xindy

A Flexible Indexing System

Roger Kehr, Institut für Theoretische Informatik, TH-Darmstadt
xindy by Topic, Edition March 1997 for **xindy** version 1.1

The **xindy** *by topic manual is intended for all practitioners that do not want to read too much about* **xindy***s details but rather start to solve concrete problems immediately. Thus, we have written it in a how-to manner starting from a concrete problem and describing all the necessary steps that are needed to solve the problem.*

Contents

1 How to ...

1.1 I don't want to write style files. How can I proceed?

You can use the wrapper program `makeindex4`. It tries to do it's very best to make **xindy** behave as `makeindex` does. But if you ever need to use some of the features of **xindy** you should learn how to write an index style. Its easy!

1.2 Eventually, I decided to write my first index style. How can I start?

Congratulations! You have made a good decision, indeed.

For a first impression, how a style file can be written from scratch reference the *tutorial* that comes with this distribution. It is written as a guided step-by-step practicing exercise and you can learn the basic concepts quite easily.

Afterwards, the best starting point is to make a copy of the template file that contains all the necessary commands that are needed to make a `makeindex`-like index. You can find it in the subdirectory `markup/tex` of the module library. Starting from this template you can remove or add commands as necessary.

Additionally, consult the library of predefined *index style modules* that comes with this distribution. Solutions for most of the typical problems can be found there, such as a module for doing case-insensitive sorting rules, or a typical TEX-like markup. Most of the time is is enough to include some of these modules and add a few additional commands.

Maybe some of the examples coming with the *test-suite* are good examples of how unusual index style files can be written.

Thus, there are many ways to learn writing an index style file. But it is very easy and after some experience you can process indexes your friends will be jealous of.

1.3 How do I use `makeindex4`?

Process your document as usual. Then run `makeindex4` on the index file. It produces an index that should equal the one you would get with an ordinary `makeindex` run. As far as you are satisfied with the default behaviour of `makeindex`, `makeindex4` will produce comparable results.

Some of the command-line options of `makeindex` are accepted by `makeindex4`, others aren't. This may change in future releases, but we recommend using plain **xindy** after a phase of investigation, since one cannot use all of its features with `makeindex4`. You will be informed about unsupported command-line arguments when running `makeindex4`.

If you have written special style files for `makeindex` they will no longer work with `makeindex4`. Go ahead and write a new style file for **xindy**.

1.4 Why a completely new indexing tool? `makeindex` works fine!

With the *International MakeIndex* project, Joachim Schrod and Gabor Herr [3, 6] have shown that adding extensions to `makeindex` is a difficult job. Thus we have decided to develop a new indexing tool from scratch. The new tool is based on a new requirements analysis and offers very interesting features for processing very complex indexing schemes. The resulting index model is described in [5].

1.5 I'm an old `makeindex` wizard. What does xindy offer that `makeindex` doesn't?

Here are the most important differences between **xindy** and `makeindex`:

Internationalization

xindy can be configured to process indexes for many languages with different letter sets and different sorting rules. For example, many roman languages such as Italian, French, Portuguese or Spanish contain accentuated letters such as À, Á, ñ. Other languages from northern Europe have letters like Ä, Ø, æ or ß which often can't be processed by many index processors not talking about sorting them correctly into an index. The xindy-system can be configured to process these alphabets by defining *sort* and *merge-rules* that allow expressing of language specific rules. One example of such a rule would be

```
(sort-rule {\tt '}ä{\tt '} {\tt '}ae{\tt '})
```

defining that a word containing the umlaut-a will be sorted as if it contained the letters `ae` instead. This is one form of how the umlaut-a is sorted into german indexes. With an appropriate set of rules one can express the complete rules of a specific language.

Location classes.

`makeindex` is able to recognize and process arabic numbers, roman numerals and letter-based alphabets as specifiers for the indexed location. Simple composite structures of these are also possible to process. This implicit recognition scheme has completely been dropped in favour of a well-defined and very powerful declaration scheme called *location-classes*. Thus, **xindy** initally does not know any location-class by default and must be instructed to accept certain location-classes. A typical declaration might look like:

```
(define-location-class "page-numbers" ("arabic-numbers"))
```

This declares that page numbers consist of the enumeration of the arabic numbers. The arabic numbers are referred to as *alphabets*. Users may use the pre-defined alphabets arabic numbers, roman numerals, etc. or define new alphabets as needed. See the tutorial that comes with this distribution for some examples.

