Programmer's Guide to Cindex
April 2022

1. Cindex Overview

Cindex for Windows and Cindex for Mac are both 64-bit applications.

Cindex for Windows is compatible with Windows 7 and higher. Cindex for Mac is compatible with MacOS 10.9 and higher.

Cindex has essentially identical capabilities on the two platforms (minor differences reflect attributes of one platform that are not available on the other). Since version 3, released in 2012, both versions of Cindex use identical index file formats, and index documents are fully interchangeable.

Code Base

The code base and production of the Windows version are managed through Microsoft's Visual Studio development environment. For MacOS the code base and production are managed through Apple's Xcode development environment.

On both platforms code that manages the user interface is substantially distinct from code that manages core operations on the index (sorting, searching, spelling, record manipulation, etc.).

For the Windows version, user interface code is written in C (the application does not use MFC). For the Mac version, user interface code is written in objective-C and uses Cocoa classes

Core code is written in C and is identical on the two platforms except for minor differences that reflect adaptations to compiler requirements.

Design

Early History

When personal computers first became available Cindex was launched as a project to automate many of the repetitive tasks of indexing. Historically, indexers would construct each index entry on an index card, then sort and combine these before producing a typed index. Cindex adopted the index-card metaphor, and in its first version (for DOS) implemented this through underlying database software (dBase), in which each index entry occupied a database record. dBase was slow, and incapable of supporting rich on-the-fly sorting, so Cindex was rewritten as a native DOS program, retaining the 'record' as the fundamental building block of the index. The architecture was 16-bit, and indexes were limited to 65534 records. Each fixed-length record could contain up to 16 variable length fields (15 levels of heading plus a 'locator' field); the user-specified record size could not exceed 2000 characters (bytes). Multiple indexes could be open concurrently (limited only by available memory).

Evolution of Architecture

Current versions of Cindex retain some structural features of the early versions. Index records are still fixed-length (up to 2000 bytes) and are still limited to 16 variable-length fields. The fixed-length record[1] is fundamental to the architecture, but the character and field limits are not. They have never been changed because there has been no indication that they are insufficient.

When Windows and MacOS became 32-bit operating systems, Cindex was reworked to exploit them, and the possible index size was enlarged to 2^{32}-1 records. Because at the time most personal computers still had relatively small amounts of directly addressable memory (potentially less than the size of an index), Cindex deployed the underlying operating system's memory mapping capabilities to provide an address space that was limited only by the storage capacity of hard disks. Current versions of Cindex still work with memory-mapped index files.

With the release of version 4 in 2019, Cindex became a 64-bit application on both platforms, but without any changed limitations (e.g., maximum number of index records is still 2^{32}-1).

Unicode

Since the release of version 3 in 2012, Cindex has provided full Unicode support, and all its internal operations are Unicode-based. This extends beyond character representation to include such things as language-specific collation for sorting index entries. Unicode support relies on the open-source International Components for Unicode (ICU).

The adoption of Unicode was accompanied by the adoption (both platforms) of the open-source Hunspell spell-checking library. Hunspell is widely deployed in many applications (e.g., Firefox) and as a result dictionaries for many languages are freely available.

Some Internal Details

Sorting and searching

Index records are always implicitly sorted, regardless of whether Cindex is set to display them in sorted order. Whenever an index record is added, or its content changed, Cindex uses the sort rules currently in effect to calculate its sorted position in the index, and stores that information in the record. To make possible this effectively instantaneous sorting, and to support essentially instantaneous lookup to find (or not) a record with particular content, Cindex uses an AVL tree. The information needed to describe a record's position in the tree is stored within the record.

[1] The fixed-length record has never been a practical limitation because the user can change record size (up to the limit of 2000 bytes) on demand.

Document structure

Index documents (filename extension ucdx) have three components: a header block, a block containing index records, and a block containing information about index 'groups'. These three blocks (described below) lie sequentially in the file. At runtime, the header block is loaded into working memory. The record block and the groups are memory-mapped, expanding and contracting as needed.

Header

The index header block contains a set of structures that define all the settable attributes of the index, including:

- Record structure (size, field specifications, etc.)
- Sorting parameters (sort language, rules, etc.)
- Locator information (how to recognize page references and cross-references)
- Formatting information (index style, layout, display font and attributes, etc.)

Record structure

Index records are placed sequentially in order of their entry in the index, starting immediately after the header.

An index record consists of a fixed-length header (28 bytes) plus space for record fields. The header, which does not count against the 2000 character limit for content, contains fields that specify the record's identifier, its modification time, its sorted position in the index, various status flags (deleted, labeled, etc.), and the ID of the user who last edited it.

An index record must contain at least two fields (a main heading field and a locator field) and may contain up to 16 (fifteen levels of heading plus the locator). The locator field is always the last field, regardless of how many are in the record. The content of each field is a null-terminated UTF-8 string, and successive fields are laid out sequentially within the record, forming a compound string terminated by a 0x7F byte.

Text attributes (italics, boldface, etc.) and font identifiers are represented by simple codes within record fields at the positions where they take effect. Each text attribute is represented by two bytes, an introductory code (0x19) followed by a byte representing the bits of attribute. Each font (of a possible 32) is represented by an introductory byte (0x1A) followed by a byte containing the font index.

Record groups

The Cindex user can define, by search or otherwise, a subset of records (a 'group') that appears to the user as though it contained the only records in the index. Groups can be temporary, or can be saved in the index document. When Cindex saves a group it stores a collection of record identifiers, not the records themselves.

Document Conventions

The following color conventions are used in this document:

File name

Class name (Objective C)

Method or function name

Type definition

Instance or global variable

Preprocessor directive/macro

2. Development Environments

MacOS

The project is managed entirely within Xcode (current version 13.3.1). The project is compiled for Intel processors. It has been tested on M1 Macs and runs correctly. The deployment target is MacOS 10.9.

All code is C or Objective-C, and deploys Cocoa classes along with native C code. The project uses Automatic Reference Counting (ARC) to manage memory allocation and release.

There are three deployment targets (Cindex 4; Cindex 4 Student; Cindex4 Demo), each having a debug and production version. The three versions differ in their capabilities. The capabilities of the student and demo versions are controlled by preprocessor macros (_STUDENTCOPY, _DEMOCOPY) defined in the project settings for the relevant target. These macros in turn are used to limit two capabilities of the student and demo versions:

- The maximum number of records an index may contain is set to 500 (Student) or 100 (Demo).
- Neither the Student nor the Demo version will check for a software update.

In all other respects the Student and Demo versions are identical to the standard version.

All files required by the project are contained within a Cindex 4 folder.

Dependencies

Public domain libraries (expat, hunspell ICU, libiconv) are obtained from MacPorts (https://ports.macports.org). The compiled libraries are in folders inside the macports folder in the project folder.

The Sparkle updater framework, obtained from https://sparkle-project.org, is contained in the Sparkle folder in the project folder. DSA keys are in the DSA Keys folder.

Help

Built-in help is provided as an Apple Help Book (https://developer.apple.com/library/archive/documentation/Carbon/Conceptual/ProvidingUserAssitAppleHelp/authoring_help/authoring_help_book.html). The HTML files are in the Help folder inside the project's en.lproj folder.

Windows

The project is managed entirely within Microsoft Visual Studio 2017. The project has a deployment target of Windows 8.1. The standard build is for a 64 bit application (x64) but the project is also able to build 32-bit (Win32) versions of the app.

All code is in C. The project does not make use of MFC classes.

There are four production configurations (Release; Publisher; Student; Demo). The Release and Publisher targets have counterpart debug configurations. The Student and Demo versions are identical to the Release version, except for:

- The maximum number of records an index may contain is set to 500 (Student) or 100 (Demo).
- Neither the Student nor the Demo version will check for a software update.

The restrictions on the student and demo versions are controlled by a preprocessor macro (TOPREC) that specifies the maximum number of records the index can contain. When defined, this macro also controls whether Cindex includes code to check for software updates.

All the files required to build the project are in the Windows folder in the supplied archive. Inside that folder all components except the public domain libraries are in the Cindex 4 folder. Public domain libraries (expat, hunspell ICU, libiconv, WinSparkle) are in folders inside the Libraries folder. They are referenced by a relative path from the project.

Cindex API

The Publishers' Edition of Cindex, provides capabilities beyond those available in the standard edition. These are described in the Publisher's Edition supplement to the User's Guide. The Publisher's Edition also provides an API that allows other windows applications to control certain of its operations. The API is built on the DDE messaging protocol, and for clients is implemented through a library provided as a dll. Source code and documentation for the API is in the cindexapi 3.2 folder inside the Windows folder.

Help

Built-in help is provided via Microsoft's HTML Help (https://docs.microsoft.com/en-us/previous-versions/windows/desktop/htmlhelp/microsoft-html-help-downloads). The help files are in the Cindex Help folder inside the Cindex 4 folder.

3. Cindex Documents

Cindex can read and write six different kinds of native documents that are directly accessible to the user (i.e., the user can access the files outside Cindex). Most are fully interchangeable between Mac and Windows. Where that is not the case, the description notes it.

In addition to documents the user can access directly, Cindex creates others (e.g., markup tags, user dictionaries) whose contents the user can manage, but cannot access outside the app. These are described later in this section.

Index Documents (filename extension ucdx)

A saved index document contains a working index. It has three components: a header block, a block containing index records, and a block containing information about index 'groups'. These three blocks lie sequentially in the file.

File Header

The elements of this are defined in HEAD structure in the headparams.h file.

char endian;	// byte order (legacy; now always little endian)
char cinkey;	// cindex key (now unused)
long bspare;	// spare
long headsize;	//size of header (bytes)
unsigned short version;	// Cindex version #
RECN rtot;	// total number of records in index RECN defined as unsigned int32
unsigned long groupsize;	// space required in file for groups
time_c elapsed;	// time spent executing commands (time_c is int32 version of time_t, required for legacy reasons to be limited to 32 bits. All times in seconds from Unix base date)
time_c createtime;	// time of creation
short resized;	// true if resized (legacy, no longer used)
short spare1;	// spare
time_c squeezetime;	// time of last squeeze (index compression)
INDEXPARAMS indexpars;	// index structure params (record size, fields requirements, etc.; defined in indexparams.h)

SORTPARAMS sortpars;	// sorting (alphabetizing) params; defined in sortparams.h
REFPARAMS refpars;	//reference params (locators, cross-references; defined in referenceparams.h)
PRIVATEPARAMS privpars;	// implicitly remembered settings related to index display; defined in indexparams.h
FORMATPARAMS formpars;	// format parameters (layout of index; defined in formatparams.h)
char stylestrings[STYLESTRINGLEN];	// strings for auto style (words for au application of italic, bold, etc.)
char flipwords[STSTRING];	// prefixes for transposition when entries flipped
char headnote[HEADNOTELEN];	// text to be displayed at head of index (unused)
FONTMAP fm[FONTLIMIT];	// mapping of local IDs in records to font names; defined in fontmap.h
RECN root;	/ number of root record of alpha tree */
short dirty;	/* TRUE if any record written before close */
IRRect mainviewrect;	// position of main document window on screen (Mac only)
IRRect recordviewrect;	// position of record entry window on screen (Mac only)
RECT vrect;	// position of main document window on screen (Windows only)
unsigned int mmstate;	// Window maxmin state (Windows only)
char spare[1024];	// spare (unused)

Index Records

Index records are laid out sequentially in the index file immediately following the header block. All records have the same fixed size, though that size can be changed when the user requires. Each record has a header defined by the RECORD structure in the recordparams.h file. The last field of the header is the start of the array of characters that constitute the record text.

