lists. In general partitions that have the lowest neighbour numbers are preferred. This is significant for structures such as 2,3-dimethylbutane where the methyl groups are all symmetrically equivalent, but once one of them has been given a lower rank (that is it has been arbitrarily coloured differently to the others) they split into two pairs and are no longer equivalent. The arbitrary colouring to break the symmetry would not have been necessary had one of the atoms been different in some other way (e.g. by being of a different atomic type). If we label the methyl groups attached to one of the 3-valent atoms as A and the methyl groups attached to the other 3-valent atom as B, then we have the following possible orders starting with A for the methyl atoms:

AABB
ABAB
ABBA

In order to canonicalise the structure correctly the same order must always be chosen.

11.2 Symmetry in Canonicalisation

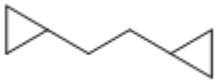
When *CanonGraph* finds two equivalent permutations for the atoms of a structure it uses the information to deduce symmetry. The function *TranspositionGetMcrAndFixSetAndUnorderedPartition* is used to set up the Omega and Phi bitmaps and the theta_from_gamma array with symmetry information. Note that most of the variable names used in the code are those specified by McKay, although there may be some differences in the exact usage of some variables. The theta_from_gamma array is later combined into the total symmetry array theta using the function *UnorderedPartitionJoin*. Theta is then used to prune the tree by deciding which descriptions need not be followed up.



As an example we take cyclohexane. Initially there is no reason to suppose any symmetry, so all the atoms are potentially different starting points for the canonical structure. *CanonGraph* finds an initial partition (rho) (assume the order is 126354) and then backtracks to find another equivalent partition. This partition (which will be 162534) has the same atoms in the 1 and 4 positions but the atom in the 2 position has been swapped with the atom in the 6 position. Similarly atom 3 has been swapped with atom 5. Since the function *CtFullCompare* has found these partitions to be identical, atom 2 must be equivalent to atom 6 and atom 3 must be equivalent to atom 5. The next partition found (213645) starts with atom 2 and is also equivalent to partition rho. This tells us that atoms 1 and 2 are symmetrically equivalent (also atoms 3 and 6 are equivalent, and atoms 4 and 5 are equivalent). This new symmetry information can be combined with the earlier information using the function *GetUnorderedPartitionMcrNode* to deduce that all 6 atoms are equivalent, and that it is not necessary to generate partitions starting with atoms 3 to 6. Hence the search tree has been pruned considerably.

11.3 'Pathological' Structures (Level 2)

A comment in the code for the function *FixCanonEquivalenceInfo* points out that the following structure requires an extra pass through *CanonGraph*.

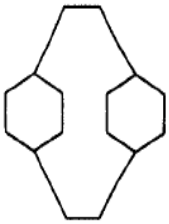


This is because the McKay algorithm does not assign the two types of 2-valent atoms to separate partitions at any stage of the canonicalisation process. It does however put them into symmetry

equivalent sets in theta and InChI uses this to generate the array nSymmRank in code following the label exit_function in *CanonGraph*. The symmetry information is passed back to *GetBaseCanonRanking* for further partitioning using the function *FixCanonEquivalenceInfo*, and if the partitioning has not been completed according to the requirements of InChI, a further call to *CanonGraph* is made.

On return from the first call to *CanonGraph* for the above example all the 2-valent atoms are given rank 6 (the 3-valent atoms have rank 8), although nSymmRank correctly identifies two separate groups of equivalent atoms (four ring atoms and two chain atoms). This information is used to re-set the partition information for the second call to *CanonGraph*.

It is interesting to note that the following structure, which is often given as a counterexample for the Morgan algorithm, also needs a second pass through McKay in the InChI code.



It appears that although McKay's algorithm produces a canonical numbering of the structure, this is not exactly what InChI requires. The second pass is to make sure that atoms are always partitioned into separate equivalence classes as InChI requires. The author of the software (Tchekhovskoi, 2011) has commented as follows on this point:

> Sometimes the canonicalization makes canonical numbers within a set of equivalent vertices non-contiguous, that is, for example, even though vertices A, B, C are equivalent, their canonical numbers arranged in ascending order would not necessarily be n, n+1, n+2. After the canonical partition has been found by the McKay's algorithm, there is a certain degree of freedom in assigning canonical numbers left: one may assign new colors to each group of equivalent vertices and rerun the canonicalization. This would keep the found sets of equivalent vertices unchanged, but their canonical numbers might change. The newly assigned by InChI colors are in order of the smallest canonical number within each set of equivalent atoms, mcr(set). The difference between the new colors are chosen to be equal to the numbers of vertices in the subsequent sets of equivalent vertices. For example, if the found sets of equivalent vertices are A, B, C in this order, that is,
>
> mcr(A) < mcr(B) < mcr(C)
>
> then the new

color(A)=|A|

color(B)=|A|+|B|

color(C)=|A|+|B|+|C|

The new canonical numbers inside A will be 1-- color(A), inside B -- color(A) +1..color(B), inside C -- color(B)+1..color(C). This approach is used in InChI to make canonical numbers within each set of the equivalent vertices contiguous. This property is used in subsequent calculations.

(*Note: mcr stands for minimal class representative.*)

In the example above there are three types of atom: 3-valent atoms (four atoms), 2-valent atoms in 6-membered rings (eight atoms) and other 2-valent atoms (four atoms). InChI imposes the requirement that symmetry equivalent atoms should be adjacent in the ordered atom list. McKay does not impose this restriction. After the first pass through the algorithm the atoms appear in the following order (using 3, 6 and X to represent the three types of atom):

666X6XXX66663333

InChI imposes the extra requirement by colouring atoms in the different symmetry sets found by the first pass through *CanonGraph* and, if necessary, calling *CanonGraph* again to re-canonicalise the structure. After the second pass for this example the order is:

66666666XXXX3333

and the InChI requirement is now met.

11.4 Fixed H Layer (Level 3)

The next level for canonicalisation takes fixed H atoms into account. Amongst other things this takes localised unsaturation into account, and allows InChI description to ignore bond types. This is implemented in the code by a further call to *CanonGraph* after the graph itself has been canonicalised. Before this call in the function *GetBaseCanonRanking* there is some code to set up H counts and to re-partition on the basis of this information. The connection table produced by the earlier calls to *CanonGraph* is passed to the new call for comparison purposes.

11.5 Isotope Layer (Level 4)

The code for the isotope layer is very similar in form to the Fixed H Layer. Isotope sort information is set up and then the structure is re-partitioned before another call to *CanonGraph*. This pass through *CanonGraph* is called when there is either heavy atom or fixed hydrogen isotope information. It is not called if there are tautomeric groups in the structure (see Levels 5 and 6 below).

11.6 Normalisation (Mobile H) Layer (Level 5)

Level 5 uses the function *FillOutAtomInvariant2* to set the information on tautomeric groups. This call is similar to the call in Level 1 above, but with the tautomeric mode set. Then the function *SetInitialRanks2* is called as at the other levels. In the levels dealing with tautomerism the function *DifferentiateRanks4* is called to refine the partitioning information instead of *DifferentiateRanks2* used in the earlier layers. The two functions differ in the way that neighbour lists are used to sort the atom list. *CanonGraph* is then called to produce the partition for this level.

11.7 Mobile H with Isotopes Layer (Level 6)

This level follows a similar path to Level 4 with functions *SetInitialRanks2* and *DifferentiateRanks2* being called to set up partitioning before the call to *CanonGraph*.