The concept of attributes.

With `makeindex` one can assign a markup to each index entry using the encapsulators (usually following the vertical bar sign in an index entry command). For example in the specification

```
\index{xindy|bold}
```

the encapsulator is `bold` which encapsulates the page-numbers in the markup-phase. An additional TEX-macro must be supplied to assign some markup with the page number. This concept has completely been dropped in **xindy** in favour of a more powerful scheme called *attributes*. Attributes can be used to (i) define several grouping and ordering rules with locations and we can define (ii) markup-tags for the document preparation system.

The result of this design decision is that the user is required to define the attributes in the style file and not in the document preparation system. The reasons lie in the more powerful markup scheme of **xindy** which can't be specified in the document processor anymore. In fact, the `makeindex`-like markup is only a small subset of **xindy**s features.

Cross references.

Cross-references were implemented in `makeindex` with the encapsulation mechanism, which only served for markup purposes. This has been completely separated in **xindy**. Here we distinguish cleanly between attributes and cross references. This makes it possible to implement *checked* cross references, i.e. cross-refernces that can be checked if they point to an existing index entry instead of somewhere "behind-the-moon".

Less command-line options.

xindy has dropped the usage of command-line options in favour of a well-defined indexstyle description language. Thus, options that could be activated at the command-line level of `makeindex` must now be specified in the indexstyle file. This sounds more restrictive than it is, because the indexstyle files can be composed from several modules which makes it possible to write style files in just a few lines of code.

Raw index parser.

The parser built into `makeindex` has completely been separated from the core **xindy** system. **xindy** understands a well-defined specification language for the raw index that is completely different from `makeindex`, but in our opinion more maintainable than the `makeindex` format. This requires a separate filter that transforms arbitrary indexes to the **xindy** format. An example filter is the program `tex2xindy` that comes with this distribution.

Summed up, some of the implicit assumptions made by `makeindex` have been replaced and now burdend to the user. The reason is that many of `makeindex`'s assumptions were no longer valid in multi-language environments or documents with arbitrary location structures. This also characterizes **xindy** more as a framework instead of a end-user-tailored product. One should notice that writing an appropriate index style is an essential part of the document preparation process and should be tailored to each document anew.

1.6 What is `tex2xindy`?

`tex2xindy` is a filter that parses '`.idx`' or similar files and converts the `\indexentry` macros into a form readable by **xindy**.

The parser of `makeindex` can be configured to recognize different quoting characters, etc. (see the man-page for `makeindex`, section *input style specifiers* for further details). We have tried to extract the parser from `makeindex` but due to several probems we have finally rewritten the parser using `lex`. Scanners written with `lex` are usually fixed to a specific character set used in the regular expressions. Our parser, `tex2xindy` is therefore not configurable. If one uses a different configuration of the `makeindex` input style specifiers, one can change the source (`tex2xindy.l`) to generate a completely new parser. From our personal experience we have rarely used more than two different parsers in practice so we have written `tex2xindy` in a form that is easily maintainable. The input specifiers are stored symbolically in the source. The definiton section looks like this:

```
KEYWORD  \\indexentry
ENCAP    \|
ACTUAL   @
ESCAPE   \\
LEVEL    !
QUOTE    \"
ROPEN    \(
RCLOSE   \)
ARGOPEN  \{
ARGCLOSE \}
```

These definitions are essentially the input style specifiers as can be found in the man-page of `makeindex`. Changing this section according to your needs and recompiling `tex2xindy` should be an easy task. Maybe we will include more pre-defined parsers in future releases if necessary.

1.7 How to write my first index style?

Copy the file `tex/makeidx.xdy` from the library to your local directory. It is documented in in a way that should make it easy to fill in new commands or remove or modify others.

1.8 How works `makeindex4`?

This job is now done automatically by `makeindex4`. It calls `tex2xindy` to transform the raw-index into the format suitable for **xindy**. `tex2xindy` emits some information about the attributes (aka. encapsulators in `makeindex`) and the usage of cross-references into a file, which has the extension '`.sta`'. The `makeindex4` program, written in `perl`, parses this statistics-file and generates the above presented indexstyle commands for you automatically including the required declaration of all attributes in the whole index and their markup.