RECN num;	// record number (RECN is typedef'd as uint32)
RECN parent;	// number of parent record (AVL tree)

```
RECN lchild;          // number of left child record (AVL tree)
RECN rchild;          // number of right child record (AVL tree)
time_c time;          // time of last change
char user[4];         // up to four initials of user who last changed the record
unsigned ismod : 1;   // unused (legacy)
unsigned isdel : 1;   // true if record deleted
unsigned ismark : 1;  // true if record marked
unsigned islock : 1;  // unused
unsigned balance : 2; // holds balance of subtree on record (AVL tree)
unsigned isgen : 1;   // TRUE if autogenerated
unsigned isttag : 1;  // TRUE if temporarily tagged (used during compress / expand)
unsigned label : 3;   // holds tag ID
short padding;        //padding to align record text on 32 bit boundary
char rtext[];         // text of record (empty array as pointer)
```

Organization of Text in a Record

The text content of a record is arranged as a sequence of UTF8 strings, each terminated by a null byte, with a 0x7F byte following the last null. Each record must start with a 'main heading' string and end with a 'locator' string that contains page references or cross-references, or equivalent. The minimum number of strings in a record is therefore two. A record may contain up to 14 additional strings representing levels of subheading. If present, the subheading strings lie sequentially between the main heading string and the locator string.

Text attributes (typefaces, styles) are encoded by special two-byte sequences contained within record strings at positions where the attributes are to be applied or removed. Possible text styles, all of which can be applied or removed together, are bold, italic, underline, small caps, superscript, subscript. A style, or style change, is introduced by a 0x19 byte followed by a byte that identifies attributes (from cintypes.h), as follows:

```
FX_BOLD = 1,
FX_ITAL = 2,
FX_ULINE = 4,
FX_SMALL = 8,
FX_SUPER = 16,
```

```
FX_SUB = 32,
FX_OFF = 64,                    // off code (attributes to be turned off)
FX_STYLEMASK = (FX_OFF-1),
FX_BOLDITAL = (FX_BOLD|FX_ITAL)
```

Each index may deploy up to 32 typefaces (fonts). There is always a default typeface, and potentially 31 others. Within index records a typeface is represented by an introductory byte (0x1A) followed by a byte that contains the numerical index that identifies it. Numerical index is associated with a named typeface via the fm array of FONTMAP structures in the index file header.

> **NOTE:** When Cindex displays compound text strings (xstrings) derived from records it may present them in color (e.g., when a record is 'tagged'). In such cases the 0x1A byte is also used to introduce a color specification. The cintypes.h file contains the set of possible values in the typeface/color byte. The xstrings in Cindex records do not contain color attributes.

Record Groups

A user can define arbitrary sets of records ("groups") that can be managed as though they were the only records in the index. A group can be defined be capturing a selection on the screen or as the result of a search, and the user can save the group with a name. Saved groups are laid out in sequential order in the index file immediately after the records. Each group has a header defined by a GROUP structure in the file groupparams.h. The last field of the header is the array of numerical identifiers of records that comprise the group.

```
int size;
time_c tstamp;              // time created (Unix base time)
time_c indextime;           // time parent index created
char gname[GROUPLEN];       // name of group
char comment[GPNOTELEN];    // comment (unused)
RECN limit;                 // max number of records that can be in group
short gflags;               // flags describing group status (defined in group.h)
LISTGROUP lg;               // search specification (if group formed from search)
SORTPARAMS sg;              // sorting rules for group
RECN rectot;                // number or records in group
int spare[32];              // unused
```

```
RECN recbase[];          //record numbers of group members (empty array as pointer)
```

Indexes in memory

When an index file is opened for use, the header is read into a HEAD structure in memory, and this working copy is modified as needed when index attributes are changed. The header is written to the file when the index is flushed or closed.

Index records and index groups are never read directly into memory. Instead, the index file is memory mapped, and the relevant parts (records, groups) are read and written from the memory-mapped object. This design dates from a period in which the available working memory on a computer could be smaller than the size of an index file, and memory mapping was required to ensure a sufficient addressable space. Given the amount of memory available on modern computers, memory mapping is almost certainly no longer necessary, but it still works well.

Template Documents (filename extension utpl)

A template is a model of an index. The template file contains only the HEAD structure of the index from which it is derived. It contains no records or groups. The user can create a new (empty) index from a template.

Style Sheets (filename extension ustl)

A style sheet contains information about the layout and appearance of an index. It can be derived from an existing index, or from the current default settings of the relevant formatting parameters. Its contents are defined by the STYLESHEET structure (formatparms.h)

```
char endian;             // byte order (legacy; now always little endian)
FORMATPARAMS fg;         // format parameters (defined in formatparms.h)
FONTMAP fm;              // mapping of local font IDs to font names; defined in fontmap.h
short fontsize;          // fontsize in points (from PRIVATEPARAMS)
int spare[32];           // unused
```

When derived from an existing index, the elements of the STYLESHEET are taken directly from the corresponding elements of the index HEAD structure. When derived from the Cindex default settings, elements are taken from the saved Cindex preferences.

When a user loads a saved stylesheet the element values in it replace the corresponding ones in the index HEAD structure.

XML Records (filename extension ixml)

The file contains index records fully encoded as XML elements, along with information about the fonts represented in them. These records can be imported into an existing index, or used to create a new index.

Abbreviations (filename extension abrf (Mac) or ucbr (Windows))

An abbreviation represents the association between a key word (the 'abbreviation') and a phrase. When the user types an abbreviation in an index record, Cindex replaces the text with its associated phrase.

Abbreviations and their associated phrases are saved in files that users can load as needed. Files are not interchangeable between Mac and Windows versions of Cindex.

MacOS

The set of abbreviations and phrases is held in an NSDictionary, where for each key-value pair the abbreviation is the key and the associated phrase its value. Cindex uses NSKeyedArchiver and NSKeyedUnarchiver to save and load the dictionary as needed.

Windows

The set of abbreviations is stored in the file as a paired text strings, each pair on a single line and separated by a tab:

abbreviationTABexpanded_phraseCRLF

in which 'abbreviation' is the abbreviation and 'expanded_phrase' is the expanded form that will be placed in records.

The first line in the file contains a (now redundant) header that identifies the file contents as abbreviations.

Archive Documents (filename extension arc)

The archive is a legacy file format (from the earliest versions of Cindex) for saving index records (including information about font and style attributes). Before Cindex deployed Unicode—introduced in version 3—Windows and Mac versions of Cindex used different character sets (CP1252 and MacOS Roman, respectively) to represent a limited set of non-ASCII characters. The archive permitted interchange between the two platforms. It has been deprecated since Cindex version 3 because of its limited capacity represent characters, but is supported for backward compatibility with earlier versions.

An archive contains three parts, a header block, a block containing records, and a block containing information about fonts used.

Header

Each document begins with a 16 byte header:

- 4 bytes that identify the source platform (Windows: 0x01010101; Mac: 0x02020202)

- A 10 byte string representation of the offset (in bytes) to a text block at the end of the file that contains information about the fonts used in records (see later description).
- 2 termination bytes 0x0D0A.

Record Text

Immediately following the header block, records are arranged sequentially.

The text of a record is contained in a C string, with each field separated from the next by a 'tab' (0x9) character. The record must contain at least two fields, a main heading and a field containing locators (page references). A record may contain up to sixteen fields, depending on the number permitted in the index that contains the record. The last field is always the locator field. The record is terminated by CRLF (0x0D0A).

A record string may contain additional information appended to the last (locator) field, but before the line terminator. This 'extended' information encodes the following:

The record status (labeled, generated or deleted).

The time at which the record was last modified (seconds since Jan 1 1970).

The four-character identifier of the user who last modified the record.

The extended information begins with the character 0x13, and has the following format:

0x13#*time_in_seconds userid*

is a single character that encodes record status:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
special_last	TRUE		label2	label1	generated	label0	deleted

The bits label0, label1, label2 when aggregated define a 3 bit field that represents the value of the label on the record (for archives made by Cindex versions < 2.0, only label0 is used).

The special_last bit indicates that the record came from an index with a required last field. The bit is set only if the special field contained text.

If any status bit is set, Bit 6 is TRUE and # resolves to an uppercase letter. Otherwise # is a space (0x20).

time_in_seconds is an ASCII decimal number representing the time (in seconds since Jan 1 1970) at which the record was last modified. *time_in_seconds* follows # without any additional space.

userid is a four-character user identifier. It is separated from *time_in_seconds* by a single intervening space (0x20).

Text Styles and Fonts

Beyond characters from the ANSI character set, a record can contain, within its text fields, codes that identify type styles and fonts. A code is identified by two characters, the first of which always has the value 0x1A. The second character contains the style or font code (never both concurrently). If the character identifies a style or styles it has the following format.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Turn off		subscript	superscript	smallcaps	underline	italics	boldface

Bits 0 through 5 identify the style(s) to be turned on or turned off. The state of Bit 7 determines whether the style(s) is to be turned on or off. When TRUE, the styles are turned off. Bit 6 is always FALSE.

If the character identifies a font it has the following format.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
	Font		id 4	id 3	id 2	id 1	id 0

Bits 7 and 5 are always FALSE and Bit 6 is always TRUE. Bits 0 through 4 encode one of 32 possible font identifiers. Font 0 is always the CINDEX default font; Font 1 is always the Symbol font. Fonts 2 through 31 can represent any fonts. The mapping of font identifiers to particular fonts is handled through the font table.

Font Information

The last component of the archive (to which the file header contains a string representation of the offset) is a text block that identifies the fonts used in records. The block is organized as a series of text lines, each terminated by CRLF (0x0D0A). Each line has this structure:

N@@preferred_font@@alternate_fontCRLF

where *N* is a decimal integer (0-31) representing the font ID, 'preferred_font' is the name of the preferred font, and 'alternate_font' is the name of the font to substitute if the preferred font is unavailable.

Tag Sets (filename extension cxtg or cstg)

Tag sets contain the specifications for XML or SGML markup of exported formatted indexes. Some tag sets of both types are installed with Cindex, and can be viewed but not edited. Users can also define and edit their own tag sets.

A tag set is contained in, and completely specified by, a TAGSET structure, defined in tags.h:

```
short tssize;          // size of TAGSET
int total;             // complete size of object
```

```
short version;          // version of tags
char extn[4];           // default extension for output file
char readonly;          // TRUE if set unmodifiable
char suppress;          // TRUE if suppressing ref leader & trailer
char hex;               // true when hex; otherwise decimal
char nested;            // TRUE when nested
char fontmode;          // font tag type
char individualrefs;    // TRUE if references coded individually
char levelmode;         // heading level tag type
char useUTF8;           // encode uchars as utf8 (SGML only)
int spare[5];
char xstr[];            // base of compound string
```

The empty xstr array at the end of the structure is the base of a compound string (xstring) that contains the array of elements specifying the names of all tags. The positions of individual tags in the array are specified by a set of enums and preprocessor definitions in tags.h.

Tag sets for XML markup and SGML markup, although contained in the same TAGSET structures, are deployed differently (XML markup is syntactically much stricter). They are distinguished by the file types in which they are saved and by the properties that can be managed by the user. For more information see the later discussion of Exporting Formatted Indexes, and the discussion of Markup Tags… under the Tools menu.

Tag sets are loaded and (if user-defined, saved) via functions in tags.m (Mac) or tagstuff.c (Windows). User-defined tag sets are kept in the following (normally hidden) directories:

Mac	~/Library/Preferences/Cindex
Windows	~\AppData\Local\Indexing Research\Cindex 4.0

User Dictionaries (filename extension pdic)

User dictionaries contain lists of user-specified words that are considered properly spelled. Dictionaries are created explicitly by the user via the Spell panel. A word is added to the active user dictionary when the user clicks the Add button. A dictionary is stored as simple plain text file that contains one word on each line. Dictionaries are automatically saved and loaded from the following (normally hidden) directories:

Mac	~/Library/Preferences/Cindex
Windows	~\AppData\Local\Indexing Research\Cindex 4.0

Preferences

Preferences contain settings that govern the overall behavior of Cindex. Cindex is installed with built-in default values. Changes made by the user are saved automatically, and restored each time the app is launched.