11.8 Fixed H Layers (Levels 7 and 8)

These two layers are only used if the user has specified command line parameters requesting them and the code has been built to allow 'Engineering Options' (see section 4).

12 Hashing Algorithms and Key Generation

The code uses the SHA-256 standard (published by NIST in 2002) to hash the major and minor strings. (Devine, 2006). According to the code comments this is freely distributed software (**sha2.c** and **sha2.h**).

Function *GetINCHIKeyFromINCHI* is used to control the generation of the key. The function first checks for a valid input InChI string and then inserts information in the key in the following order:

Major Block (In the example: C3H4FI/c1-2-3(4)5/h2H,1H3 hashes to DKEPSVLMHRNWRG)
Minor Block – the stereochemistry (In the example: /b3-2+ hashes to NSCUHMNN)
Standard Flag (In the example: S)
Version (Hard coded: A)
Separator (-)
Protonation Flag (In the example: N because there is no protonation in this structure)
Terminator (ASCII 0)

InChI=1S/C3H4FI/c1-2-3(4)5/h2H,1H3/b3-2+
DKEPSVLMHRNWRG-NSCUHMNNSA-N

13 Overall Control Functions

The following sections describe some of the most significant functions in the InChI code and give a summary of the flow of control.

13.1 GetStdINCHI and GetINCHI

Wrapper functions for obtaining InChI representations.

For definitions of the function parameters structures inchi_Input and inchi_Output see **inchi_api.h**.

13.2 GetINCHI1

Main calling function for converting CT to InChI.

1. Call *parse_options_string* to parse command line options and then call *ReadCommandLineParms* to set up the global variables for the given options.
2. Parameter bStdFormat ensures that standard command line options are set regardless of options set on command line. For bit values see **ichi.h**.
3. Call *ExtractOneStructure* to extract CT data from input structure ip into internal structure orig_inp_data.
4. Call *ProcessOneStructure* to process to InChI.
5. Outputs as required.
6. Tidy up.

13.3 ExtractOneStructure

1. Call *SetAtomProperties* and *SetBondProperties* for each atom.
2. Call *SetAtomAndBondProperties* for each atom.
3. Call *SetNumImplicitH* to set up implicit H counts.
4. Call *Extract0DParities* to set up stereo parities input from the MOL file.
5. Log any errors in the structure using *TreatReadTheStructureErrors*.

14 Basic Connection Table Functions

14.1 SetAtomProperties

Set up the properties for an atom in at from ati. (Atom type, number, charge, radical, co-ordinates)

14.2 SetBondProperties

Set up the properties for a bond in at from ati and check for consistency. (Bond order, stereo, neighbouring atom)

14.3 SetAtomAndBondProperties

Set valency and atom number. Replace explicit H with isotopic counts. Set relative mass for isotopes.

14.4 SetNumImplicitH

Loop over all the atoms to set up explicit H count for non-metallic atoms using call to *get_num_H*.

14.5 get_num_H

Calculate number of H atoms given the valency of the atom in the input structure and the standard list of valencies allowed by InChI. Code to deal with radicals and special instances of N and S.

14.6 Extract0DParities

Set up stereo parities as passed in from the MOL file.

14.7 ProcessOneStructure

1. Call *CreateOneStructureINChI* to process the structure.
2. Call *SortAndPrintINChI* to sort the component InChI's.

14.8 CreateOneStructureINChI

1. Call *PreprocessOneStructure* to disconnect bonds that do not obey the InChI rules (e.g. those involving metals), alter other structural features that do not obey the InChI rules and determine the number of components in the structure. This function orders the components according to the number of atoms they contain. The final sorting can only be done after the InChI for each component has been determined
2. Main component processing cycle – loop for each component
3. Call *GetOneComponent* to extract CT for this component
4. Call *CreateOneComponentINChI* to get InChI for this component

14.9 PreprocessOneStructure

Note that many of the activities of this function may result in the production of error or warning messages via *AddMOLfileError*. These are not generally detailed individually.

1. Call *DuplicateOrigAtom* to obtain working structure (prep_inp_data).
2. Call *fix_odd_things* to put structure into standard InChI form.
3. Code for calls to *FixAdjacentRadicals* is effectively commented out by definition of FIX_ADJ_RAD.
4. Call *DisconnectSalts* to disconnect metal and ammonium salts.
5. Call *bMayDisconnectMetals* to check if there are any metal bonds that still need disconnecting.
6. If salts have been disconnected, use *ReconcileAllCmlBondParities* to reconcile bond parities.
7. Call *MarkDisconnectedComponents* to identify the components and mark the component number of each atom.
8. Call *bNumHeterAtomHasIsotopicH* to count the number of implicit and explicit isotopic H atoms and set global flags.
9. If call to *bCheckUnusualValences* detects unusual valency, set flag. There is a comment in the code that this 'should be called before metal disconnection'; presumably the call to *bMayDisconnectMetals* above did not do the actual disconnection but only marked bonds for disconnection.
10. If bonds need to be disconnected in a co-ordination compound, *MarkDisconnectedComponents* is called again to re-assign atoms to components.
11. *DisconnectMetals* is called to do the disconnection.
12. Reset bond parities for disconnected structure.
13. Call *ReconcileAllCmlBondParities* to reconcile parities that might have changed as a result of metal disconnection.

14.10 DuplicateOrigAtom

Copy atom properties.

14.11 CreateOneComponentINChI

1. Call to *Create_INChI* to create InChI for component.
2. Call to *SetConnectedComponentNumber* to set component numbers of atoms in current component.
3. Tidy up and error processing.

14.12 Create_INChI

Main chemical and canonical ranking function.

1. Call *remove_terminal_HDT* to remove explicit H atoms.
2. Call *MarkRingSystemsInp* to mark which ring system each atom belongs to.
3. Call *mark_alt_bonds_and_taut_groups* to perform main normalisation process.
4. Call *inp2spATOM* to set up data for examining tautomerism.
5. Call *set_stereo_parity* to set stereo parities from 3D co-ordinates.
6. Call *GetBaseCanonRanking* to do canonicalisation.
7. Call *set_stereo_parity* to set stereo parities.
8. Call *set_atom_iso_sort_keys* to set up isotopic sort keys.
9. Count tautomer groups with *CountTautomerGroups* and set up tautomer structures.
10. Call to *FillOutINChI* controls the production of the InChI string.

15 Structure Standardisation Functions

15.1 fix_odd_things

Loop over all atoms to fix structure drawing. The code contains extensive comments referring to Table 1 in the Technical Manual.

15.2 DisconnectSalts

1. Use *bIsAmmoniumSalt* to detect ammonium salts, and *DisconnectAmmoniumSalt* to disconnect if discovered.
2. Use *bIsMetalSalt* to detect metal salts, and *DisconnectMetalSalt* to disconnect if discovered.

15.3 bIsMetalToDisconnect

Checking if metal atom still needs disconnecting.

15.4 bIsAmmoniumSalt

Implementation of ammonium salt detection rules (Technical Manual page 21).

15.5 DisconnectAmmoniumSalt

Performs disconnection of ammonium salt according to rules (Technical Manual page 21).

15.6 bIsMetalSalt

Implementation of metal salt detection rules (Technical Manual page 21).

15.7 DisconnectMetalSalt

Performs disconnection of metal salt according to rules (Technical Manual page 21).

