Zebra - User’s Guide and Reference

Sebastian Hammer

Adam Dickmeiss

Heikki Levanto

Mike Taylor

Zebra - User's Guide and Reference

by Sebastian Hammer

by Adam Dickmeiss

by Heikki Levanto

by Mike Taylor

Zebra is a free, fast, friendly information management system. It can index records in XML/SGML, MARC, e-mail archives and many other formats, and quickly find them using a combination of boolean searching and relevance ranking. Search-and-retrieve applications can be written using APIs in a wide variety of languages, communicating with the Zebra server using industry-standard information-retrieval protocols.

This manual explains how to build and install Zebra, configure it appropriately for your application, add data and set up a running information service. It describes version 1.3.7 of Zebra.

Table of Contents

Chapter 1. Introduction

Overview

Zebra (http://indexdata.dk/zebra/) is a high-performance, general-purpose structured text indexing and retrieval engine. It reads records in a variety of input formats (eg. email, XML, MARC) and provides access to them through a powerful combination of boolean search expressions and relevance-ranked free-text queries.

Zebra supports large databases (tens of millions of records, tens of gigabytes of data). It allows safe, incremental database updates on live systems. Because Zebra supports the industry-standard information retrieval protocol, Z39.50, you can search Zebra databases using an enormous variety of programs and toolkits, both commercial and free, which understand this protocol. Application libraries are available to allow bespoke clients to be written in Perl, C, C++, Java, Tcl, Visual Basic, Python, PHP and more - see the ZOOM web site (http://zoom.z3950.org/) for more information on some of these client toolkits.

This document is an introduction to the Zebra system. It explains how to compile the software, how to prepare your first database, and how to configure the server to give you the functionality that you need.

Features

This is an overview of some of Zebra's most important features:

- Very large databases: logical files can be automatically partitioned over multiple disks.
- Arbitrarily complex records. The internal data format is a structured format conceptually similar to XML or GRS-1, which allows lists, nested structured data elements and variant forms of data.
- Robust updating - records can be added and deleted "on the fly" without rebuilding the index from scratch. Records can be safely updated even while users are accessing the server. The update procedure is tolerant to crashes or hard interrupts during database updating - data can be reconstructed following a crash.
- Configurable to understand many input formats. A system of input filters driven by regular expressions allows most ASCII-based data formats to be easily processed. SGML, XML, ISO2709 (MARC), and raw text are also supported.
- Searching supports a powerful combination of boolean queries as well as relevance-ranking (free-text) queries. Truncation, masking, full regular expression matching and "approximate matching" (eg. spelling mistakes) are all handled.
- Index-only databases: data can be, and usually is, imported into Zebra's own storage, but Zebra can also refer to external files, building and maintaining indexes of "live" collections.
- Zebra is written in portable C, so it runs on most Unix-like systems as well as Windows NT. A binary distribution for Windows NT is available at http://ftp.indexdata.dk/pub/zebra/win32/, and pre-built packages are available for some Linux distributions: Red Hat 7.x RPMs at http://ftp.indexdata.dk/pub/zebra/RedHat7.X/ and Debian packages at http://ftp.indexdata.dk/pub/zebra/debian/

Z39.50 protocol support:

- Protocol facilities: Init, Search, Present (retrieval), Segmentation (support for very large records), Delete, Scan (index browsing), Sort, Close and support for the “update” Extended Service to add or replace an existing XML record.
- Piggy-backed presents are honored in the search request - that is, a subset of the found records can be returned directly with a search response, enabling search and retrieval to happen in a single round-trip.
- Named result sets are supported.
- Easily configured to support different application profiles, with tables for attribute sets, tag sets, and abstract syntaxes. Additional tables control facilities such as element mappings to different schema (eg., GILS-to-USMARC).
- Complex composition specifications using Espec-1 (partial support). Element sets are defined using the Espec-1 capability, and are specified in configuration files as simple element requests (and, optionally, variant requests).
- Multiple record syntaxes for data retrieval: GRS-1, SUTRS, XML, ISO2709 (MARC), etc. Records can be mapped between record syntaxes and schemas on the fly.

Applications

Zebra has been deployed in numerous applications, in both the academic and commercial worlds, in application domains as diverse as bibliographic catalogues, geospatial information, structured vocabulary browsing, government information locators, civic information systems, environmental observations, museum information and web indexes.

Notable applications include the following:

DADS - the DTV Article Database Service

DADS is a huge database of more than ten million records, totalling over ten gigabytes of data. The records are metadata about academic journal articles, primarily scientific; about 10% of these metadata records link to the full text of the articles they describe, a body of about a terabyte of information (although the full text is not indexed.)