Preferences are contained in the prefs structure, defined in globals.h (Mac) and cglobal.h (Windows).

```
long key;                           // key number identifies file generation
struct hiddenpref hidden;           // hidden preferences
struct genpref gen;                 // general preferences
struct langprefs langpars;          // spell checker language settings
INDEXPARAMS indexpars;              // index structure prefs
SORTPARAMS sortpars;                // sort preferences
REFPARAMS refpars;                  // ref preferences
PRIVATEPARAMS privpars;             // private preferences
FORMATPARAMS formpars;              // format preferences
char stylestrings[STYLESTRINGLEN];  // strings for auto style
char flipwords[STSTRING];           // strings for flipped words
short hotkeys[MAXKEYS];             // hot key codes (Windows only)
char keystrings[STSTRING];          // hot keys strings (Windows only)
char spare[STSTRING];               // unused (Mac only)
```

The working copy of preferences is held in the global variable g_prefs in globals.m (cglobal.c). This is statically initialized with default values. Each time the app is launched, it looks for a saved copy of the preferences to replace the built-in ones:

- **Mac:** the function global_readefaults() in globals.m is called to load saved preferences from an NSData object saved in standardUserDefaults. If no object has been saved, the built-in defaults are used, and saved as current values.
- **Windows.** The function file_loadconfig() in files.c is called to load saved preferences from ~\AppData\Local\Indexing Research\Cindex 4.0. If no file has been saved, the built-in defaults are used. Current values are saved after launch.

NOTE: on both platforms, if the user holds down the shift key when the app is launched, it loads the built-in defaults and sets them as saved preferences.

Some members of the prefs structure are set explicitly by the user. Others are set implicitly.

Explicit Settings

The gen member (structure genpref defined in globals.h or cglobal.h) contains the settings managed by the Preferences… command.

hidden. Only one member of the hiddenpref structure is used. The user member contains the user's ID, set either via the first tab of the Preferences …panel or, if the *Prompt for User ID* Preferences button is checked, via the panel displayed on application launch.

Implicit Settings

langpars (langprefs structure defined in globals.h or cgolbal.h) contains the most recent values of some settings from the Spell panel presented by the Spelling… command.

When no index is open for work, the indexpars, sortpars, refpars, privpars, formpars, stylestrings, flipwords and (for Windows only) hotkeys and keystrings members of prefs are set by various commands in the Documents menu and the Tools menu. For example, the Sort… command given when no index is open sets the sortpars member of prefs. Later sections of this document, covering the Document menu and Tools menu, provide particulars of other settings.

Other Documents

Import

Cindex can import index records from text files in which each record is contained on a single line, with fields delimited by some marker. Content can be imported from simple tab or quote-delimited text files, and from variants of these created by the indexing programs SKY Index and Macrex. The latter contain, in addition to the record text, specific character sequences to identify styles/fonts. The later section Importing Index Records contains more information about importing records.

Export

Cindex can export formatted indexes in several standard file formats. The later section Exporting Formatted Indexes provides more information.

4. Architecture

This section describes the organization of Cindex code. It begins with overviews of the Mac and Windows architectures, then provides notes about the code supporting major functions.

Interface code is platform-specific, but much of the underlying C code is common to both platforms, and exists in identically-named files (e.g., sort.m and sort.c, which contain core sorting code for Mac and Windows respectively).

A Note on Internal Representation of Text

The text of Cindex records is held in a sequence of null-terminated UTF8 strings, with the last one followed 0x7F as a termination byte. This 'compound string' or 'xstring' is widely deployed throughout the app. Many functions, and several macros, exist to manipulate it as a unit. See strings_c.h (Mac) and strings.h (Windows).

The xstring containing record text also contains formatting codes (fonts, styles) that are embedded within it (see Organization of Text in a Record). On both platforms some text operations (e.g., displaying on the screen, or recovering edited text from windows), require an object in which the text and its attributes are separately represented. The ATTRIBUTEDSTRING (attributedstrings.h) is used to provide a bridge between the xstring and native attributed text formats. Functions to manage ATTRIBUTEDSTRING are in attributedstrings.m (Mac) and attributedstrings.c (Windows).

Mac Overview

The application is built on Cocoa classes. The application (instantiated in the IRApp class, a subclass of NSApplication) deploys the document architecture subsystem (NSDocumentController, NSDocument, and associated classes) to manage Cindex documents. The general design substantially separates the management of user interface from the management of operations on documents.

A list of Mac code files with notes on their functions is provided as Appendix 1.

Document Architecture

When launched, the app spawns a single instance of IRIndexDocumentController (a subclass of NSDocumentController) and sets that as the application delegate.

The application instance (IRApp) does almost nothing. Most of the major top level operations (opening and saving files, document creation, managing settings, etc.), are run through its delegate (IRIndexDocumentController).

Different types of Cindex documents are subclasses of NSDocument (e.g., IRIndexDocument, IRIndexArchive, IRAbbreviations) and are managed by the document controller (IRIndexDocumentController).

Document windows on screen are managed by subclasses of NSWindowController. The index document (IRIndexDocument) has three window controllers, each of which manages a different kind of window:

IRIndexDocWController to manage the main index display

IRIndexRecordWController to manage the record to manage the record-entry window

IRIndexTextWController to manage various forms of text display related to reports.

The window managed by each of these window controllers contains a main view, and sometimes additional subviews for controls. Each main view is a subclass of NSTextView:

IRIndexView manages the displayed index in the main window.

IRRecordView manages the display in the record window.

IRTextView manages the display in the text window.

Other Top-Level Components

Outside the document architecture the app uses subclasses of NSWindowController to present NSPanels through which the user manages various capabilities and settings (e.g., search and replace, checking spelling, gathering index statistics, sorting entries).

With three exceptions (panels for search, replace, spelling) the panels are presented modally, and require explicit closure by the user.

Although most panels through which the user makes settings or triggers actions operate on the frontmost index, some can operate in two modes:

- If called when an index is open, the panel is presented modally on the frontmost index document and settings apply to that document only.
- If called when no index is open, the panel presents default settings that the app will apply when it creates a new index.

The following controllers (grouped in the project under 'Dual Interface') have this dual role: CrossRefsController, GroupingController, HeadFootController, HeadingsController, MarginColumnController, PageRefsController, RecordStructureController, RefSyntaxController, SortController, StyledStringsController, FlipStringsController, StyleLayoutController.

The various controller subclasses are more fully described later in this document.

Startup and Termination

Startup and termination code is executed in the app delegate (IRIndexDocumentController).

Command Dispatch

Most app operations are managed through the main menu and its submenus. The menus are contained in the file MainMenu.nib. Each menu item has an associated action

that is either explicitly directed to a method in the owning class (IRIndexDocumentController), or more commonly is directed at the First Responder, and is automatically passed up the app's responder chain until the system finds an instance of the action method in the chain. For the main index window the responder chain starts with IRIndexView; for the record-editing window the chain starts with the IRRecordView. In both cases the chain ends with the app delegate (IRIndexDocumentController).

Menu items are enabled and disabled (via the validateMenuItem method in the relevant responder) according to the context in which they are called. The validation methods rely on menu item tags (defined in cindexmenuitems.h) to identify menu items that need attention. Toolbar items in the main index window and the record entry window are validated in the same way, via validateToolbarItem.

Opening Documents

Menu commands for opening and closing native documents are executed by the app delegate. Commands for importing and exporting index content in other file formats are executed through the IRIndexDocument instance for the index.

When a call is made to open a document, the app delegate method openDocumentWithContentsOfURL:display:completionHandler: checks the file type and if necessary calls a conversion function (e.g., v3_convertindex()) to convert a document created by an earlier version of the app into the current document format (the structure of saved documents is described in Index Documents). openDocumentWithContentsOfURL (via a call to super) then calls makeDocumentWithContentsOfURL:ofType:error: If the URL points to an existing index, this method (via super) calls readFromURL:ofType:error: If the URL points to an archive or xml records, the app calls initWithName:template:hideExtension:error:outError to create a new index. In either case the app then sets up the internal representation of the index.

Internal Representation of the Index Document

Each instance of IRIndexDocument contains a master structure (INDEX, defined in indexdocument.h) that holds the information need to manage the index in memory.

The first step creates a mapped memory representation of the document file via a call to mfile_open (contained in mfile.m). The resulting configuration is saved in the MFILE structure in the INDEX structure.

After mapping is configured, the index HEAD structure is copied from the mapped memory into the INDEX. The records and groups in an index are not copied. They are always addressed through the mapped image. To allow for the addition of index records, index_setworkingsize() in index.m is called to create empty space in the mapped image. Unused space is removed when the index is closed.

A call to _installIndex initializes objects needed by the index, such as the collator for sorting Unicode. To allow for the addition of index records, index_setworkingsize() in

index.m is called to create empty space in the mapped image (unused space is removed when the index is closed). The app then ensures that the memory mapped records will be aligned on 4-byte boundaries (they might not be if the file was created by Cindex 3, a 32-bit app) before scanning all the records to check their integrity (index_checkintegrity in index.m). If the checks fail, the app offers remedies as best it can. If checks are passed, the app checks that fonts deployed in the index are available or, if not, have substitutes that are available.

The INDEX is the key internal object through which the app manages almost everything about an index. Pointers to it are widely deployed through the app. When the app changes attributes of an index (record structure, formatting specifications, sorting rules, etc.) the changed attributes are held in the HEAD member of the INDEX.

Displaying Index Content

The main view of the index is managed by the IRIndexView class (a subclass of NSTextView), which is owned by the NSWindow managed by the IRIndexDocWController. The IRIndexView content is an NSMutableAttributedString—a class that manages text and associated formatting attributes (fonts, styles, paragraph layout, etc.). After an index has been loaded, the IRIndexDocWController method _redisplay is called via a notification. _redisplay then calls through to _fillViewFromRecord followed by _loadMainView to specify the layout of formatted entries render them on screen.

The app allows the user instantly to view, in formatted form, any part of the index (e.g., entries that begin with a specified word or phrase). The user can instantly switch between differently formatted layouts (e.g., indented, or run-in, fully formatted or draft formatted, etc.) and can also scroll freely through the formatted text. The app therefore has to quickly render any arbitrary part of the index. The full formatted text of a large index could be 10-30 MB or more—far larger than can be rendered speedily into a single NSMutableAttributedString. The app therefore formats only that fragment of index that the user will see in the window.

Building displayed text from records

The first stage builds an array of entries to be laid out on the screen. For a *draft-formatted* index, the displayed text of an entry conforms closely to what is in the record; for a *fully formatted* index, the displayed text can differ substantially from what is in records: headings can be collapsed, depending on whether the index is indented or run-in, page references are sorted and abbreviated/conflated as needed; cross-references are sorted and placed after page references; formatting punctuation is inserted where required by the chosen layout; formatting codes are inserted where automatic styling is required for specified words in headings, or components of page references.

Starting with the record that will appear at the top of the screen, then moving sequentially through records, _fillViewFromRecord calls. _makeDescriptorForRecord to

build the formatted text of each entry and initialize a LayoutDescriptor with that text. The LayoutDescriptor calculates the number of lines the entry will occupy on the screen. _fillViewFromRecord creates as many descriptors as are needed to fill the screen, and places them in an array.

The formatted text of an entry is held in the LayoutDescriptor as an IRAttributedDisplayString (a subclass of NSAttributedString). The IRAttributedDisplayString is a wrapper for an IRDisplayString. During its initialization, the IRDisplayString builds the formatted text of an entry by calling draft_buildentry() in drafttext.c for a draft display, or form_buildentry() in formattedtext.m for a fully formatted display. draft_buildentry() and form_buildentry() each consumes as many index records as are required to build a complete entry.