15.8 bMayDisconnectMetals

Uses *bIsMetalToDisconnect* to detect metals that still need disconnecting (rule on page 22 of Technical Manual).

15.9 MarkDisconnectedComponents

1. Initial loop to mark atoms in components with component number.
2. Count the number of atoms in each component.
3. Sort the components to put the largest first.

4. Renumber the components and adjust the component number for each atom.

15.10 bNumHeterAtomHasIsotopicH

For common non-metals, count the number of implicit and explicit isotopic H atoms, taking charge and radicalisation into account.

15.11 bCheckUnusualValences

Call *detect_unusual_el_valence* for each atom and add error message if unusual valency found.

15.12 detect_unusual_el_valence

Loop through known valencies for atom type to detect unusual valency. Return non-zero if valency is unusual.

15.13 DisconnectMetals

1. Check for any metal bonds that need breaking.
2. Replace implicit H on metals with explicit H.
3. For metals, call *DisconnectOneLigand* to remove the ligand bonds one at a time.
4. Use *DisconnectOneLigand* to disconnect metal-metal bonds.

15.14 DisconnectOneLigand

1. Uses *DisconnectInpAtBond* to disconnect bonds.
2. If the disconnected bond is to a common non-metal (excluding C), adjust the charge (if possible) of the ligand atom to make the valency of the atom “correct”, and adjust the charge on the metal to compensate for this.

15.15 DisconnectInpAtBond

Uses *RemoveInpAtBond* to remove bond from CT.

15.16 RemoveInpAtBond

Remove bond from atom’s bond array and adjust atom and bond stereo parities if necessary.

15.17 ReconcileAllCmlBondParities

Reconciliation of bond parities.

15.18 remove_terminal_HDT

Remove explicit H atoms from connection table and store them as attributes of the heavy atoms. Make allowances for stereochemical implications of removed atoms. There are some exceptional H atoms that remain explicit (for instance H-H).

15.19 MarkRingSystemsInp

Sets nRingSystem for each atom. Note that the ring systems numbers start at one and that non-ring atoms are assigned to a ring containing a single atom. The algorithm distinguishes ring systems, so fused rings are allocated to the same system. Spiro-ring systems count as fused.

16 Tautomerism Functions

16.1 mark_alt_bonds_and_taut_groups

1. Call *AllocateAndInitBnStruct* to set up BNS (balanced network structure).
2. Call *AllocateAndInitBnData* to allocate BNS data memory.
3. Call *SetForbiddenEdges* to protect bonds.
4. Call *BnsAdjustFlowBondsRad* to deal with radicals and aromatic bonds.
5. Call *BnsTestAndMarkAltBonds* to mark bonds that are affected by tautomerism or normalisation.
6. Call *RemoveNPProtonsAndAcidCharges* to remove protons on charged heteroatoms.
7. Call *MarkChargeGroups* to set potentially charged groups.
8. Process charged groups.
9. Code comments 'Main Cycle Begin'.
10. Call *MarkTautomerGroups* to look for potentially tautomeric groups.
11. After the end of the 'main cycle' Call *MergeSaltTautGroups* to combine groups where H atoms and /or negative charges need to be further delocalised.

16.2 AllocateAndInitBnStruct

1. Set up memory for BNS.
2. Convert alternating bonds to single.
3. Add information to BNS about vertices (atoms) and edges (bonds) in network.

16.3 AllocateAndInitBnData

Allocate memory for BNS data.

16.4 SetForbiddenEdges

1. Protect single bonds to acetyl and nitro groups by the rules 1 and 2 in Table 5 of the InChI Technical Manual.
2. Call *fix_special_bonds* to set remainder of fixed bonds specified in Table 5.
3. Call TempFix_NH_NH_Bonds - this function will not be called under the standard compilation settings.

16.5 fix_special_bonds

Apply rules for fixing bonds as specified in Table 5 items 3 to 13.

16.6 TempFix_NH_NH_Bonds

Detect and mark -NH-NH- or -NH-NH3 (sic) bonds in BN_STRUCT.

16.7 BnsAdjustFlowBondsRad

1. Check for valency miscounting due to aromatic bonds.
2. Run *RunBalancedNetworkSearch* to set up BNS data.
3. Run *SetBondsFromBnStructFlow* to set the bonds that need changing as a result of running the BNS algorithm.
4. Run *RestoreBnStructFlow* to reset BNS information.

16.8 RunBalancedNetworkSearch

This seems to be a multi-pass wrapper for *BalancedNetworkSearch*. Calls are made to the function until its return value is non-negative. The code comment is 'Run BNS until no aug pass is found'. The return from the current function is nSumDelta, commented in the code as 'number of eliminated pairs of dots'. This suggests an iterative search for alternate paths.

16.9 SetBondsFromBnStructFlow

Uses the function *SetAtomBondType* to set bond types in connection table from bond types in the BNS structure.

16.10 RestoreBnStructFlow

BNS utility function.

16.11 BnsTestAndMarkAltBonds

Loop over all bonds and use the BNS (balanced network search) method to determine alternating chain or ring systems.

1. Set up for each bond by calling *nMinFlow2Check*, *nMaxFlow2Check* and *nCurFlow2Check* to set up information required for testing sequence of bond types.
2. Call *bNeedToTestTheFlow* to decide if further processing is required.
3. Call *bSetFlowToCheckOneBond* to perform actions concerned with setting up for the BNS algorithm.
4. Call *RunBalancedNetworkSearch* to perform the BNS algorithm.
5. Call *bSetBondsAfterCheckOneBond* to change marker on first bond.
6. Call *SetBondsFromBnStructFlow* to change the remaining bond markers as required by the BNS results.

16.12 nMinFlow2Check

Checking flow for BNS algorithm.

16.13 nMaxFlow2Check

Checking flow for BNS algorithm.

16.14 nCurFlow2Check

Checking flow for BNS algorithm.

16.15 bNeedToTestTheFlow

Checking flow for BNS algorithm.

16.16 bSetFlowToCheckOneBond

Checking flow for BNS algorithm.

16.17 bSetBondsAfterCheckOneBond

Changes the bond type marks on one bond as determined by BNS.

16.18 SetBondsFromBnStructFlow

Change markers on bonds according to BNS.

16.19 RemoveNPProtonsAndAcidCharges

1. Use *SimpleRemoveHplusNPO* to do simple removal of protons from selected hetero atoms. (Technical Manual Step 5.1)
2. Use *HardRemoveHplusNP* to remove protons from hetero atoms in more complicated circumstances – cases where the hetero atom is part of a conjugated system. (Technical Manual Step 5.1)
3. Use *SimpleRemoveAcidicProtons* to remove acidic protons in simple cases. (Technical Manual Step 5.2) This step may need repeating presumably because early steps may reveal a new structure that needs processing.
4. Use *HardRemoveAcidicProtons* to remove acidic protons in more complicated cases. (Technical Manual Step 5.2)
5. Use *SimpleAddAcidicProtons* to add acidic protons in simple cases. (Technical Manual Step 5.3) This step may need repeating for the same reason as with Step 5.2.
6. Use *HardAddAcidicProtons* to add acidic protons in more complicated cases. (Technical Manual Step 5.3)

16.20 SimpleRemoveHplusNPO

Uses *GetAtomChargeType* to determine if H needs to be removed from an atom then calls *AddOrRemoveExplOrImplH* to do the actual removal.

16.21 GetAtomChargeType

Examine the chemistry of charged atoms taking into account the type of the atom and its neighbourhood.