It allows students and researchers at DTU (Danmarks Tekniske Universitet, the Technical College of Denmark) to find and order articles from multiple databases in a single query. The database contains literature on all engineering subjects. It’s available on-line through a web gateway, though currently only to registered users.

More information can be found at http://www.dtv.dk/help/dads/index_e.htm

NLI-Z39.50 - a Natural Language Interface for Libraries

Fernuniversität Hagen in Germany have developed a natural language interface for access to library databases. http://ki212.fernuni-hagen.de/nli/NLIintro.html In order to evaluate this interface for recall and precision, they chose Zebra as the basis for retrieval effectiveness. The Zebra server contains a copy of the GIRT database, consisting of more than 76000 records in SGML format (bibliographic records from social science), which are mapped to MARC for presentation.

(GIRT is the German Indexing and Retrieval Testdatabase. It is a standard German-language test database for intelligent indexing and retrieval systems. See http://www.gesis.org/forschung/informationstechnologie/clef-delos.htm)

Evaluation will take place as part of the TREC/CLEF campaign 2003 http://clef.iei.pi.cnr.it or http://www4.eurospider.ch/CLEF/

For more information, contact Johannes Leveling <Johannes.Leveling@FernUni-Hagen.De>

ULS (Union List of Serials)

The M25-Link systems team (http://www.m25lib.ac.uk/M25link/) are involved in a project called ULS to provide a union catalogue for periodicals in 21 member libraries. They do this with an unusual architecture which they call a “non-distributed virtual union catalogue”.

The member libraries send in data files representing their periodicals, including both brief bibliographic data and summary holdings. Then 21 individual Z39.50 targets are created, each using Zebra, and all mounted on the single hardware server. The live service provides a web gateway allowing Z39.50 searching of all of the targets or a selection of them. Zebra’s small footprint allows a relatively modest system to comfortably host the 21 servers.

More information can be found at http://www.m25lib.ac.uk/ULS/

Various web indexes

Zebra has been used by a variety of institutions to construct indexes of large web sites, typically in the region of tens of millions of pages. In this role, it functions somewhat similarly to the engine of google or altavista, but for a selected intranet or a subset of the whole Web.

For example, Liverpool University’s web-search facility (see on the home page at http://www.liv.ac.uk/ and many sub-pages) works by relevance-searching a Zebra database which is populated by the Harvest-NG web-crawling software.

For more information on Liverpool university’s intranet search architecture, contact John Gilbertson <jgilbert@liverpool.ac.uk>

Kang-Jin Lee <lee@arco.de>, has recently modified the Harvest web indexer to use Zebra as its native repository engine. His comments on the switch over from the old engine are revealing:

> The first results after some testing with Zebra are very promising. The tests were done with around 220,000 SOIF files, which occupies 1.6GB of disk space.
>
> Building the index from scratch takes around one hour with Zebra where [old-engine] needs around five hours. While [old-engine] blocks search requests when updating its index, Zebra can still answer search requests. [...] Zebra supports incremental indexing which will speed up indexing even further.

> While the search time of [old-engine] varies from some seconds to some minutes depending how expensive the query is, Zebra usually takes around one to three seconds, even for expensive queries. [...] Zebra can search more than 100 times faster than [old-engine] and can process multiple search requests simultaneously
>
> I am very happy to see such nice software available under GPL.

Support

You can get support for Zebra from at least three sources.

First, there's the Zebra web site at http://indexdata.dk/zebra/, which always has the most recent version available for download. If you have a problem with Zebra, the first thing to do is see whether it's fixed in the current release.

Second, there's the Zebra mailing list. Its home page at http://indexdata.dk/mailman/listinfo/zebralist includes a complete archive of all messages that have ever been posted on the list. The Zebra mailing list is used both for announcements from the authors (new releases, bug fixes, etc.) and general discussion. You are welcome to seek support there. Join by sending email to <`zebra-request@indexdata.dk`> with the word `subscribe` in the body of the message.

Third, it's possible to buy a commercial support contract, with well defined service levels and response times, from Index Data. See http://indexdata.dk/support2/ for details.

Future Directions

These are some of the plans that we have for the software in the near and far future, ordered approximately as we expect to work on them.