The call to _fillViewFromRecord is followed by a call to _loadMainView, which adds the text managed by each descriptor to the main display.

When the user issues a command to display a specified part of the index (e.g., entries beginning with a specified letter or phrase) the app always fully refreshes the display via a call to _fillViewFromRecord followed by _loadMainView. When the user steps through the displayed index line-by-line using the keyboard, the app calls stepRecord:from: which modifies (rather than rebuilds) the list of display descriptors. When the user steps by page, or scrolls by arbitrary amounts using the scroll bar, the app (via a notification) calls _scrollMainView to first calculate which record's content should appear at the top of the window, then resets the view, by modifying or replacing the existing set of display descriptors.

Estimating text position within the formatted index

The display in the main index window shows (via scroll bars) the relative position of the text in the index and (via the thumb) the fraction of the index displayed. Similarly, when the user moves the scroll bar to a new position in the window, the app shifts the display to the corresponding position within the index. Without building a full representation of the displayable index (prohibitively slow for a large index) it is impossible to know precisely how much space the formatted text occupies, how much of it is displayed on the screen, and where within the full text the displayed segment sits. To manage scrolling the app therefore uses estimates of a) the size of the fully formatted index; b) the fraction of the index displayed on the screen; c) the position within the index of the displayed segment.

When the main window is first opened or the user changes the display format (e.g., from draft-formatted to full-formatted) the IRIndexDocWController calls windowTitleForDocumentDisplayName to (among other things) estimate the overall size of the displayed index. If the window is resized the IRIndexDocWController calls _sizemainView. Both methods call through to _setScrollLimit, which lays out a sample of

records to estimate how much space the whole index will need, and what fraction of it can be displayed in the window.

Whenever the main display is refreshed (e.g., after the user issues a command to display a particular part of it), the IRIndexDocWController calls _setScrollIndicator. This in turn calls sort_findpos() in sort.m to find the ordinal position of the first displayed record in the sorted index. That position is then used to set the scrollbar thumb.

When the user moves the scrollbar thumb, the IRIndexDocWController calls _scrollMainView: to find the correct segment of the index to display. Depending on the thumb position (top, bottom, or somewhere else), this then calls sort_top(), sort_bottom() or sort_findpos() to find the index record at the top, bottom or some intermediate position in the sorted index.

Windows Overview

The Windows version of the app uses the Win32 API and is written entirely in C. It does not use MFC or any other object frameworks. Application design generally follows the standard structure of Win32 programs.

A list of Mac code files with notes on their functions is provided as Appendix 2.

Document Architecture

Cindex uses the Multiple Document Interface (MDI) to manage frame, client and child windows. The frame window is the main application window. Each index is associated with one or more child windows. The client window is an invisible container for child windows.

Each open index can have up to three user-visible MDI child windows, whose classes and attributes are configured by the initcindex() function in main.c:

> *Main index window*. Code to manage this is in viewset.c.
>
> *Record-entry window*. Code to manage this is in modify.c. The record window, if present, floats on top of the main index window, or appears embedded within it, according to user preference.
>
> *Text window* (for displaying various forms of text display related to reports). Code to manage this is in text.c.

Each index has two additional MDI windows that are invisible. They work together to provide containers for the main index window and the record-entry window.

> The 'container' window, managed through code in containerset.c, provides a container for the main index window, and also for the record-entry window if the user opts for an embedded display (i.e., the record-entry window appears at the foot of the main window).

The 'record container' window, managed through code in rcontainerset.c, provides a container for the record-entry window when the user has opted for a floating display (i.e., the record-entry window floats on top of the main index window).

Window Data

In the absence of an object-oriented environment, information needed for managing document windows and their states is kept in a window-specific structure attached to each window, as follows:

Main index window	LFLIST (defined in viewset.h)
Record entry window	MFLIST (defined in modify.c)
Text window	EFLIST (defined in text.h)
Container window	CFLIST (defined in containerset.h)
Record container window	CFLIST (defined in containerset.h)

All these structures have a common first member (WFLIST, defined in indexdocument.h) that contains information about the owning index.

The window structure is initialized when the window is created, and a pointer to it is saved in the window's extra memory, using the setdata() function (util.c). The structure is retrieved with the getdata() inline function (util.h).

Other Top-Level Components

Outside the MDI architecture, the app uses standard Windows dialog boxes to present panels through which the user manages various capabilities and settings (e.g., search and replace, checking spelling, gathering index statistics, sorting entries).

With three exceptions (panels for search, replace, spelling) the panels are presented modally via the Windows function DialogBoxParam(), and require explicit closure by the user.

Although most panels through which the user makes settings or triggers actions operate on the frontmost index, some can operate in two modes:

- If called when an index is open, the panel is presented modally on the frontmost index document and settings apply to that document only.
- If called when no index is open, the panel presents default settings that the app will apply when it creates a new index.

The following menu commands manage dual role panels:

Margins & Columns…	(formset.c, entry point fs_margcol())
Headers & Footers…	(formset.c, entry point fs_headfoot())
Grouping Entries…	(formset.c, entry point fs_entrygroup())

Style & Layout…	(formset.c, entry point fs_stylelayout())
Headings…	(formset.c, entry point fs_headings())
Cross-References…	(formset.c, entry point fs_crossrefs())
Page References…	(formset.c, entry point fs_pagerefs)
Styled Strings…	(formset.c, entry point fs_styledstrings)
Record Structure…	(indexset.c, entry point is_setstruct())
Reference Syntax…	(indexset.c, entry point is_refsyntax())
Smart Flip Words…	(indexset.c, entry point is_flipwords())
Sort…	(sort.c, entry point ss_sortindex())

Settings made via these panels are more fully described later in this document.

Command Dispatch

Most app operations are managed through the main menu and its submenus, which are specified in the resource file cindex.rc.

On launch the app starts WinMain() in main.c, initializes window classes and other core objects, creates the frame window, and then enters the message handling loop where it makes calls to Windows GetMessage(). With the exception of messages sent to any modeless dialogs (the panels that manage Find…, Replace…, Spelling…) messages are sent by DispatchMessage() to the frame window's frameproc() (main.c), and from there are dispensed to relevant components of the app. Menu commands are handled by dofcommand(). This first checks (via com_findcommand() in commands.c) that the command is from a standard menu. If so, it receives a COMGROUP structure (commands.h) that contains information about how the command should be executed, together with flags that specify conditions under which it's enabled, and whether it should be passed to an index window or handled by the frame window process. If the command is intended for an index, and any index is open, it is dispatched to the frontmost index window, and handled by the window's procedure. Otherwise, dofcommand() calls other functions as specified by the particulars of the command.

Menu items and toolbar buttons are enabled and disabled in the context of the active window. When that window receives the WM_INITMENUPOPUP message it calls functions in commands.c to configure (enable, disable, check, or uncheck, etc.) menu items and buttons according to the current context.

Opening Documents

Menu commands for opening and closing documents and for importing and exporting content in other formats are managed by functions in files.c, as follows:

The Open… command calls opendocument() in files.c to present a panel through which the user chooses the document to be opened. After the panel has been closed, fileopen() calls convertdoc() to check the file and if necessary call a conversion function (e.g.,

v3_convertindex()) to convert a document created by an earlier version of the app into the current document format (the structure of saved documents is described in Index Documents). Depending on the document type chosen, the app then calls file_openindex(), file_openarchive(), file_opentemplate(), abbrev_open() or file_openstylesheet()to handle the contents.

Internal Representation of the Index Document

Each index document has an associated master structure (INDEX, defined in indexdocument.h) that holds the information need to manage the index in memory. The file_openindex() function calls file_setupindex() to establish a new INDEX structure and install it in a linked list of open indexes. file_setupindex() then creates a mapped memory representation of the document file via a call to mfile_open() (contained in mfile.c). The resulting configuration is saved in the MFILE structure in the INDEX structure.

After mapping is configured, the index HEAD structure is copied from the mapped memory into the INDEX. The records and groups in an index are not copied. They are always addressed through the mapped image.

A call to installIndex() (in files.c) initializes objects needed by the index, such as the collator for sorting Unicode. To allow for the addition of index records, index_setworkingsize() in index.c is called to create empty space in the mapped image (unused space is removed when the index is closed). The app also scans all the records to check their integrity (index_checkintegrity in index.c). If the checks fail, the app offers remedies as best it can. If checks are passed, the app checks that fonts deployed in the index are available or, if not, have substitutes that are available.

The INDEX is the key internal object through which the Cindex manages almost everything about an index. Pointers to it are widely deployed through the app. When the app changes attributes of an index (record structure, formatting specifications, sorting rules, etc.) the changed attributes are held in the HEAD member of the INDEX.

The INDEX structure is also the vehicle for binding the index document to its windows: the WFLIST member of the window data structure attached to each index window (main window, record-entry window, etc.) has the INDEX structure as its sole member.

Displaying Index Content

After the index is loaded and checked, installIndex() (files.c) calls container_setwindow() to create the main window then calls view_allrecords() in viewset.c. This indirectly calls view_redisplay(), which manages the layout of text on the window. view_redisplay() first identifies the record whose content is to be placed at the top of the screen, then, for every line that will be displayed on the screen constructs the text that should be laid out on it.

Building displayed text from records

The first stage builds an array of entries to be laid out on the screen. For a *draft-formatted* index, the displayed text of an entry conforms closely to what is in the record; for a *fully formatted* index, the displayed text can differ substantially from what is in records: headings can be collapsed, depending on whether the index is indented or run-in, page references are sorted and abbreviated/conflated as needed; cross-references are sorted and placed after page references; formatting punctuation is inserted where required by the chosen layout; formatting codes are inserted where automatic styling is required for specified words in headings, or components of page references.

To construct the screen layout view_redisplay() calls setlines() in viewset.c. For each line of text that can fit on the screen, setlines() maintains information about its contents in a DLINE structure. The array of DLINE structures, covering all lines on the screen, is maintained as the lines member of the window's LFLIST.

Starting with the record that will appear at the top of the screen, then moving sequentially through records, setlines() fills the lines array. It first calls draft_measurelines()or form_measurelines(), depending on the view, to build an entry from the record(s) and determine the number of lines it will occupy on the screen. The draft entry is constructed by buildentry() in draftstuff.c, and the fully formatted entry by form_buildentry() in formstuff.c. Each function consumes as many records as are needed to complete the entry. After building the entry, measurelines() fills an array of LSET structures with information about its line breaks and indents. For each of the LSET structures in array delivered by measurelines(), setlines() packages that and other information about the entry in a member of the lines array.

After setlines() has specified the layout of screen lines, it invalidates the main window's view area. This triggers a call to lupdate() in viewset.c, which, via drawlines(), calls draft_disp() or form_disp() to write the specified number of lines of text to the screen.

Estimating text position within the formatted index

The display in the main index window shows (via scroll bars) the relative position of the text in the index and (via the thumb) the fraction of the index displayed. Similarly, when the user moves the scroll bar to a new position in the window, the app shifts the display to the corresponding position within the index. Without building a full representation of the displayable index it is impossible to know precisely how much space the formatted text occupies, how much of it is displayed on the screen, and where within the full text the displayed segment sits. To manage scrolling the app therefore uses estimates of a) the size of the fully formatted index; b) the fraction of the index displayed on the screen; c) the position within the index of the displayed segment.

When the main window is first opened or the user changes the display format (e.g., from draft-formatted to full-formatted) view_redisplay() calls setscrollrange(), which lays out a

sample of records to estimate how much space the whole index will need, and what fraction of it can be displayed in the window.

Whenever the main display is refreshed (e.g., after the user issues a command, or uses the scrollbar, to display a particular part of it), the app calls setvscroll() in viewset.c. This in turn calls sort_findpos() in sort.c to find the ordinal position of the first displayed record in the sorted index. That position is then used to set the scrollbar thumb.