16.22 AddOrRemoveExplOrImplH

If the first parameter (nDelta) is positive, the function simply increments the implicit H count. Otherwise it removes explicit H atoms and increments the atom H (and isotope) counts.

16.23 HardRemoveHplusNP

Remove of protons from tautomeric groups and deal with some cancelling charges.

16.24 SimpleRemoveAcidicProtons

Determine charge type of each atom and remove protons if necessary.

16.25 HardRemoveAcidicProtons

Removes protons and creates tautomeric groups.

16.26 SimpleAddAcidicProtons

Determine charge type of each atom and add protons if necessary.

16.27 HardAddAcidicProtons

Adds protons and creates tautomeric groups.

16.28 MarkChargeGroups

Delocalises charges using *GetChargeType* to get candidate charged atoms for marking.

16.29 MarkTautomerGroups

16.29.1 Handle 1-3 Tautomers

1. Count and list potential end points for tautomers.

2. Check in pBNS that the bonds are not protected (*ALLOWED_EDGE*) and that they are not already tautomeric (bond type obtained from pBNS by *ACTUAL_ORDER*).
3. Check with *FindAccessibleEndPoints* that the end points listed above are suitable.
4. Call *RegisterEndPoints* to add new tautomeric groups or merge the new group with existing tautomeric groups.
5. Call *SetTautomericBonds* to set bonds to tautomeric.

16.29.2 Handle 1,3 Keto-Enol Tautomerism

Similar to previous section with rules amended for keto-enol systems.

16.29.3 Handle 1,5 Tautomerism

Uses *nGet15TautInAltPath* to determine 1,5 tautomeric systems.

16.29.4 Handle 4-Pyridinol Ring Tautomerism

Calls *nGet12TautIn5MembAltRing* to confirm tautomerism.

16.29.5 Handle Pyrazole Tautomerism

Calls *nGet12TautIn5MembAltRing* to confirm tautomerism.

16.29.6 Handle Tropolones

Calls *nGet14TautIn7MembAltRing* or *nGet14TautIn5MembAltRing* to confirm tautomerism.

16.30 FindAccessibleEndPoints

Find pairs of atoms that are end points of alternate paths.

16.31 RegisterEndPoints

Create new tautomeric groups from end points or merge end points into existing tautomeric groups.

16.32 SetTautomericBonds

Change specific bond types (single or double) to tautomeric.

16.33 nGet12TautIn5MembAltRing

One of several functions to find and check ring tautomerism (*nGet14TautIn7MembAltRing*, *nGet14TautIn5MembAltRing*, *nGet12TautIn5MembAltRing*, *nGet15TautIn6MembAltRing*).

Calls *DFS_FindTautInARing* to check tautomerism.

16.34 DFS_FindTautInARing

Calls one of several functions (specified by *CheckDfsRing*) to check the ring. Functions are *Check5MembTautRing*, *Check6MembTautRing*, *Check7MembTautRing*.

16.35 Check5MembTautRing

Also *Check6MembTautRing* and *Check7MembTautRing*.

Code comment: check if a tautomeric 5-member ring (pyrazole derivatives) has been found.

16.36 GetChargeType

Tests atom iat for rules on charging. The atom must be a suitable hetero atom, charged and in a ring that contains at least 5 atoms.

17 Stereochemistry Functions

17.1 set_stereo_parity

Calls *set_stereo_atom_parity* and *set_stereo_bonds_parity* to deduce parities for atoms and bonds respectively.

17.2 set_atom_iso_sort_keys

For each atom use *make_iso_sort_key* to set up the isotopic sort key. For atoms that are not in tautomeric groups this will include the atom isotope and isotopic H counts. For atoms in tautomeric groups, only the atom isotope is set.

17.3 make_iso_sort_key

Sets up sort key with supplied information.

17.4 CountTautomerGroups

1. Loop through tautomeric groups to find maximum number. Count the number of end points for each tautomeric group.
2. Mark tautomeric groups that do not need to be processed (no H atoms or inconsistent number of end points).
3. Re-number the groups and end points.

17.5 inp2spATOM

Copy atom data from inp_at to at.

18 Canonicalisation Functions

18.1 GetBaseCanonRanking

Controlling function for canonicalisation. This function follows the flowchart given in Figure 30 of the InChI Technical Manual.

1. Decide whether the structure is tautomeric and set appropriate flags.
2. Call *CreateNeighList* to set up lists of neighbours for non-tautomeric and tautomeric structures as required. These lists will be used in differentiating atom ranks.
3. Call *FillOutAtomInvariant2*. This sets up the structure pAtomInvariant used in sorting the initial ranks.
4. Call *SetInitialRanks2* to set ranks from atom colours.
5. Call *DifferentiateRanks2* to set ranks from neighbour lists.
6. Calls to *CanonGraph<nn>* functions to refine canonical ordering according to McKay's algorithm. The various *CanonGraph<nn>* functions are all wrappers for *CanonGraph* itself. (Hydrogenless skeleton Ct_NoH)
7. Initial call to *CanonGraph*(01).
8. Call *FixCanonEquivalenceInfo* to determine bChanged.
9. Depending on bChanged, a call may be needed to *CanonGraph*(02). This will happen if atoms that are not symmetrically related are given the same rank. For details see description of *FixCanonEquivalenceInfo*. (Hydrogenless skeleton Ct_NoH)
10. Loop through atoms to see if there are any canonically equivalent atoms that have different fixed H counts.
11. If there are, call *SetInitialRanks2* and *DifferentiateRanks2* to re-set the ranks including fixed H counts.
12. *CanonGraph*(03) is then called with an input CT. The pointer to the input CT was left blank in the earlier calls to *CanonGraph*. Note that on this call to *CanonGraph* the parameter digraph is set to true whereas in the earlier calls it was set to false. The subsequent calls 4, 5, 6 and 8 also set digraph to true. (Immobile H Ct_NoTautH)
13. If there are any specific isotopes set (including specific non-tautomeric H isomers), fill out key information for the atoms in the CT and then call *SetInitialRanks2* and *DifferentiateRanks2* to re-set the ranks to take this information into account.
14. *CanonGraph*(04) is then called to get a new set of rankings taking the isomer information into account. (Isotopic atoms & H Ct_NoTautHIso)
15. Call *FillOutAtomInvariant2* with the bTautGroupsOnly flag set.
16. Call *SetInitialRanks2* to set ranks.
17. Call *DifferentiateRanks4* to set ranks from neighbour lists.

18. Set up extra auxiliary invariant information and call the last two functions again with atom auxiliary information set.
19. *CanonGraph*(05) is then called to get a new set of rankings taking the tautomer information into account. (Mobile H groups Ct_Base)
20. The next section deals with tautomeric groups that contain isotopes (either H or heavy atom).
21. First count the number of mobile isotopic H atoms and the number of non-mobile isotopic atoms.
22. Then call *SetInitialRanks2* and *DifferentiateRanks2* to re-set the ranks including mobile isotopic H counts.
23. Call *CanonGraph*(06) to rank the atoms with the mobile isotope information included. (Isotopic: atoms, immobile H, mobile H groups, atoms that may have exchangeable H Ct_BaseIso)
24. Further calls to *CanonGraph* (07 (Δ(fixed H) Ct_FixH) and 08 (Isotopic: atoms and H Ct_FixHIso)) are in the code, but these can only be called by specifying the compilation option ENABLE_ENGINEERING_OPTIONS and including the command line parameter FIXEDH. These are presumably test modes and not for general use.