Another problem is the automatic detection of cross-references. As noted above, `makeindex` handles cross-references with its encapsulation mechanism, a scheme which has been dropped in **xindy** and replaced by a more powerful mechanism. To implement a simple plug-in mechanism we have extended the syntax of the `tex2xindy` filter to identify encapsulators of the form

```
\indexentry{...|encap{...}}{...}
```

as a cross-reference, whereas encapsulators of the form

```
\indexentry{...|encap}{...}
```

are treated as ordinary attributes. This is standard practice defining cross-references in `makeindex`. Thus, `tex2xindy` distinguishes these two forms of encapsulators as opposed to `makeindex` and our plug-in `makeindex4` generates the appropriate definitions of the cross-reference classes as well.

1.9 How works the actual key feature of `makeindex` with **xindy**?

The treatment of the actual key (usually denoted with `@`, the at-sign) has changed with **xindy**. Specifying index entries with a specific markup can be done in `makeindex` with the actual key. The `makeindex-3` system and **xindy** offer the *merge-* and *sort-rules* to transform a key into different representations, limiting the need to specify an actual key. For example they support a style of writing

```
\index{\bf{VIP}}
```

which can be transformed with a rule like

```
(merge-rule "\bf{\(.*\)}" "\1" :again :bregexp)
```

which removes the macro definition for merging and sorting keywords, but keeping the original definition for markup purposes. Therefore we don't need any actual keys for all keywords written in boldface.
The `makeindex` behaviour, that the two keywords

```
\index{VIP}
\index{VIP@\bf{VIP}}
```

are seen as two distinct index entries, can be simulated using the following definition:

```
(merge-rule "\bf{\(.*\)}" "\1~e" :again :bregexp)
```

This rule tells **xindy** to remove the boldface macro for merging and sorting purposes but defines the replacement to include the special character `~e`, which is the last character in the alphabet (ISO-Latin in our case). This makes **xindy** treat them as different keywords and positions the boldface keyword right behind the one without any markup. Thus we receive the following mapping:

```
Keyword:    Merged and sorted as:   Appears in the index as:
VIP             VIP                        VIP
\bf{VIP}        VIP~e                      \bf{VIP}
```

With this new style of writing keywords and defining their markup, the need to explicitly specifying the print key (aka. actual key) has convinced us to remove the `makeindex` way of defining keywords.

1.10 I want to process an index for my native language. What must I do?

What makes `makeindex` hardly usable in non-English speaking countries is its lack of support of language specific alphabets and sort orderings. For example, many roman languages such as Italian, French, Portuguese or Spanish contain accented letters such as À, Á, ñ. Other languages from northern Europe have letters like Ä, Ø, æ or ß which often can't even be processed by many index processors let alone sorting them correctly into an index.
Two problems must be solved when processing indexes with a new languages:

1. The *sort ordering* of the indexed terms must be specified in an appropriate manner. This problem can be solved using the so-called *keyword mappings*.

2. The *letter groups* that partition the indexed terms into separate sections must be specified.

The **xindy** system can be configured to process these alphabets by defining *sort* and *merge rules* that allow expressing language specific rules.

The keyword mappings are as follows: The *merge key* is generated from the *main key* with the so called *merge mapping*. The merge mapping can be specified with the command `merge-rule`. The *sort key* is derived from the merge key using the *sort mapping* specified with the `sort-rule` command. The following scheme shows this mapping process:

$$\mathsf{key} \xrightarrow{\textit{merge mapping}} \mathsf{merge\ key} \xrightarrow{\textit{sort mapping}} \mathsf{sort\ key}$$

The index style commands accomplishing this task are `sort-rule` and `merge-rule`. One example of such a rule would be

```
(sort-rule "ä" "ae")
```

defining that a word containing the umlaut-a will be sorted as if it contained the letters `ae` instead. This is one form of how the umlaut-a (ä) is sorted into german indexes. With an appropriate set of rules on can express the complete rules of a specific language.

An example of how an appropriate mapping for some of the Roman languages could look like is:

```
(sort-rule "à" "a")
(sort-rule "á" "a")
(sort-rule "ã" "a")
(sort-rule "è" "e")
(sort-rule "é" "e")
(sort-rule "ç" "c")
```

This makes the accented letters be sorted as their unaccented counterparts, yielding the desired sort ordering.