Publisher's Edition

The Publishers' edition of Cindex extends the capabilities of the standard configuration. The additional capabilities (described in user documents) are enabled by the PUBLISH preprocessor macro. One additional project file (apiservice.c) is required by the Publishers' configuration. It is excluded from the other project builds.

Printing Index Content

On both platforms, printing the index deploys substantially the same code as displaying entries on the screen, but with output sent to a different drawing context, and with adjustments to manage the layout of entries continued across page breaks, and the addition of adornments such as page headers and footers.

On the Mac, printing deploys the NSDocument architecture. The Print… command eventually calls through to printOperationWithSettings:error: in IRIndexDocument. This retrieves a IRIndexPrintView (a subclass of NSView) from the IRIndexDocWController. The IRIndexPrintView then receives calls to rectForPage: to define the rectangle that specifies the part of the whole index view (i.e., the fully formatted document) covered by the specified page. rectForPage: also generates the text for the page, by calling the IRAttributedDisplayString method initWithIRIndex:paragraphs:record: as often as needed to fill the required number of lines.

On Windows, the Print… command results in a call to Iprint() in viewset.c, which sets up the drawing context for printing, then calls formimages() to lay out pages. For each page, formimages() calls setlines() to specify that content of each line on the page, and drawlines() to draw the text to the printing context.

Importing Index Records

Cindex can import records in two distinct forms: XML, in which record content is explicitly structured, and tab- or quote delimited text, in which the fields of each record are contained on a single line. The latter type includes the export or backup (archive) file formats used by other indexing software (Macrex, SKY Index). Platform-specific code opens the document to be imported, then to create records calls core code that is substantially the same on both platforms.

Mac

The command to import records into an existing index calls importRecords: in IRIndexDocument. For importable file types this creates an instance of IRIndexArchive (a

subclass of NSDocument) and calls its initWithContentsOfURL:ofType:forIndex: method. This then calls readFromURL:ofType:error: which in turn calls loadDataRepresentation:ofType: to read file content and create records to be loaded into the existing index. loadDataRepresentation:ofType: identifies the type of import and calls type-specific functions to read data from the file.

Windows

The menu command to import records into an existing index calls mc_import() in files.c. After capturing the document specification this calls imp_loaddata() in import.c.

Core Code

From loadDataRepresentation:ofType: (Mac) or imp_loaddata() (Windows) XML content and delimited content are handled by two different code paths. XML is parsed and loaded by functions in xmlparser.m (xmlparser.c); delimited content is parsed and loaded by functions in import.m (import.c).

import.m/import.c contains code to manage multiple delimited formats (identified in import.h). These vary in character encoding (some use UTF-8 Unicode, others various older 8-bit character sets) and in the way they represent text attributes (fonts, styles). In the function imp_readrecords()(Mac), or readrecords() (Windows) the app sets up a character code converter if the source is not UTF-8, then for every line in the file calls the function checkline() to scan for errors and convert the text into the standard representation used in a RECORD. If there are no errors, or errors are resolved, imp_readrecords()/readrecords() makes a second pass through the file, this time capturing each line and creating a record from it.

Exporting Formatted Indexes

Cindex can export formatted indexes in a variety of formats. After executing platform-specific code to obtain information about the document to be produced, the app calls core code that is substantially the same on both platforms.

Mac

The IRIndexDocument instance manages document export, through calls to standard methods implemented by the NSDocument architecture. The Save To… command calls runModalSavePanelForSaveOperation:delegate:didSaveSelector:contextInfo: to obtain information about the document type and its destination. The app then calls saveToURL:ofType:forSaveOperation:completionHandler: which calls writeToURL:ofType:error: which in turn calls dataOfType:error: to create an NSData object to be saved. dataOfType:error: calls through to formexport_write() in formattedexport.m to generate the formatted text and return it as a text buffer wrapped in NSData.

Windows

The command to export a formatted index calls exp_export() in export.c. This in turn calls through to formexport_write() in formattedexport.c to open a memory-mapped file then lay out the formatted text in the file buffer.

Core Code

Code to produce the formatted text for export is substantially identical on both platforms. The key entry point is the call to formexport_write(). This function is passed pointers to EXPORTPARAMS and FCONTROLX structures. EXPORTPARAMS, defined in export.h, specifies which records (or formatted pages) are to be written. FCONTROLX, defined in indexdocument.h contains the information required to build index entries in the chosen format (RTF, InDesign, etc.). For each file format in which Cindex can export an index, there is a statically defined instance of FCONTROLX with fields set to format-specific values. A pointer to the relevant static instance of FCONTROLX is passed in the call to formexport_write(). The format-specific instances of FCONTROLX, together with functions that are called to initialize settings and cleanup when writing is complete, are in the following files:

Plain Text	textwriter.m (textwriter.c)
Rich Text Format	rtfwriter.m (rtfwriter.c)
QuarkXPress	xpresswriter.m (expresswriter.c)
InDesign	indesign.m (indesign.c)
Index-Manager	imwriter.m (imwriter.c)
Tagged Text	taggedtextwriter.m (taggedtextwriter.c)

For each record in the range to be exported, formexport_write()calls formexport_buildentry() to build the formatted entry and append it to the text buffer that will ultimately be saved.

Among the export formats, Tagged Text is special because the tag sets can be user-defined. The structure of a tag set is described under Tag Sets. Tag sets are configured through the Markup Tags… command in the Tools menu.

Adding and Editing Records

Mac

Record content is created or edited via the IRIndexRecordWController, which presents the editing window on the screen. A new or existing record is opened for editing via a call to openRecord:. This in turn calls _setRecord:, which retrieves the specified record and conditions its text, as needed, before calling setText: on the window's IRRecordView. setText: creates an NSAttributedString from the record text and loads it as the content of

the view. The IRRecordView handles editing commands and keystrokes, calling various methods as needed.

When editing is finished (ether by the user passing on to another record, or closing the window), the IRIndexRecordWController calls canAbandonRecord. Via the getText: method in IRRecordView this recovers the record text and checks whether it is changed. If text is changed (canAbandonRecord returns FALSE), canCompleteRecord is called to check for errors; if none are found the original record text is replaced with the edited text.

For each edited record, the app calls sort_addtolist() in sort.m to save its record number. Whenever the editing window becomes inactive, or is closed, the app calls sort_resortlist() to replace the sort-tree entries for the edited records.

Windows

Record content is created or edited via code in modify.c. The command to open a new or existing record calls mod_settext(), which then calls mod_setwindow() to create and place the editing window on the screen. Within that window, editable text is managed by a Rich Edit control. mod_settext() calls loadfields() to retrieve the specified record and condition its text, as needed. loadfields() in turn calls settestring() to install the text and attributes in the edit control.

When editing is finished (ether by the user passing on to another record, or closing the window), mod_canenterrecord() is called. This in turn calls canabandon(), which calls gettestring() to recover the record text and check whether it has changed. If text is changed (canabandon()returns FALSE), mod_canenterrecord() calls enterrecord() to check for errors; if none are found the original record text is replaced with the edited text.

For each edited record, the app calls sort_addtolist() in sort.c to save its record number. When the editing window is closed, the app calls sort_resortlist() to replace the sort-tree entries for the edited records.

Sorting

The fundamental sorting operation compares the contents of two records on some sorting rule to establish whether they match (are equal), or, if not, the direction of mismatch (inequality).

With variations that are discussed below, the sorting rule is simple: it runs through record fields sequentially, comparing their text strings in lexicographical (dictionary) order, until it finds a mismatch. If two records match in all their text fields, the app compares the contents of the locator fields, as text strings if the locator fields contain cross-references, and as sequences of numerical locators if they contain page references. This sequence of tests is implemented in the match() function in sort.m or sort.c.

There are many ways of lexicographically ordering text, especially in the context of sorting index entries. Some arise from language differences (e.g., in Swedish Å appears after Z), others from varied conventions that specify the values of accented letters, or whether punctuation should be attended to or ignored. The user tailors the ordering via the Sort… command in the Tools menu.

The user's Sort settings give rise to a specific sort rule that controls the evaluation of character sequences in record fields. Within the match() function, the tailored evaluation of characters in each record field is undertaken by the col_match() function in collate.m or collate.c. Supporting code in collate.c filters the text per the user's chosen sort rule, then passes the filtered string for evaluation by the Unicode collating functions provided by ICU.

Sorting the Whole Index

The app keeps index records permanently sorted, by whatever sorting rule (e.g., ascending alphabetical order) the user has specified.

Cindex uses an AVL tree to manage the sorted order of records. This supports three key capabilities:

1. Rapid lookup (search). The app can effectively instantly establish whether a record with specified content exists in the index.
2. Rapid traversal. The app can very quickly traverse the records in sorted order, regardless of their order of entry in the index.
3. Rapid insertion/deletion. A record can very quickly be added to, or removed from, the sorted sequence.

The core code for managing lookup, traversal, and insertion/deletion is contained in the module sort.m (Mac) or the essentially identical sort.c (Windows). The top level function for sorting the whole index is sort_resort().

The header of each index record contains three member variables (parent, lchild, rchild) set by the sorting code to specify the record's position in the index. This information is always kept up to date as records are added to the index or edited but can be ignored when needed—for example when the user wants to view index records in their order of entry in the index, or in a group.

Sorting Groups

When a user creates a group of records (see earlier discussion of Groups), the app holds a list of the group's records in the order of their appearance in the index when the group was formed. The user can sort the group independently of the whole index by using the Sort… command when only the group is visible. When sorting a group the app does not touch the main sort tree. Instead, via sort_sortgroup(), it uses the standard C qsort() function to sort the record identifiers in the group's list, producing a reordered

list as a result. All the information about the sorted order of records in a group is maintained within the group structure; records themselves are not touched.

The sorted order of records in a group is not updated automatically when a user edits its records. However, the main sort tree is kept up to date even when the user is working with a group, so records always appear in their correct positions when the full index view is restored.

Searching

Cindex supports two distinct types of search for record content. The first exploits the structure of the sort tree to lookup records that *begin* with specified text (or, if the index is sorted by the values of in the locator field, records that contain specified locators). A search of this kind is, under most circumstances, effectively instantaneous. The second kind of search scans the index to find records that contain (or do not contain) one or more attributes that can be richly specified (e.g., records that contain a particular phrase *and* were edited by a particular user *or* were created after a specified date).

Lookup

When the main index window is active and the user starts typing, the app passes characters to com_findrecord() in commandutils.m or commandutils.c. If the user entered a number, this then calls search_findbynumber() to find the record with that number; if instead the user typed what might be the starting text of a record com_findrecord() calls search_findbylead(). This then queries the sort tree if all index records are on view, or triggers a binary search if a group is on view.

Exhaustive Search

When searching for content or attributes unrelated to sort keys, the app scans index records sequentially. The search specification can be rich, covering the text content and many other attributes of records. The user constructs the search specification via the panel displayed by the Find… command. The Find… panel is managed by a FindController (Mac), and by code in findset.c (Windows—entry point find_setwindow()). This places the search specification in a LISTGROUP structure (searchparams.h), calls checkFindValid (Mac) or fcheck() (Windows) to validate the contents, then calls search_findfirst(), which runs through records to be searched, passing each to search_findbycontent()until it finds one that satisfies the specification in the LISTGROUP.

Replacing

Cindex supports global search-and-replace via the Replace… command. This presents a Replace panel that is managed by a ReplaceController (Mac) and by code in replaceset.c (Windows—entry point rep_setwindow()). The search side of the operation draws substantially on code also used by the Find… command: on the Mac ReplaceController and FindController are both subclasses of SearchController; on Windows, search functions in findset.c are called from code in replaceset.c. Both platforms use the LISTGROUP

structure to specify search parameters. Panel management code places the replacement specification in a REPLACEGROUP structure, defined in searchparams.h.