18.2 CreateNeighList

Create a neighbour list (a graph) from the lists of connections to each atom. The graph contains a list of pointers to lists of connections for each atom.

18.3 FillOutAtomInvariant2

Sets up information used for initial sorting of atom ranks.

The switch bTautGroupsOnly determines whether to set up complete information or just for tautomeric groups.

1. Count number of C atoms, explicit H (including D and T), and atoms of other types. Also set up an array of the symbols of the other types.
2. Sort the symbol array using the comparison function *CompChemElemLex* and if there are C atoms present, add ‘C’ at the start of the element list. If there are H atoms present add ‘H’ at the end of the list.
3. Set up the atom invariant for each atom.
4. The first entry in the val array is the order of the atom type in the Hill formula.
5. The next entry is the number of connections to the atom.
6. Optionally followed by indicators of hydrogen counts.
7. Optionally followed by indicators of tautomerism.
8. The iso_sort_key is then set up for the atom.
9. Set up tautomeric group information.

18.4 CompChemElemLex

Compares first 2 characters of string.

18.5 SetInitialRanks2

1. Initialise atom numbers.
2. Sort atom numbers using the comparison function *CompAtomInvariants2*. The structure pointed to by pAtomInvariant2 is used for sorting.
3. Loop backward through atoms to assign ranks. All the atoms in a cell are given the rank of the greatest atom index in that cell. The function *CompAtomInvariants2Only* is used to decide if atoms are equivalent.

18.6 CompAtomInvariants2

Use *CompAtomInvariants2Only* to decide on precedence. In the event of a tie, the pointers themselves are used to decide precedence.

18.7 CompAtomInvariants2Only

Compares atoms a1 and a2 according to information in the structure pointed to by the global variable pAtomInvariant2ForSort. This global will have been set up by the calling function prior to starting the comparison.

The order in which the comparison is done is as follows.

1. Loop through the val array of pAtomInvariant2ForSort for the first AT_INV_BREAK1 (the number of atom invariant values that have to be compared) members until a difference is found. Precedence is given to the atom with the greatest val.
2. Precedence is given to the atom with the greater iso_sort_key.
3. Loop through the remaining members of the val array up to the AT_INV_LENGTH (the number of atom invariant values that need to be compared if using the iso sort key) member until a difference is found. Precedence is given to the atom with the greater iso_aux_key.
   Note: the comparison code in this loop seems strange because it only returns if there is no difference in the values; this is in contrast to the logic of the previous loop which returns when there is a difference.

18.8 DifferentiateRanks2

1. Set up the global pointer pn_RankForSort as pnCurrRank for sort comparisons.

2. Use either *tsort* (which is defined as *insertions_sort*) with comparison function *CompRank* or *qsort* with comparison function *CompRanksOrd* to sort the atom list. The sort used depends on the value of bUseAltSort.
3. Iterate through the functions *SortNeighLists2* and *SetNewRanksFromNeighLists* (using comparison function *CompNeighListRanksOrd*) until the ranking orders do not change. We then have the best ranking in pnCurrRank.

18.9 DifferentiateRanks4

Similar to *DifferentiateRanks2*, but with different sorting functions. The sorting function correspond to different levels of call to the McKay canonicalisation algorithm.

18.10 insertions_sort

Looks like a bubble sort. Presumably this sort is preferred to the C *qsort* because the lists to be sorted are either very short or in close to the correct order to start with.

18.11 CompRank

Compares ranks for sorting.

18.12 CompRanksOrd

Compares ranks for sorting and uses pointers in the event of a tie.

18.13 SortNeighLists2

Loops through all atoms to sort the neighbour lists using *insertions_sort_NeighList_AT_NUMBERS* for atoms that have duplicate rank with another atom.

18.14 insertions_sort_NeighList_AT_NUMBERS

Sorts the neighbour list supplied in base. The first entry in the base array is the number of connections. Uses a bubble sort because the list of connections is very short.

18.15 SetNewRanksFromNeighLists

Sorts an atom number list using either *tsort* or *qsort* (as in *DifferentiateRanks2*) with comparison function as supplied by the calling function (comment indicates that the function used is always *CompNeighListRanksOrd*).

18.16 CompNeighListRanksOrd

Comparison function that returns result of *CompNeighListRanks*. If this results in a tie, the return is based on the value of the pointers.

18.17 CanonGraph

This function is the fundamental implementation of the canonicalisation algorithm. According to the code comment “A naive implementation of graph canonical numbering algorithm from "Practical Graph Isomorphism" by Brendan D. McKay, *Congressus Numerantium*, Vol. *30* (**1981**), pp. 45 - 87. Note: Several typos fixed, added chem. struct. Specifics”.

The main result of this function is to deposit the best partition in rho and use it in *exit_function* to renumber the structure in canonical order.

The variable lab is always true in this implementation; the code would be slightly simpler if tests on lab were to be removed along with code that is only executed if lab is false. The variable has been retained for consistency with McKay’s published algorithm (Tchekhovskoi, 2011). The reason that lab is always set is that we want to find the best value for rho and that may be better than zeta. If we only find zeta we might end up with different values depending on the original number scheme. This would still give useful information about symmetry.

The code contains several debug sections with the statement ‘int stop = 1’, presumably to provide a convenient breakpoint. These could be removed from the release version and would result in some code simplification.

There is a consistent ‘off by one’ problem in the code with the index k. This is presumably because the original algorithm is based on arrays starting with one (as in Fortran) instead of zero (as in C). The code could be made slightly simpler and easier to follow by refactoring to the native C style.

18.17.1 Initialisation Section

1. Assign memory to and initialise structures.
2. Use *UnorderedPartitionMakeDiscrete* to initialise theta. theta records which atoms are equivalent by symmetry. This function makes them all different to start with.
3. Call *PartitionIsDiscrete* to determine if pi is already discrete. The structure pi is at this stage the partition handed to *CanonGraph* by the calling function. If the partition is already discrete, no more processing is required here. Examples: methane, methyl chloride.

4. Call *PartitionSatisfiesLemma_2_25* (McKay's lemma 2.25). Note that Lemma 2.25 is only used if dig is false and that this is the only the case for the '01', '02' and '07' calls to *CanonGraph*. McKay says that the Lemma does not hold for graphs with loops. Tchekhovskoi (2011) states

   > "As to the dig variable (digraph, or directed graphs), it should be 1 in case of directed graphs. The presence of the directed graph is determined by the structure of layer(s) optimized, not by the molecular graph. In practice, digraph means that when calculating an equitable partition, for an edge A->B the color of A affects the color of B, but not vice versa. […] This approach is needed to keep the previously optimized layer unchanged while optimizing the next layer."

   Presumably the lemma can be used for layers '01', '02' and '07' because they are not based on earlier information, although it is not clear that this applies to '07', or to '02' either. It is also not clear if removing the Lemma altogether would cause serious inefficiency in canonicalisation. According to Tchekhovskoi's comment above, loops are concerned with the need to maintain a pre-determined order (from an earlier layer) and not with whether or not the chemical structure contains rings. This notation is somewhat confusing for chemists.
5. Call *CtPartClear* to do some initialisation, but this may be unnecessary.