Sometimes it is necessary to specify keyword mappings that tell the system to put something *behind* something else. For instance, we'd like to map the character *ö* behind the letter *o*. No problem if you use the special characters `~b` and `~e` which are called the *beginning* and *ending* characters. The first letter lexicographically precedes all other letters whereas the latter one comes after all others. Our mapping problem can now be specified as follows.

```
(sort-rule "ö" "o~e")
```

Now the *ö* is directly positioned after the *o* but before *p*.

See the manual for a detailed description of this feature. Also be informed that the keyword mappings can be specified with regular expressions. Rules of the form

```
(merge-rule "[-$()]" "")
```

are possible. This on removes all letters of the defined letter class. Regular expression substitutions are possible as well. Refer to the manual for an exact description.

1.11 In my index the capitalized words appear after the lowercase words. Why?

The default sort ordering sorts letters according to their ordinal number in the ISO Latin alphabet. As a consequence the lowercase letters appear before the uppercase letters. To sort them case-insensitively use the command

```
(require "lang/latin/caseisrt.xdy")
```

This module defines the appropriate sort rules for the letters 'A-Z' for latin-based alphabets. If your language has more letters simply add the missing ones into your style file. Have a look at the module to see how to the sort rules are defined.

1.12 In my index there are no letter groups, yet!

Letter groups for latin based alphabets can be defined with the command

```
(require "lang/latin/letgroup.xdy")
```

If your language needs additional letter groups you can insert them into the previously defined letter group with inserting definitions of the following form:

```
(define-letter-group "ly" :after "l" :before "m")
(define-letter-group "ny" :after "n" :before "o")
```

This adds two more letter groups to the latin alphabet. Group *ly* is inserted between *l* and *m*, and *ny* is inserted between *n* and *o*. This is how two additional letters of the Hungarian alphabet can be inserted.

1.13 How can I get rid of any formatting information in the keyword?

Assume you have index entries containing arbitrary formatting information. For example you write your index entries in TEX in the following form:

```
\index{\bf{In boldface please}}
```

To avoid specifying for each index entry the print key separately as can be done with the following command

```
\index{In boldface please@\bf{In boldface please}}
```

you can instead define a rule doing this task for you:

```
(merge-rule "\\bf *{(.*)}" "\1" :eregexp :again)
```

This extended regular expression matches all strings that are surrounded by this formatting command and in the *merge phase* the formatting command is simply stripped off. Thus, you don't need to write an explicit print key anymore.

If for some reason the same word appears more than once in the index, each time having another markup tag as in the following example

```
index
{\tt index}
```

you must be warned that a rule like

```
(merge-rule "{\\tt *(.*)}" "\1" :eregexp :again)
```

is probably not correct. In this case the above strings are both mapped into the string `index` thus joining their location references into one index entry. This happens because the result of the merge mapping is used as the equality citerium which views both keywords as equal. To avoid this you should specify instead

```
(merge-rule "{\\tt *(.*)}" "\1~e" :eregexp :again)
```

With the additional meta character `~e` the substitution of the second key word is placed *after* the first one making them different index entries. If the second keyword should appear first, use `~b` instead.

1.14 In my index the word *-foo* appears before *bar*. What must I do?

Especially for hierarchical indexes sometimes the result is not as expected due to special characters appearing in the keyword. In the following example the word 'card' should appear before '-eyed' since the hyphen should not count as an ordinary character by means of sorting.

```
green
   -eyed  12
   card   15
```

This is especially problematic if the list of words on the second level is very long. To make the hyphen be simply ignored during the sorting process you should specify the following command in the index style:

```
(sort-rule "-" "")
```

This makes '-eyed' be sorted as 'eyed' thus making it appear *after* 'card' as desired.

1.15 I want to use letter ordering instead of word ordering in my index.

According to the *Chicago Manual of Style* there exist two different schemes of sorting word lists. In *word ordering* a blank precedes any letter in the alphabet, whereas in *letter ordering* it does not count at all. The following example borrowed from the `makeindex` man-page illustrates the difference:

```
Word Order:         Letter Order:
 sea lion            seal
 seal                sea lion
```

By default, **xindy** uses word ordering. To use letter ordering include the appropriate module with the following command:

```
(require "ord/letorder.xdy")
```

It actually defines the following command:

```
(sort-rule " " "")
```

This simply removes all blanks from the keyword resulting in the desired behaviour.