After validating the search specification with checkFindValid (Mac) or fcheck() (Windows), the panel management code calls through to search_findfirst() to find the first record that matches the search specification.

If valid target text/attributes are found, the panel management code then builds and validates the replacement text (which may include some of the target text) via a call to repset() (Mac) or rep_setup() (Windows). It then calls search_reptext() in search.m (search.c) to make the replacement. For each record in which text is replaced, the app calls sort_addtolist() in sort.m (sort.c) to save its record number. After all replacements have been made the app calls sort_resortlist() to replace the sort-tree entries for the changed records.

Spelling

Cindex uses the open-source spelling checker Hunspell. Hunspell is widely deployed in other applications (e.g., Firefox) and as a result dictionaries for many languages are freely available.

The user checks spelling via the panel displayed by the Spelling… command. Code that manages the Spell… panel (for Mac, in SpellController.m; for Windows, in spellset.c) places the search specification in a SPELL structure (spell.h), validates the settings, then calls sp_findfirst() in spell.m or spell.c to find the first record that contains an unknown word. sp_findfirst() calls sp_checktext() to evaluate record contents. This in turn parses fields and then individual words, passing each word in turn to hspell_spellword() in hspell.m or hspell.c. hspell_spellword() calls out to the Hunspell library to look up the word.

Dictionaries

Cindex is supplied with an installed Hunspell dictionary for US English. Users can add dictionaries for other languages, or for special (e.g., medical) terms, and can create personal dictionaries that are used in addition to the current language dictionary. The *User's Guide* explains how and where to install dictionaries.

When the Spelling panel is first opened, the app calls hspell_init() in hspell.m (hspell.c), which looks in specified places for language and other special dictionaries, and makes a list of those that are available for the user's currently selected language. The app then loads the user's preferred language dictionary along with any user-created personal dictionaries, and displays those available in the Spell… panel.

Document Settings

Settings that control the structure of the index, and its appearance on the screen or when printed, are made via a set of panels provided through the Document menu.

For all the items in this menu, the user can make settings in two contexts:

- If no index is active, the settings are saved as preferences (in the prefs structure, defined in globals.h) and become the defaults applied to any new index the user creates.
- If any index is open, the settings apply to the frontmost one, and are saved in the HEAD structure of the index document (see Index Documents).

Margins & Columns

Settings are managed by the panel displayed by the Margins & Columns… command. Code to manage the panel is in MarginColumnController.m (Mac), and formset.c (Windows—entry point fs_margcol()). The panel places the settings in a MARGINCOLUMN structure, which in turn is contained in a PAGEFORMAT structure, contained in a FORMATPARAMS structure. These structures are defined in formatparams.h.

Headers & Footers

Settings are managed by the panel displayed by the Headers & Footers… command. Code to manage the panel is in HeadFootController.m (Mac), and formset.c (Windows—entry point fs_headfoot()). Depending on whether the user has set 'Facing Pages' via Margins and Columns… settings, the Headers & Footers… panel displays 2 or 4 tabs, each of which contains settings for a header or a footer. The settings from each tab are saved in a HEADERFOOTER structure, which in turn is contained in a PAGEFORMAT structure, contained in a FORMATPARAMS structure. These structures are defined in formatparams.h.

Grouping Entries

These settings control how entries are grouped in the formatted display. They are managed through the panel displayed by the Grouping Entries… command. Code to manage the panel is in GroupingController.m (Mac), and formset.c (Windows—entry point fs_entrygroup()). The panel places the settings in a GROUPFORMAT structure, which in turn is contained in a ENTRYFORMAT structure, contained in a FORMATPARAMS structure. These structures are defined in formatparams.h.

Style & Layout

These settings control the appearance of individual entries in the formatted display. Settings are managed by the panel displayed by the Style & Layout… command. Code to manage the panel is in StyleColumnController.m (Mac), and formset.c (Windows—entry point fs_stylelayout). The panel places the settings in elements of an ENTRYFORMAT structure, which in turn is contained in a FORMATPARAMS structure. These structures are defined in formatparams.h.

Headings

These settings control the text attributes of different levels of heading in the formatted display. Settings are managed by the panel displayed by the Headings… command. Code to manage the panel is in HeadingsController.m (Mac), and formset.c (Windows—entry point fs_headings). For each level of heading, the panel places the settings in a

FIELDFORMAT structure. An array of FIELDFORMAT structures (one for each possible level of heading) is in turn contained in an ENTRYFORMAT structure, contained in a FORMATPARAMS structure. These structures are defined in formatparams.h.

Cross-References

These settings control the placement and text attributes of cross-references in the formatted display. Settings are managed by the panel displayed by the Cross-References... command. Code to manage the panel is in CrossRefsController.m (Mac), and formset.c (Windows—entry point fs_crossrefs()). The panel places the settings in a CROSSREFFORMAT structure, which in turn is contained in an ENTRYFORMAT structure, contained in a FORMATPARAMS structure. These structures are defined in formatparams.h.

Page References

These settings control the formatting of locators (page references) in the formatted display. Settings are managed by the panel displayed by the Page References... command. Code to manage the panel is in PageRefsController.m (Mac), and formset.c (Windows—entry point fs_pagerefs()). The panel places the settings in a LOCATORFORMAT structure, which in turn is contained in an ENTRYFORMAT structure, contained in a FORMATPARAMS structure. These structures are defined in formatparams.h.

Styled Strings

This setting specifies words and phrases that, if found in records, should be rendered in particular text style(s) when displaying formatted entries. Settings are managed by the panel displayed by the Styled Strings... command. Code to manage the panel is in StyledStringsController.m (Mac), and formset.c (Windows—entry point fs_styledstrings()). For each text/attribute pair the panel code constructs a string in which the first byte contains style attributes, and the remainder is the text content (utf8). The strings containing each text/attribute pair are concatenated into a compound string (xstring). If the setting is made for the active index, the xstring is saved in the stylestrings element of the index HEAD structure; if the user is making a default setting, the xstring is saved as the stylestrings element of the prefs structure (defined in globals.h).

Record Structure

These settings specify basic attributes of index records, and particular attributes of individual record fields. Settings are managed by the panel displayed by the Record Structure... command. Code to manage the panel is in RecordStructureController.m (Mac), and indexset.c (Windows—entry point is_setstruct()). The panel places settings for individual fields in a set of FIELDPARAMS structures; the array of these is in turn contained in an INDEXPARAMS structure. These structures are defined in indexparams.h.

Reference Syntax

These settings specify how Cindex should recognize cross-references and page references in index records, and (in the case of page references) limitations on permissible values. Settings are managed by the panel displayed by the Reference

Syntax… command. Code to manage the panel is in RefSyntaxController.m (Mac), and indexset.c (Windows—entry point is_setstruct()). The panel places settings for individual fields in a set of FIELDPARAMS structures; the array of these is in turn contained in an INDEXPARAMS structure. These structures are defined in indexparams.h.

Smart Flip Words

This setting specifies an array of leading and trailing articles/prepositions that Cindex can move within a record field when it exchanges (flips) headings in records. (The same list of words is also used by the Check Index… command when checking the consistency of prepositions/conjunctions in headings). Settings are managed by the panel displayed by the Smart Flip Words… command. Code to manage the panel is in FlipStringsController.m (Mac), and indexset.c (Windows—entry point is_flipwords()). The set of words is contained in a single string (utf8). If the setting is made for the active index, the string is saved in the flipwords member of the index HEAD structure; if the user is making a default setting, the xtring is saved as the flipwords member of the prefs structure (defined in globals.h).

Tools

Check Index

This command scans the index looking for possible errors. The app presents a panel through which the user specifies the checks to be made. Settings the user makes are automatically saved as preferences, and are restored when the panel is next displayed. Code to manage the panel is in CheckController.m (Mac), and toolset.c (Windows—entry point ts_checkindex()). User's settings are placed in a CHECKPARAMS structure (defined in tools.h). This structure is passed to the function tool_check() in tools.m (tools.c), which runs the checks and returns results in the CHECKERRORS array in the CHECKPARAMS structure.

The Check Index command is new in Cindex 4.1. It consolidates and extends some checks that were available in previous versions of Cindex. As a result, checks are managed by disparate segments of code, called as needed from within tool_check().

- Checks on the integrity of headings are undertaken directly by tool_check().
- Checks on page references (counts and overlaps) require tool_check() to run through the index building fully formatted entries.
- Cross-references are verified by a call to search_verify() in search.m (search.c).
- Orphaned subheadings are detected by a call to tool_join() in tools.m (tools.c).

Reconcile Headings

This command checks whether index entries have unnecessary levels of subheading, or headings have qualifying phrases that would better be organized as subheadings. The app presents a panel through which the user specifies the details of the task. Code to manage the panel is in ReconcileController.m (Mac), and toolset.c (Windows—entry point

ts_reconcile()). User's settings are placed in a JOINPARAMS structure (defined in tools.h). This structure is passed to the function tool_join() in tools.m (tools.c), which runs the operation.

Manage Cross-References

This tool can do two things: it can generate cross-references automatically by consulting an authority file, and it can convert cross-references into fully qualified entries. The app presents a panel through which the user specifies the required operation. Code to manage the panel is in GenerateRefsController.m (Mac), and toolset.c (Windows—entry point ts_managecrossrefs()). When selection is complete, the app calls search_autogen()if generating cross-references, or search_convertcross() if converting cross-references to fully qualified entries. Both functions are in search.m (search.c).

Alter References

This command adjusts page references, or removes them, to accommodate changes in pagination of the work being indexed. The app presents a panel through which the user specifies the adjustment to be made. Code to manage the panel is in AlterRefsController.m (Mac), and toolset.c (Windows—entry point ts_adjustrefsrefs()). User's settings are placed in an adjstruct structure (refs.h) and that is passed to the function ref_adjust() in refs.m (refs.c) to complete the operation.

Split Headings

This command deploys a user-specified rule to break the contents of individual records into multiple records. The app presents a panel through which the user specifies the adjustment to be made. Code to manage the panel is in SplitController.m (Mac), and toolset.c (Windows—entry point ts_splitrefs()). User's settings are placed in a SPLITPARAMS structure (tool.h) and that is passed to the function tool_explode() in tools.m (tools.c) to complete the operation.

Sort

This command sorts (collates) index entries by user-specified rules. The app presents a panel through which the user chooses the sort rules. (The command is also available when no index is open for use, in which case settings are simply saved as defaults for new indexes when they are created.) Code to manage the panel is in SortController.m (Mac), and sortset.c (Windows—entry point ss_sortindex()).

When settings are complete, the app calls col_init() in collate.m (collate.c) to initialize the Unicode collation code with the new settings, then (if the whole index is on view) calls sort_resort() to resort the index, or sort_sortgroup(), if a group of entries is on view. Both functions are in sort.m (sort.c).

For fuller information on how entries are sorted, see the earlier discussion under Sorting.

Compress

This command minimizes the size of the index by combining duplicate records and removing unneeded ones. The app presents a panel through which the user chooses which kinds of records to remove. Code to manage the panel is in CompressController.m (Mac), and sortset.c (Windows—entry point ss_compress()). Flags set by the user are passed to the sort_squeeze() function in sort.m (sort.c) to complete the operation.

Expand

This command duplicates and modifies records as needed to ensure that each record contains only a single page reference or cross-reference. This command is dispatched through IRIndexDocumentController (Mac), and sortset.c (Windows—entry point ss_expand()), which calls sort_squeeze() in sort.m (sort.c) with an appropriate flag.

Count Records

This command displays basic information about records in the index. The app presents a panel through which the user chooses record attributes of interest. Code to manage the panel is in CountController.m (Mac), and toolset.c (Windows—entry point ts_count()). Flags set by the user are passed to the search_count() function in search.m (search.c) to complete the operation.