18.17.2 Preliminary Section to Get Initial Partition

6. Start of preliminary section to get what the code comment calls 'the first leaf'. This obtains an initial Minimal Class Representative (mcr) ordering and stores it in rho as the best partition found to date. Subsequent orderings can then be compared to rho, and if better, stored as the new best partition.
7. Get the first multiple atom (vertex) cell from pi using *PartitionGetFirstCell* (information returned in W).
8. Use *CellGetMinNode* to get the first node (atom) in the cell found by (7).
9. Check with *PartitionSatisfiesLemma_2_25* again.
10. Call PartitionColorVertex to reduce the rank of the selected atom and re-partition the current level of pi.
11. Use *CtPartFill* to fill in Lambda using the atom numbers stored in pi for the current partition.
12. Check Lambda against the fixed rho using *CtPartCompare*. Note that the fixed rho does not exist on the first passes through *CanonGraph* (the pointer to fixed rho is NULL for these passes).
13. Use *CtPartCopy* to copy Lambda to pzb_rho (the best CT found so far) and zf_zeta (the first complete CT found).

14. Use *PartitionIsDiscrete* to check if the current level of pi is discrete. If it is we have the first possible value for rho and can leave the first leaf node, otherwise we carry on re-partitioning pi until it is discrete.
15. End of loop to find starting partition.

18.17.3 Set Up For Testing Further Partitions

16. Call *CtPartInfinity* to initialise zf_zeta CT.
17. Use *PartitionCopy* to copy latest partition (i.e. discrete pi) into zeta.
18. If there is a fixed rho call *CtFullCompare* to set up qzb_rho_fix, re-set rho from pi and use *CtCompareLayersGetFirstDiff* to re-set some variables; otherwise just copy the latest level of pi into rho with Partition copy and use *CtPartInfinity* to initialise pzb_rho CT.
19. Use *CellMakeEmpty* to initialise the cell at level k of W and then drop back by one level
20. We then enter a long section of code which is controlled by 'goto's and labels. After the previous step there is an unconditional jump to L13. The code at L13 gets an alternative node at the current level (which was decremented in the previous step) and then proceeds to test the resultant partition against the current rho.

18.17.4 Loop to Find Best Value of Partition rho

21. Section following label L2.
22. Use *PartitionColorVertex* to update partition pi with the rank of the test atom v[k-1] reduced.
23. Again update Lambda using *CtPartFill*.
24. Use *CellMakeEmpty* to initialise the cell at level k of W.
25. Use *CtPartCompare* to compare Lambda and zf_zeta to see if hz_zeta should be set to k. This means that the current CT (Lambda) is as good as the first discovered CT (zf_zeta).
26. Similar comparison on Lambda and pzb_rho_fix to make sure that Lambda is as good as the fixed CT handed down from the previous layer.
27. Code concerned with testing for the best value of rho.
28. There are 3 possibilities at the end of the L2 block. (1) If the current partition is discrete we go directly to the L7 block. (2) This may be a new isomorphism and we re-partition at the current level and loop back to L2. (3) Nothing was found so we drop through to L6 and go back to an earlier level.

18.17.5 Backtrack

29. Section following L6.
30. Decide which level to return to.

31. May go directly to L13 or use *PartitionGetMcrAndFixSet* to get node equivalencies into omega and phi before going to L12. However it is not clear from the test examples used in compiling this document under what circumstances the jump to L12 is performed.

18.17.6 Found a Better rho or an Isomorphism

32. Section following L7.
33. Tests on h_zeta and hz_zeta.
34. The use of labels here is somewhat confusing as the result of the code not being structured according to normal C language standards.
35. If Lambda and zf_zeta compare exactly using *CtFullCompare* we have an isomorphism and use *PartitionGetTransposition* to make gamma (which reflects the symmetry implied by the isomorphism) and then jump to L10. Gamma will be used to determine equivalent nodes in L10.

18.17.7 Deal with Potentially Better Value for rho

36. Section following label L8.
37. This code is reached when a potentially better rho has been reached.
38. Tests for better partitioning (qzb_rho > 1) and that the new partitioning is consistent with the fixed partitioning from the previous layer.

18.17.8 Found a Better rho

39. Section following label L9.
40. Copy the latest partition into rho.
41. Copy the latest CT into pzb_rho.
42. Return to L6 to start backtracking.

18.17.9 Deal With Isomorphism

43. Section following label L10.
44. Call *TranspositionGetMcrAndFixSetAndUnorderedPartition* to get node equivalencies into theta_from_gamma using gamma, omega and phi.
45. Call *UnorderedPartitionJoin* to merge theta_from_gamma into theta. If there is no change in theta, skip to L11 which effectively looks for a new test node.
46. Test with *GetUnorderedPartitionMcrNode* to see if we can skip over L11 and L12.
47. If we are skipping, set level back to h_zeta. (Comment - this code could be better structured in C.)

18.17.10 Backtrack After Isomorphism

48. Section following label L11.
49. Set level to h_rho or h_zeta depending on whether lab is true. Since lab is always true this means that we are looking for the best value of rho.

18.17.11 Prepare to Start from Backtrack

50. Section following label L12.
51. May call *CellIntersectWithSet*, depending on value of e at current level (e is set in section following label L17 but it is not known how that section can be reached – see section 18.17.16 below for L17).

18.17.12 Get Next Node for Testing

52. Section following label L13.
53. Check if we should quit for either user request or time out.
54. Check if the level is 0 and if it is, all possibilities have been examined and we can proceed to the normal exit.
55. Set h_rho and h_zeta. Both variables both seem to correspond to McKay's 'h', and a comment in the code at this point hints that they were introduced to solve some problem that showed up when the algorithm was put into use.
56. Use *CellGetMinNode* to get next node (atom) to try as the earliest cell in the next trial partition. Then drop through to L14 for testing.

18.17.13 Test Potential New Partition

57. Section following label L14.
58. Use calls to *GetUnorderedPartitionMcrNode* to detect if the current node (v[k-1]) and the next test node (tvh) are symmetrically equivalent in theta, and if they are increment the counter (index) through the number of nodes in the current cell.
59. Use *CellGetMinNode* to get the next test node at the current level (v[k-1]).
60. If the nodes that need testing at this level have been exhausted (node = infinity), go to L16 to decrement the level.
61. Test with *GetUnorderedPartitionMcrNode* to see if a symmetry equivalent atom has already been tested, and if it has loop back to L14 to test another atom.

18.17.14 Found a Potential New Partitioning

62. Section following label L15.
63. Update t_Lemma and hz_zeta.
64. Update qzb_rho and hz_rho.

65. Call *UpdateCompareLayers* to re-initialise kLeast_rho.
66. Update qzb_rho_fix and hz_rho_fix.
67. Call *UpdateCompareLayers* to re-initialise kLeast_rho_fix.
68. Unconditionally jump back to L2 to re-partition pi according to new test node.

18.17.15 Backtrack

69. Section following label L16.
70. This point is reached if a new partition was not found in the L14 section.
71. Store information and backtrack by decrementing k.
72. Return to L13 for testing for other partitionings.

18.17.16 Code with Unknown Purpose

73. Section following label L17.
74. This code has not been reached by any of our testing examples. It looks like something designed to reduce the number of permutations that need to be examined.
75. Test on e flag for the current level. If it is not set, call *NodeSetFromVertices* for this level and then do more checking.
76. Set flag in e for current level.
77. Use *CellGetMinNode* to set next node and go to L13 for testing.