1.16 My document does not have page numbers, but a different scheme. What must I do?

The ability to deal with user-definable location structures is one of the most important new features of **xindy**. Many documents have a document structure that is not based on page numbers. Typical examples are manuals or appendices that come with a *chapter/page-per-chapter* numbering scheme, URLs, Bible verses, etc. One can even imagine the Greek alphabet as possibly appearing in a location reference. In our analysis we have found many interesting examples of location references that made us to develop the concept of *location classes*.

A location class is defined by a sequence of alphabets. An alphabet can be the set of arabic numbers (0, 1, 2, ...) or the roman numerals (i, ii, iii, ...). These are built-in alphabets in **xindy**. Addtionally, one can define more alphabets in the index style with a command like

```
(define-alphabet "weekdays"
       ("mon" "tue" "wed" "thu" "fri" "sat" "sun"))
```

Based on alphabets one can now compose a location class as follows:

```
(define-location-class "weekday-hours"
       ("weekday" :sep ":" "arabic-numbers"))
```

This class description indicates that all location refernces matching this template are viewed as correct instances of this class. Here `:sep` makes the dot serving as a *separation string* separation the alphabets from each other. Example instances of this class are:

```
mon:23, thu:45, sun:17
```

For more detailed information consult the description of the command `define-location-class` in the reference manual.

1.17 I don't want to have ranges in my index. What can I do?

By default, **xindy** joins three successive location references into a *range*. Ranges are used as an abbrevation for a long sequence of location references. For exmaple the sequence

```
12, 13, 14, 15, 16
```

would be shorter represented as

```
12-16
```

If you don't want to have ranges, simply define your location class in the form

```
(define-location-class ... :min-range-length none)
```

The argument `:min-range-length none` avoids forming of ranges. Arbitrary numbers instead of `none` define the minimum length of a sequence of location references that are needed to form a range. **xindy**s default value is 2.

1.18 I want to markup ranges of different length differently. How do I accomplish this?

A common way of tagging ranges is as follows: a range of length 1 is printed with the starting page number and the suffix 'f.', those of length 2 with suffix 'ff.', and all others in the form 'X–Y'.

Assume we want to do this for the location class *pagenums* we can specify the markup as follows:

```
(markup-range :class "pagenums" :close "f."  :length 1 :ignore-end)
(markup-range :class "pagenums" :close "ff." :length 2 :ignore-end)
(markup-range :class "pagenums" :sep "--")
```

The first command indicates that a range *(X,Y)* of length 1 should be printed in the form *Xf.*, a range of length 2 as *Xff.* and all others in the form X–Y. The switch `:ignore-end` causes the end of range location reference Y to be suppressed in the resulting output.

1.19 I need to suppress some of the markup tags. How can I do this?

Sometimes it is necessary to hide some of the parts of the index. If you have a text formatter that allows comments or macros that possibly expand to nothing, just define appropriate markup that makes things invisible to the formatter. For example, with TEX you can define a macro like this

```
\def\ignore#1{}
```

If you additionally define markup like this

```
(markup-index :open "\ignore{" :close "}")
```

you can throw away the complete index if you like, which would be a real pity!

1.20 Whats it all about those cross references?

Cross references are references pointing to an item in the index itself. Typical examples are:

`foo-bar` *see* `baz`

With `makeindex` cross references could be specified with the encapsulation mechanism. This has completely been removed in **xindy** and we have made cross references real first-class objects.

In **xindy** one can declare different cross reference classes, whose purpose is (a) to make all instances of a certain class appear next to each other, and (b) to specify appropriate markup with them.

`tex2xindy` recognises all index entries of the form

```
\index{...|\macro{where}}
```

as cross references. Here `macro` stands for an arbitrary macro name and `where` is interpreted as the target keyword of the cross references.
If you want to use these cross references with **xindy**; add the following line to your style file.

```
(define-crossref-class "macro")
```

Additionally, you can assign specific markup to cross references using the `markup-crossref-` commands.