Index Statistics

This command displays a statistical summary of the index, including information about entries and references, and space they consume when they are laid out on pages. The command requires Cindex to create a fully formatted index as if printed. On the Mac this can be slow. Code to manage the panel and the production of statistics is in StatisticsController.m (Mac), and toolset.c (Windows—entry point ts_statistics()).

Abbreviations

Users can insert arbitrary phrases in records by typing abbreviations that represent them. Users can define abbreviations directly while entering text in records, or through the panel displayed by the Abbreviations… command. Through the panel users can also modify existing abbreviations, and load, save and delete sets of them. Code to manage the panel is in AbbreviationController.m (Mac), and abbrev.c (Windows—entry point abbrev_view()[??]). On the Mac , abbreviations are saved in an NSDictionary (the abbreviation as key, the expansion as value) in standardUserDefaults; on Windows, abbreviations and their expansions are saved as tab-separated components on single lines in a text file.

Function Keys (Mac)/Hot Keys (Windows)

Function keys and Hot keys enable the user to enter preset text in records with a single keystroke or keystroke combination.

On the Mac, code to manage the panel and assign text to keys is in FunctionKeyController.m. The user's key settings are placed in an NSDictionary, and saved in standardUserDefaults.

On Windows, code to manage hot keys and their associated text is contained in keytool.c. The key specifications and their associated strings are saved respectively in the hotkeys and keystrings members of the global prefs structure (see Preferences).

Fonts

The Fonts panel enables the user to check which fonts are used in the index, to specify alternate fonts to be used when a specified font is unavailable, and to cleanup unused fonts (if any). The panel shows each font used in the index and an alternative to be used when the preferred font is not available. Code to manage the panel and font changes is in ManageFontController.m (Mac), and toolset.c (Windows—entry point ts_fontsub()).

Each index can use up to 32 different typefaces in records and in other displayable entities such as headings to groups of formatted entries. Each font used is managed through a FONTMAP structure, defined in fontmap.h. The fm member of the index HEAD structure is the array of 32 FONTMAPs. The first entry in this array (index 0) contains the default font in which index entries are displayed.

Within records, a font is specified by an introductory byte (0X1A) followed by its index in the fm array (see Organization of Text in a Record). Whenever the user selects a font while editing record text, or specifies a font while using Replace…, the app checks whether the font is already in the fm array, and adds it if not. Functions for managing fonts (checking use, remapping, etc.) are in type.c (Mac) and typestuff.c (Windows).

Markup Tags

This command enables the user to manage sets of tags for marking up text in formatted indexes. Code to manage the panel is in MarkupTagsController.m (Mac), and tagstuff.c (Windows—entry point ts_managetags()). Settings are saved in a TAGSET structure, defined in tags.h. The organization of TAGSET is described under Tag Sets.

The Markup Tags panel provides for the separate management of XML tag sets and SGML tag sets. For each type of tag (XML or SGML) the currently selected tag set is saved in standardUserDefaults (Mac) or in the Registry (Windows). It then becomes the tag set deployed when the user exports a formatted index as an XML document or SGML document (See Exporting Formatted Indexes).

Groups

This command enables the user to manage saved groups (see the earlier discussion of Record Groups). Code to manage the panel is in GroupsController.m (Mac), and edit.c (Windows—entry point edit_managegroups()). Operations selected by the user (e.g., revising or deleting, etc.) are handled by relevant functions in group.m (group.c).

5. Installers and Updaters

This section describes the machinery for installing Cindex on Mac and Windows platforms, for registering users, and for updating installed apps.

Mac

Installer

Before Release of Version 4.1.5

Until recently, Cindex used an installer built using Apple's now obsolete PackageMaker. PackageMaker was originally adopted for building the installer because during installation it enabled creation of a folder (in ~/Library/Application Support/Cindex) in which the app placed the user's registration information (the Application Support folder is not normally writable by apps).

The last Cindex 4.1 installer package dates from April 2020, and was shipped until March 2022. This installer is not code-signed, so users need to override standard MacOS protections to complete the installation. After installation, the app automatically looks for the most recent update (currently 4.1.5) and offers to install it. See below on Updater.

There are three installers for MacOS:

- Cindex Installer.pkg [standard version of Cindex]
- Cindex Student Installer.pkg [student version of Cindex]
- Cindex Demo Installer.pkg [demo version of Cindex]

Each installs the relevant copy of Cindex.

Since Version 4.1.5

Cindex is supplied on a DMG disk image, from which the user can install the app simply by dragging its icon onto the Applications folder icon. There are three disk images, for the standard, student, and demo versions of Cindex.

The app is not code-signed, so requires user authentication when first run.

User Registration

When launched, the app presents a splash window via SplashWindowController. In its awakeFromNib method this checks whether the app is already registered with a valid serial number. If so, the launch completes. If not, awakeFromNib presents a panel through which the app collects registration information (user's name, company, and license).

The user-visible license is a simple encoded version of a serial number. When the registration information is entered, the app attempts to decode the license by calling

the c function checkserial() in SplashWindowController.[2] If successful, it saves the information in a hidden file in /Library/Application Support/Cindex, from which it retrieves the information on subsequent launches of the app.

Updater

The Mac version of Cindex uses Sparkle to manage updating. This is included as a framework and is configured per Sparkle installation instructions. Via Preferences (General tab), Cindex allows the user to configure automatic checks for updates, and set the frequency of checks. The app also allows the user (via the Cindex menu) to check manually for an update. There is no explicit code to manage updating—it is all handled by the Sparke framework and its integration via the app's info.plist.

Only the full version of Cindex can be updated. Student and Demo versions of Cindex cannot be updated.

Updater files are retrieved from the Indexing Research server (see additional note at the end of this document).

Windows

Installer

The app installer (CindexSetup.exe) is built using the free application Inno Setup. The installer is created from a script that specifies the set of files to be installed, Registry keys to be created for the file types and clipboard data types the app supports, and other app configuration details. The script also includes a fragment of Pascal code to validate the user's license information. The current installer is code-signed by Frances Lennie.

Installer scripts, their sets of input files, and built installers are contained in the Installer folder in the Windows archive.

Installer Versions

Four different installer scripts produce the following flavors of Cindex:

- CindexSetup.exe [standard version of Cindex]
- CindexPublisherSetup.exe [publisher's edition of Cindex]
- CindexStudentSetup.exe [student version of Cindex]
- CindexDemoSetup.exe [demo version of Cindex]

Installed Student and Demo versions of Cindex cannot be updated.

User Registration

User registration is managed as an integral part of installation. During setup, the installer presents a panel through which the user enters registration details (user's name,

[2] We use a simple MacOS app to generate user-facing license strings for serial numbers (both Mac and Windows). That app, and its source code, are included with the Cindex code delivery.

company, and license). The installer includes a small script to decode and validate the license. If the license is successfully validated, the installer saves the registration information in the Windows Registry. If the license cannot be validated, installation fails.

Updater

The Windows version of Cindex uses WinSparkle to manage updates. WinSparkle (modeled on Sparkle for MacOS) is included as a dll in the installer package. Via Preferences (General tab), Cindex allows the user to configure automatic checks for updates. The app also allows the user to check manually for an update, via the Help menu. Code to configure WinSparkle updating is contained in the WinMain() function in main.c.

Only the Standard and Publishers' versions of Cindex can be updated. Student and Demo versions of Cindex cannot be updated.

The updater files are retrieved from the Indexing Research server (see next section).

Delivering Updates

When triggered by a routine check, or manually by the user, the Mac and Windows versions of Cindex interrogate different versions of a small file (appcast.xml) downloaded from the Indexing Research server. This file contains information about the most recent version of Cindex. If the most recent version is not the version that requested appcast.xml, the app presents the user with an alert, and information about what is new, then offers to download and install the update.

If the user approves, the update is downloaded from a URL specified in appcast.xml, then installed.

> **Note:** URLs for the different versions of appcast.xml (but not URLs of additional files required for updating) are built into the distributed apps.

Different updater file sets are maintained for the different versions of Cindex, as follows:

Mac

The app looks initially for http://www.indexres.com/CinMac4/appcast.xml. This file in turn contains URLs to two additional components:

1. An HTML file that provides a description of changes http://www.indexres.com/CinMac4/UpdateNotes.html
2. A zip archive that contains a new version of the app: http://www.indexres.com/CinMac4/Cindex.zip

Windows (Standard Edition)

The app looks initially for http://www.indexres.com/CinWin4/appcast.xml. This file in turn contains URLs to two additional components:

1. An HTML file that provides a description of changes
   http://www.indexres.com/CinWin4/UpdateNotes.html
2. An installer (code-signed) that contains updated components of the app:
   http://www.indexres.com/CinWin4/CindexPublisherUpdater.exe

Windows (Publishers' Edition)

The app looks initially for http://www.indexres.com/CinWinPub4/appcast.xml. This file in turn contains URLs to two additional components:

1. An HTML file that provides a description of changes
   http://www.indexres.com/CinWinPub4/UpdateNotes.html
2. An installer (code-signed) that contains updated components of the app:
   http://www.indexres.com/CinWinPub4/CindexPublisherUpdater.exe

Upgrades

The app provides a rudimentary mechanism for informing the user about the availability of an upgrade (i.e., a new version of Cindex that can be purchased). When the app is launched it calls HTTPService (Mac) or httpservice() in httpservice.c (Windows) to read a file (versionupdate.json) from the indexres server. The app looks for this file in the following fixed location:

https://storage.googleapis.com/indexres-d3231811-9b04-47b0-8756-5da84afef700/downloads/versionupdate.json

versionupdate.json contains a json dictionary with information that, if current, is displayed in an alert. Here is a sample dictionary:


```
{
    "minVersion": 300,
    "messageTitle": "A New Version Of Cindex Is Now Available",
    "endDate": "2019-9-30",
    "startDate": "2019-8-20",
    "message": "Cindex version 4, containing many new features, is now available from Indexing Research as an upgrade. Would you like to learn more?",
    "versionType": 7,
    "maxVersion": 309,
    "messageType": 0,
    "url": "https://www.indexres.com/resources/cindex-4-features"
}
```

Appendix 1. Mac Code Files

AbbreviationController.m	Manages the Abbreviations panel (Tools menu).
AlterRefsController.m	Manages the Alter References panel (Tools menu).
AttributedStringCategories.m	Extends NSAttributedString string with methods to manage font conversions.
attributedstrings.m	Functions to compose and decompose a compound string of record text into an ATTRIBUTEDSTRING object that separates text from font and style attributes.
CheckController.m	Manages the Check Index panel (Tools menu).
collate.m	Functions to extract text strings from records, condition them, then compare them using the Unicode collation rules.
commandutils.m	Manages error and warning messages + support functions for identifying dates and record ranges.
CompressController.m	Manages the Compress panel (Tools menu)
CountController.m	Manages the Count Records panel (Tools menu)
CrossRefsController.m	Manages the Cross-references panel (Document menu).
drafttext.m	Functions to create draft formatted entries from records.
export.m	Functions to export index records.
ExportOptionsController.m	Manages the Options panel (available in the Save panel) when using the Save To command.
FilterController.m	Manages the Hide by Attribute panel (View menu).
FindController.m	Manages the Find panel (Edit menu). Subclass of SearchController.
FlipStringsController.m	Manages the Smart Flip Word panel (Document menu).
formattedexport.m	Functions to export a formatted index.
formattedtext.m	Functions to create fully formatted entries from records.
FunctionkeyController.m	Manages the Function keys panel (Tools menu).