18.17.17 Prepare Information for Function Return After Successful Computation of rho

78. Section following exit_function.
79. Test for error on bRhoIsDiscrete flag. This is a check that a partition has actually been found.
80. Use *CtFullCompare* to check pzb_rho_fix against pzb_rho.
81. Fill out nSymmRank with equivalences derived from theta. The symmetry ranks are used in the calling function to determine whether the graph has been partitioned so that atoms which are symmetrically different are not canonically equivalent.
82. Copy information from rho into rank and order arrays for return. Also copy the CT into the return structure.

18.17.18 Tidy Up Before Function Return

83. Section following exit_error frees memory allocated earlier in the function. This section is used after an error, but also drops through from the exit_function section.

18.17.19 UnorderedPartitionMakeDiscrete

Assign each atom to its own partition.

18.18 PartitionIsDiscrete

Decide if every atom in the input partition has its own rank.

18.19 PartitionSatisfiesLemma_2_25

McKay's lemma 2.25. Counts partition size (i.e. number of partitions) and number of non-trivial partitions (i.e. partitions with more than one atom), and then does the specified lemma test.

18.20 PartitionGetFirstCell

Gets the next cell to be partitioned.

18.21 CellGetMinNode

Gets the node within a cell that is to be used for the next partitioning and returns it as the value of the function.

18.22 PartitionColorVertex

Re-colours the node (vertex) given in the parameter list and then re-partitions the graph given the new colouring. First the rank of the nominated vertex is reduced and then either *DifferentiateRanks4* or *DifferentiateRanks3* is used to assign ranks to the remaining vertices. The choice of function depends on the value of the digraph parameter.

18.23 CtPartFill

Fills in the part of the connection table that is known from the current partition ranks. This will depend on the distinct ranks within the current partition. The CT is only filled in as far as the ranks are distinct within the partition. For instance if the ranks are 1 2 4 4 5 6, only the first 2 atoms are distinct. The next part of the algorithm will have to assign separate ranks to the 2 atoms with rank 4 before the remainder of the CT can be filled in. The CT is re-numbered according to the ranks specified in the partition.

18.24 CtPartCopy

Copies part of an internal CT structure to another internal CT structure.

18.25 CtPartInfinity

Initialises connection table structure.

18.26 PartitionCopy

Copies partition and adds mask bit to the copied partition.

18.27 CellMakeEmpty

Initialise the cell information for level k of baseW.

18.28 GetUnorderedPartitionMcrNode

Returns the equivalence of node v in the UnorderedPartition – effectively the equivalence (symmetry) value of the atom as currently known.

18.29 UpdateCompareLayers

Re-initialises kLeast entries above specified level (hzz).

18.30 CtPartCompare

There are very many comments in the code of this function. The purpose of the function is to decide if the CTs represented by the partitions are different and, if so, which is the 'better'. The result will be used to decide whether to continue with the current partition in the *CanonGraph* function. The exact nature of the comparison depends on the Digraph variable. This is because in some layers of the canonicalisation a fixed CT from a previous layer will already be available.

18.31 PartitionGetMcrAndFixSet

Sets up the bitmaps supplied as parameters to the function using the supplied partition.

18.32 PartitionGetTransposition

Makes gamma the transposition of pFrom to pTo.

18.33 TranspositionGetMcrAndFixSetAndUnorderedPartition

Sets up the bitmaps and the unordered transposition array supplied as parameters using the supplied transposition array. The unordered transposition array gives the equivalent atoms in the structure deduced from the current partition.

18.34 UnorderedPartitionJoin

Joins two partitions (parameters p1 and p2) so that p2 contains the total atom equivalences deduced so far by the algorithm.

18.35 GetUnorderedPartitionMcrNode

Calls *nGetMcr2* to get the equivalence class for the supplied atom. This function is used in determining if two atoms are symmetrically related.

18.36 nGetMcr2

Works through the equivalence array and returns the number of the atom equivalence set to which the supplied atom belongs.

18.37 FixCanonEquivalenceInfo

The comment in the code gives an example of a pathological structure which leads to non-equivalent atoms having the same rank after the first pass through *CanonGraph*.

In the example all the 2-connected atoms have the same rank, but are not equivalent. The function rebuilds the current rank array to differentiate between the non-symmetrically equivalent atoms.

1. Use *qsort* on the atom number array with the comparison function *CompRanksOrd*. Note that the ranks for sorting are stored in the global pointer pn_RankForSort (set to the nSymmRank array).
2. Call *SortedEquInfoToRanks* to put the new ranking order into nTempRank.
3. Compare the new and old ranking orders, and if they are different copy the new into the old.
4. Call *SortedRanksToEquInfo* to set up the nSymmRank array with the atoms grouped according to symmetry.
5. Set value of bChanged for return.

18.38 CompRanksOrd

Comparison function for ranks array. Differentiates on rank, and if the ranks are equal on the pointers themselves.

18.39 SortedEquInfoToRanks

Puts new ranks into nSymmRank by looping backwards through nAtomNumber and for each atom in each symmetrically equivalent group assigning the highest atom number in the group as its rank. Assumption: the atoms are sorted by their initial ranking number in ascending order (See *CompRanksOrd*).

19 InChI Output Functions

19.1 FillOutINChI

Controls the output to the InChI string. The output is sent to member strings of the inchi_Output structure as follows:

szInChI – the InChI identifier itself
szAuxInfo – auxiliary InChI information
szMessage – messages from the code
szLog – InChI log information

19.2 SetConnectedComponentNumber

Sets the component number for atoms in at array.

19.3 SortAndPrintINChI

1. Sort component InChIs. Uses *qsort* to do two sorts, firstly with *CompINChINonTaut2* as the comparison function, secondly with *CompINChITaut2*. It is not obvious why there is a loop to do two qsorts (TAUT_NUM, TAUT_NON and TAUT_YES are defined in **mode.h**) because it looks as if the second sort (TAUT_YES) will always overwrite the first sort (TAUT_NON).
2. There are then two options, and the one chosen depends on the print options set up in INPUT_PARMS.
3. The second option results in one or more calls to *OutputINChI2*.

19.4 CompINChINonTaut2

1. Returns comparison of fragments by *CompINChI2* in non-tautomeric mode.
2. If this fails to differentiate the fragments and the flag CANON_FIXH_TRANS is set (which appears to always be the case), *CompINChI2* in tautomeric mode is used.
3. If the fragments are still not differentiated, the original fragment order numbers are used.

19.5 CompINChITaut2

This is similar to *CompINChINonTaut2*, with the modes reversed.

1. Returns comparison of fragments by *CompINChI2* in tautomeric mode.
2. If this fails to differentiate the fragments and the flag CANON_FIXH_TRANS is set (which appears to always be the case), *CompINChI2* in non-tautomeric mode is used.
3. If the fragments are still not differentiated, the original fragment order numbers are used.

19.6 CompINChI2

Comparison function for fragments. Comparison works in the following order.