2 Practical Tips

2.1 What LaTeX-package should I use in conjunction with xindy?

We strongly recommend using the LaTeX2e-package `index` written by David M. Jones, which is available at CTAN. It supports multiple indexes as well as several shortcuts to easily index terms in a document. Multiple indexes support the generation of several indexes for one document. For instance, one can make an author or command index in addition to a global index.

2.2 What editor should I use when writing xindy style files?

Use `Emacs` or `XEmacs`. Turn on the Lisp-mode with

```
M-x lisp-mode
```

and you can properly indent commands using `M-q`. To enter this mode automatically add the following lines to the end of the style file:

```
^L
;; Local Variables:
;; mode: lisp
;; End:
```

The `^L` (Control-L) can be entered with `C-q C-l`.

2.3 I have written a module for processing language *foolandic*. What must I do?

Great! Send it to us! It will become a part of the system in the next release.

3 Errors and Problems

3.1 The keyword mappings don't work as expected!

Sometimes the keyword mappings don't work as expected. Especially in cases with several regular expressions one might get confused about what rule matches exactly when. We have incorporated a detailed logging mechanism that lets one step by step follow the rules that accomplish the keyword mapping.

When running **xindy** you can turn on this feature with the command line option ‘`-L`’. This option followed by one of the numbers 1, 2, or 3 turns on the appropriate debugging level. Turning on level 2 or 3 and specifying a log-file with the command line option ‘`-l`’ a trace of the mappings is recorded in the log-file. A sample output looks like the following:

```
Mappings: (add (merge-rule :eregexp ‘^\\bf{(.*)}’ ‘\1’ :again)).
Mappings: (add (merge-rule :eregexp ‘^\\"([AEOUaeou])’ ‘\1’)).
 ...
Mappings: (compare ‘\"A\"a’ :eregexp ‘^\\bf{(.*)}’)
Mappings: (compare ‘\"A\"a’ :eregexp ‘^\\"([AEOUaeou])’) match!
Mappings: (compare ‘\"a’ :eregexp ‘^\\bf{(.*)}’)
Mappings: (compare ‘\"a’ :eregexp ‘^\\"([AEOUaeou])’) match!
Mappings: (merge-mapping ‘\"A\"a’) -> ‘Aa’.
```

This trace shows that initially two regular expression mappings have been added to the rule set. The second section shows how the keyword ‘`\"A\"a`’ is compared to these rules and substitutions are applied as matches are found. In the last line the result of the keyword mapping is reported.

3.2 I’m totally confused by the markup scheme!

A very important feature is the ability to *trace* all markup tags **xindy** emits in the markup phase. Simply use the command line switch `-t` or insert the command

```
(markup-trace :on)
```

into the index style. This informs **xindy** to emit additional pseudo markup that can be used to understand and debug the markup phase. An example output might look like the following:

```
<INDEX:OPEN>
  <LETTER-GROUP-LIST:OPEN>
    <LETTER-GROUP:OPEN ["a"]>
      <INDEXENTRY-LIST:OPEN [0]>
        <INDEXENTRY:OPEN [0]>
          <KEYWORD-LIST:OPEN [0]>
            <KEYWORD:OPEN [0]>
 ...
```

The symbolic tags directly lead one to the command that is responsible for the definition of that markup tag. For example, the tag `LETTER-GROUP-LIST:OPEN` indicates that the command `markup-letter-group-list` is responsible for replacing this symbolic tag by a real one. Give it a try if you find yourself confused by your own markup specification.

4 References

The following books and papers were referenced in this document:

1. D. E. Knuth, *The TeXbook*, Addison-Wesley, 1984

2. Leslie Lamport, LaTeX *A Document Preparation System*, Addison-Wesley, 1986
3. Joachim Schrod, *An International Version of MakeIndex*, Cahiers GUTenberg, 10, p81-90, 1991
4. Pehong Chen and Michael A. Harrison, *Automating Index Preparation*, TR-87/347, University of California Berkeley, 1987
5. Roger Kehr, **xindy**, *Ein Flexibles Indexierungssystem*, Studienarbeit FB Informatik, TH-Darmstadt, November 1995.
6. Joachim Schrod and Gabor Herr, *MakeIndex Version 3.0*, Technical University of Darmstadt, 1991