GenerateRefsController.m	Manages the Manage Cross-References pane (Tools menu).
globals.m	Functions to load and save preferences.
GoToController.m	Manages the Go To panel (View menu).
group.m	Functions to manage record groups.
GroupingController.m	Manages the Grouping Entries panel (Document menu).
GroupsController.m	Manages the Groups panel (Tools menu).
HeadFootController.m	Manages the Headers & Footers panel (Document menu).
HeadingsController.m	Manages the Headings panel (Document menu).
hspell.m	Functions to interface with the hunspell library.
HTTPService.m	Connects to the indexres server to download and display an alert about a new version of Cindex.
import.m	Functions to import index records.
imwriter.m	Functions to prepare content for export to Index-Manager.
indesign.m	Functions to prepare content for export to InDesign.
index.m	Miscellaneous functions to check integrity/repair the index, set its size in memory, read and write headers, flush records.
IRAbbreviations.m	Unused.
IRApp.m	Application main class (subclass of NSApplication).
IRAttributedDisplayString.m	Subclass of NSAttributedString that wraps an IRDisplayString and exposes the text and attributes of its contained ATTRIBUTEDSTRING.
IRDisplayString.m	Subclass of NSString that holds its content as an ATTRIBUTEDSTRING object.
IRIndexArchive.m	Manages the import of index archive documents.
IRIndexDocument.m	Provides overall management of the index document (subclass of NSDocument).

IRIndexDocumentController.m	Application delegate (subclass of NSDocumentController). Manages all index documents.
IRIndexDocWController.m	Manages the main index window (subclass of NSWindowController).
IRIndexPrintView.m	Manages the view into which index content is printed (subclass of NSView).
IRIndexRecordWController.m	Manages the record-entry window (subclass of NSWindowController).
IRIndexTextWController.m	Manages the auxiliary text display window associated with the index (subclass of NSWindowController).
IRIndexView.m	Manages the view in which index content is displayed in the main window (subclass of NSView).
IRPrintAccessoryController.m	Manages Cindex-specific elements of the standard Print panel.
IRRecordView.m	Manages the view within which record text is displayed in the editing window (subclass of NSView).
IRTableHeaderView.m	Provides a method to manage table headers for AbbreviationController and FunctionKeyController (subclass of NSTableHeaderView).
IRTextView.m	Manages the view in which text is displayed in the index auxiliary text window (subclass of NSView).
IRToolbarItem.m	Provides a validation method for toolbar items in the main index window and the record entry window (subclass of NSToolbarItem).
LayoutDescriptor.m	Class that takes extracts various attributes of the text displayed on the screen, including how it is laid out on screen, and the index records that gave rise to it.
locales.m	Functions for retrieving information about the user's locale (country).
main.m	Program entry point.
ManageFontController.m	Manages the Fonts panel (Tools menu).
MarginColumnController.m	Manages the Margins & Columns panel (Document menu).

MarkupTagsController.m	Manages the Markup Tags panel (Tools menu).
MessageController.m	Manages the panel through which a message about an available upgrade is displayed.
mfile.m	Functions for managing memory mapping of files.
NSArray+NSArray.m	Method that extends NSArray to manage sorting.
NSColor+NSColor.m	Method that extends NSColor for managing accent colors (MacOS 10.14 and higher).
NSDocument+NSDocument.m	Empty category used only to provide a header to declare the changeSaveType method.
NSMutableAttributedString+NSMutableAttributedString.m	Methods to extend NSAttributedString to manage tab stops.
PageRefsController.m	Manages the Page References panel (Document menu).
PreferencesController.m	Manages the Preferences panel (Cindex menu).
ReconcileController.m	Manages the Reconcile Headings panel (Tools menu).
records.m	Functions for retrieving and updating records, and manipulating record text.
RecordStructureController.m	Manages the Record Structure panel (Document menu).
refs.m	Functions for managing page references within the locator field of a record.
RefSyntaxController.m	Manages the Reference Syntax panel (Document menu).
regex.m	Functions for forming regular expressions, and finding and replacing matched text.
ReplaceController.m	Manages the Replace panel (Edit menu). Subclass of SearchController.
ReplaceFontController.m	Manages the font replacement panel (subpanel of Replace panel).
rtfwriter.m	Functions to prepare content for export as Rich Text Format.
SaveGroupController.m	Manages the Save Group panel (File menu).

search.m	Functions for searching for content in records, for replacing found content, and for verifying, converting, and generating cross-references.
SearchController.m	Superclass of FindController and ReplaceController, providing common functionality
sort.m	Functions for managing the AVL sort tree. Also contains the function for compressing records.
SortController.m	Manages the Sort panel (Tools menu).
spell.m	Functions for checking spelling and replacing misspelled words.
SpellController.m	Manages the Spelling panel (Edit menu).
SpellTextField.m	Subclass of NSTextfield used in Spell panel. Provides special handling of mouse click.
SplashWindow.m	Splash window (subclass of NSWindow). Managed by SplashWindowController.
SplashWindowController.m	Manages splash display on startup. Also presents registration panel if needed.
SplitController.m	Manages Split Headings panel (Tools menu).
StatisticsController.m	Manages Index Statistics panel (Tools menu).
StorageCategories.m	Methods that extend NSTextStorage. The linesForRange: method is used when laying out a print view.
StringCategories.m	Methods that extend NSString.
strings_c.m	Miscellaneous functions for manipulating strings and compound string (xstrings).
StyledStringsController.m	Manages the Styled String panel (Document menu).
StyleLayoutController.m	Manages the Style & Layout panel (Document menu).
swap.m	Functions to swap byte alignment of objects created on different processor architectures. Used in converting old index files to version 3/4 format.
taggedtextwriter.m	Functions to prepare content for export as XML or SGML tagged text.
tags.m	Functions to open, save and convert markup tag sets.

TextStyleController.m	Manages the Text Style panel called from other panels (e.g., Margins & Columns, Headers & Footers, Headings, etc.)
TextViewCategories.m	Methods that extend NSTextView. Used in IRIndexView and IRRecordView.
textwriter.m	Functions to prepare content for export as plain text.
tools.m	Functions to implement the operations specified by the Check Index, Reconcile Headings, and Split Headings panels (Tools menu).
translate.m	Functions to translate legacy record formats in data imported from early versions of Cindex.
type.m	Functions to manage fonts, and to search for and manage font and style attributes in records.
units.m	Functions to manage units for text layout.
UserIDController.m	Manages the User Identifier panel (presented on launch, according to Preferences setting).
utilities.m	General utility functions, including some utf16 to utf8 conversions.
utime.m	Functions for parsing and formatting dates and times.
v1handler.m	Functions to convert version 1 documents to the format used by version 2.
v3handler.m	Functions to convert older documents (from versions 1 and 2) to the formats used by versions 3 and 4.
VerifyRefsController.m	Unused since version 4.1.
xmlparser.m	Parses records imported from IXML files.
xmlwriter.m	Formats records as XML for exporting to IXML files.
xpresswriter.m	Functions to prepare content for export to QuarkXPress.

Appendix 2. Windows Code Files

abbrev.c	Functions to manage sets of abbreviations.
apiservice.c	Handles DDE commands from API (Publishers' Edition only).
attributedstrings.c	Functions to compose and decompose a compound string of record text into an ATTRIBUTEDSTRING object that separates text from font and style attributes.
cglobal.c	Declares global variables and initializes preferences.
clip.c	Displays clipboard text in a window (Window menu)
collate.c	Functions to extract text strings from records, condition them, then compare them using the Unicode collation rules.
commands.c	Manages error and warning messages + support functions for identifying dates and record ranges.
containerset.c	Functions for managing the container window that holds the main index window, and, depending in user configuration, also the record entry window.
dde.c	Functions to manage DDE callbacks (used to open documents when user double-clicks document icon). Additional functions used by Publishers' Edition to manage Cindex API calls.
draftstuff.c	Functions to create draft formatted entries from records.
edit.c	Functions for managing the following commands (Edit menu): Delete/Restore records; Label records; Demote Headings; Duplicate records. Also the Save Group command (File menu) and the Groups panel (Tools menu).
errors.c	Functions for displaying error, warning, and info messages.
export.c	Functions to export index records.
files.c	Functions to manage document opening and closing.
findset.c	Functions to manage the Find panel (Edit menu).

formattedexport.c	Functions to export a formatted index
formset.c	Functions to manage the following panels in the Document menu: Margins & Columns; Grouping Entries; Headers & Footers; Style & Layout; Headings; Cross-References; Page References; Styled Strings.
formstuff.c	Functions to create fully formatted entries from records.
group.c	Functions to manage record groups.
hspell.c	Functions to interface with the hunspell library.
httpservice.c	Connects to the indexres server to download and display an alert about a new version of Cindex.
import.c	Functions to import index records.
imwriter.c	Functions to prepare content for export to Index-Manager.
indesign.c	Functions to prepare content for export to InDesign.
index.c	Miscellaneous functions to check integrity/repair the index, set its size in memory, read and write headers, flush records.
indexset.c	Functions to manage the following panels in the Document menu: Index Structure; Reference Syntax; Smart Flip words.
json.c	Functions to parse JSON (used by httpservice when retrieving information about upgrades).
keytool.c	Functions for managing the Hot Keys panel, and configuring hot keys.
locales.c	Functions for retrieving information about the user's locale (country).
macros.c	Functions to record and replay macros, and load and save them. Called from Record Events, and Play Events (Tools menu).
main.c	Program entry point. Also contains initialization code.
mfile.c	Functions for managing memory mapping of files.

modify.c	Functions for managing the record-entry window, including event handling, command dispatch, and text editing.
print.c	Functions for printing the index and contents of windows.
rcontainerset.c	Functions for managing the container window that holds the record entry window.
recole.c	Contains callbacks for the object linking and embedding interface, dragging/copying and pasting records into the main index window.
records.c	Functions for retrieving and updating records, and manipulating record text.
refs.c	Functions for managing page references within the locator field of a record.
regex.c	Functions for forming regular expressions, and finding and replacing matched text.
registry.c	Functions for managing registry keys and settings.
reole.c	Contains callbacks for the IRichEditOle interface, used by the rich edit control in the record-entry window.
replaceset.c	Functions to manage the Replace panel (Edit menu).
rtfwriter.c	Functions to prepare content for export as Rich Text Format.
search.c	Functions for searching for content in records, for replacing found content, and for verifying, converting, and generating cross-references.
searchset.c	Functions to manage the Go To panel (View menu) and the Hide by Attribute panel (View menu).
sort.c	Functions for managing the AVL sort tree. Also contains the function for compressing records.
sortset.c	Functions to manage the Sort panel (Tools menu).
spell.c	Functions for checking spelling and replacing misspelled words.
spellset.c	Functions to manage the Spelling panel (Edit menu).

strings.c	Miscellaneous functions for manipulating strings and compound string (xstrings).
tagstuff.c	Functions to manages the Markup Tags panel (Tools menu) and to open, save and convert markup tag sets.
tagwriter.c	Functions to prepare content for export as XML or SGML tagged text.
text.c	Functions to manage the auxiliary text display window associated with the index.
textwriter.c	Functions to prepare content for export as plain text.
tools.c	Functions to implement the operations specified by the Check Index, Reconcile Headings, and Split Headings panels (Tools menu).
toolset.c	Functions to manage the following panels (Tools menu): Check Index; Reconcile Headings; Manage Cross-References; Alter References; Split Headings; Count Records; Index Statistics; Fonts.
translate.c	Functions to translate legacy record formats in data imported from early versions of Cindex.
typestuff.c	Functions to manage fonts, and to search for and manage font and style attributes in records.
util.c	General utility functions, including some for scaling windows and for utf16 to utf8 conversions.
utime.c	Functions for parsing and formatting dates and times.
v3handler.c	Functions to convert older documents to the formats used by versions 3 and 4.
viewset.c	Functions for managing the main index view, including event handling, command dispatch, and text display.
xmlparser.c	Parses records imported from IXML files.
xmlwriter.c	Formats records as XML for exporting to IXML files.
xpresswriter.c	Functions to prepare content for export to QuarkXPress.