1. Non-empty fragment takes precedence over empty fragment.
2. Two empty fragments cannot be distinguished.
3. Non-deleted fragment takes precedence over empty fragments.
4. Next use *CompareHillFormulasNoH* to differentiate on Hill formula.
5. Fragment with greater number of non-terminal H atoms takes precedence.
6. Fragment with greater atomic number in sorted atom list takes precedence. Code comment states this will be the same if the Hill formulae are the same.
7. Longer CT takes precedence.
8. Greater CT array takes precedence.
9. Greater total H count takes precedence.
10. Greater individual atom non-tautomeric H count takes precedence, with the exception that atoms with zero count take precedence over atoms with non-zero count.
11. Next use *CompareTautNonIsoPartOfINChI* to differentiate on non-isotopic tautomeric parts.
12. Next test on ‘non-tautomeric "fixed H" specific’.
13. Next use *CompareInchiStereo* to differentiate on non-isotopic stereo.
14. Fragment with greater number of isotopic atoms takes precedence.
15. Compare individual isotopic atoms in isotopic atom list. Greater atom number takes precedence, and if the atom numbers are the same, the greater isotopic difference takes precedence.
16. Compare individual isotopic H. For each atom greater T count takes precedence, followed by D count, followed by H (proton) count.
17. Fragment with greater number of isotopic groups takes precedence.
18. Next use *CompareInchiStereo* to differentiate on isotopic stereo.
19. If both fragments are charged, the lesser total charge takes precedence.
20. If only one fragment is charged, the uncharged takes precedence.

Note: the penultimate line in the function (commented ‘stable sort’) has been commented out. Presumably the calling function is expected to deal with a tie at this point.

19.7 CompareHillFormulasNoH

Compare Hill formula excluding H atoms. Considers both element types and the number of each element in the formula.

19.8 CompareTautNonIsoPartOfINChI

Compares tautomeric part of InChI.

19.9 CompareInchiStereo

Compares stereo part of InChI. Bonds are compared first then atoms.

19.10 OutputINChI2

Sets options for InChI output and calls *OutputINChI1*.

19.11 OutputINChI1

Outputs information to various files.

Output_file is used for returning the InChI string itself to the external calling function.

Text is added to the output file using the function *inchi_ios_print* for the InChI Identifier (as described in the Technical Manual – Figure 2) and the Auxiliary Information (as described in the Technical Manual – Figure 2a).

19.12 inchi_ios_print

Outputs string information to string buffer or to file or the standard output channel according to settings.

20 Key Generation Functions

20.1 GetStdINCHIKeyFromStdINCHI

Wrapper function for *GetINCHIKeyFromINCHI* to obtain standard InChI Key.

20.2 GetINCHIKeyFromINCHI

Main calling function to obtain InChI key from InChI.

1. Checks for valid InChI string.
2. Allocates memory for output strings.
3. Get major and minor strings from InChI for hashing.
4. Protonation treated as a special case and a flag is added to the end of the key after the minor part, the standard/non standard flag, a version flag and a separating hyphen.
5. The code uses the SHA-256 standard (published by NIST in 2002) to hash the major and minor strings (Devine, 2006). According to the code comments this is freely distributed software (**sha2.c** and **sha2.h**). No further details on the algorithms used are given here.

20.3 AddMOLfileError

Adds the specified error message to the error string. Most errors are generated by functions that read and pre-process connection tables.

Most of the error messages are hard-coded in English within the system. A few are generated where it is helpful to give more specific information about a problem.

20.4 GetCanonLengths

Utility to count lengths of various arrays and put the result into the structure s.

21 Files and Functionality

File	Lines	Functions	Contents
aux2atom.h	2770	13	General memory assignment
ichican2.c	5146	78	Canonicalisation
ichicano.c	2208	16	Canonicalisation utilities
ichicans.c	1640	27	Stereo
ichiisot.c	110	3	Sort keys
ichimak2.c	1203	12	Hill formula and stereo
ichimake.c	4725	34	Comparisons and utilities
ichimap1.c	865	30	Comparisons and tree stuff
ichimap2.c	2885	30	Sorting and ranking for canonicalisation
ichimap4.c	1647	2	Stereo
ichinorm.c	3324	43	Chemical structure
ichiparm.h	2613	7	Command line parsing
ichiprt1.c	4148	22	Printing
ichiprt2.c	1591	26	Strings for printing
ichiprt3.c	3200	26	More strings
ichiqueu.c	1412	16	Checking tautomeric structure
ichiread.c	8114	54	InChI input/output
ichiring.c	336	11	Rings and queues
ichirvr1.c	4929	54	Tautomers and structure manipulation
ichirvr2.c	6354	30	Tautomers and structure manipulation
ichirvr3.c	5508	3	Tautomers and structure manipulation
ichirvr4.c	3196	22	Tautomers and structure manipulation
ichirvr5.c	1175	3	Tautomers and structure manipulation
ichirvr6.c	1296	1	Tautomers and structure manipulation
ichirvr7.c	2330	21	Comparison and display
ichisort.c	550	27	Sort and comparison
ichister.c	3848	44	Stereo
ichitaut.c	4412	36	Tautomerism
ichi_bns.c	9603	104	BNS and tautomerism
ichi_io.c	1030	21	IO utilities
ikey_base26.c	1344	9	Hashing
ikey_dll.c	552	8	Key DLL
lreadmol.h	1266	17	MOL file reading

runichi.c	3986	31	High level controls
sha2.c	419	8	Hashing utilities/testing
strutil.c	4595	39	Structure utilities
util.c	1124	39	Structure utilities

22 Glossary of Terms Used in InChI Software

mcr – minimal class representative

BNS – Balanced Network Search

DFS – Depth First Search

BFS – Breadth First Search

23 References

Apodaca, R. (2006), “InChI Canonicalization Algorithm”, http://depth-first.com/articles/2006/08/12/inchi-canonicalization-algorithm/

Augeri, C. J. (2008), “On Graph Isomorphism and the Pagerank Algorithm”, Dissertation, Air Force Institute of Technology. http://www.sagemath.org/files/thesis/augeri-thesis-2008.pdf

Devine, C. (2006). “FIPS-180-2 compliant SHA-256 implementation”, http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf

Digital Chemistry (2010a), “Test Suites for the InChI Software”, Report prepared for the “InChI Trust InChI Software Assessment Contract”, revised version 8 March 2010.

Digital Chemistry (2010b), “Rewriting the InChI Software in C++ and/or Java”, Report prepared for the “InChI Trust InChI Software Assessment Contract”, 8 March 2010.

Fowler, M (1999), “Refactoring: Improving the Design of Existing Code”, Addison Wesley.

IUPAC (2010), “IUPAC International Chemical Identifier (InChI) InChI version 1, software version 1.03 (2010) API Reference”, 15 June 2010. http://www.iupac.org/inchi/download/version1.03/INCHI-1-DOC.zip

Kocay, W. & Stone, D. (1993), "Balanced Network Flows", *Bulletin of the Institute of Combinatorics and its Applications*, **7**, 17-32.

Kocay, W. & Stone, D’ (1995), "An Algorithm for Balanced Flows", *Journal of Combinatorial Mathematics and Combinatorial Computing*, **19**, 3-31.

McKay, B. D. (1981), “Practical graph isomorphism”, *Congressus Numerantium*, **30**, 45-87. http://cs.anu.edu.au/~bdm/nauty/pgi.pdf

Stein, S. E., Heller, S. R., Tchekhovskoi, D. V. and Pletnev, I. V. (2010), “IUPAC International Chemical Identifier (InChI) InChI version 1, software version 1.03 (2010) Technical Manual”, 25 June 2010. http://www.iupac.org/inchi/download/version1.03/INCHI-1-DOC.zip

Symyx (2010), “CTfile Formats”, http://www.symyx.com/downloads/public/ctfile/ctfile.pdf

Tchekhovskoi, D. V. (2011), e-mails to Digital Chemistry, 14, 23 and 24 Feb 2011.