- Improved support for XML in search and retrieval. Eventually, the goal is for Zebra to pull double duty as a flexible information retrieval engine and high-performance XML repository. The recent addition of XPath searching is one example of the kind of enhancement we're working on.
- Access to the search engine through SOAP/RPC API to allow the construction of applications without requiring Z39.50 tools. This will shortly be available by means of Index Data's SRW-to-Z39.50 gateway, currently in beta test.
- Finalisation and documentation of Zebra's C programming API, allowing updates, database management and other functions not readily expressed in Z39.50. We will also consider exposing the API through SOAP.
- Support for the use of Perl both for access to the Zebra API and for building extension "plug-ins" such as input filters. The code for this has been contributed to the source tree by Peter Popovics <`pop@indexdata.dk`>, and is in the process of being integrated and tested.
- Improved free-text searching. We're first and foremost octet jockeys and we're actively looking for organisations or people who'd like to contribute experience in relevance ranking and text searching.

Programmers thrive on user feedback. If you are interested in a facility that you don't see mentioned here, or if there's something you think we could do better, please drop us a mail. Better still, implement it and send us the patches.

If you think it's all really neat, you're welcome to drop us a line saying that, too. You can email us on <info@indexdata.dk> or check the contact info at the end of this manual.

Chapter 2. Installation

An ANSI C compiler is required to compile the Zebra server system — `gcc` works fine if your own system doesn't provide an adequate compiler.

Unpack the distribution archive. The `configure` shell script attempts to guess correct values for various system-dependent variables used during compilation. It uses those values to create a `Makefile` in each directory of Zebra.

To run the configure script type:

```
./configure
```

The configure script attempts to use C compiler specified by the `CC` environment variable. If this is not set, `cc` or GNU C will be used. The `CFLAGS` environment variable holds options to be passed to the C compiler. If you're using a Bourne-shell compatible shell you may pass something like this:

```
CC=/opt/ccs/bin/cc CFLAGS=-O ./configure
```

The configure script support various options: you can see what they are with

```
./configure --help
```

Once the build environment is configured, build the software by typing:

```
make
```

If the build is successful, two executables are created in the sub-directory `index`:

`zebrasrv`
: The Z39.50 server and search engine.

`zebraidx`
: The administrative indexing tool.

You can now use Zebra. If you wish to install it system-wide, then as root type

```
make install
```

By default this will install the Zebra executables in `/usr/local/bin`, and the standard configuration files in `/usr/local/share/idzebra` You can override this with the `--prefix` option to configure.

Chapter 3. Quick Start

In this section, we will test the system by indexing a small set of sample GILS records that are included with the Zebra distribution, running Zebra a server against the newly created database, and searching the indexes with a client that connects to that server.

Go to the `examples/gils` subdirectory of the distribution archive. The 48 test records are located in the sub directory `records`. To index these, type:

```
zebraidx update records
```

In this command, the word `update` is followed by the name of a directory: `zebraidx` updates all files in the hierarchy rooted at that directory.

If your indexing command was successful, you are now ready to fire up a server. To start a server on port 2100, type:

```
zebrasrv @:2100
```

The Zebra index that you have just created has a single database named `Default`. The database contains records structured according to the GILS profile, and the server will return records in USMARC, GRS-1, or SUTRS format depending on what the client asks for.

To test the server, you can use any Z39.50 client. For instance, you can use the demo command-line client that comes with YAZ:

```
yaz-client localhost:2100
```

When the client has connected, you can type:

```
Z> find surficial
Z> show 1
```

The default retrieval syntax for the client is USMARC, and the default element set is `F` (“full record”). To try other formats and element sets for the same record, try:

```
Z>format sutrs
Z>show 1
Z>format grs-1
Z>show 1
Z>format xml
Z>show 1
```

```
Z>elements B
Z>show 1
```

> **Note:** You may notice that more fields are returned when your client requests SUTRS, GRS-1 or XML records. This is normal - not all of the GILS data elements have mappings in the USMARC record format.

If you've made it this far, you know that your installation is working, but there's a certain amount of voodoo going on - for example, the mysterious incantations in the `zebra.cfg` file. In order to help us understand these fully, the next chapter will work through a series of increasingly complex example configurations.

Chapter 4. Example Configurations

Overview

`zebraidx` and `zebrasrv` are both driven by a master configuration file, which may refer to other subsidiary configuration files. By default, they try to use `zebra.cfg` in the working directory as the master file; but this can be changed using the `-c` option to specify an alternative master configuration file.

The master configuration file tells Zebra:

- Where to find subsidiary configuration files, including both those that are named explicitly and a few "magic" files such as `default.idx`, which specifies the default indexing rules.
- What record schemas to support. (Subsidiary files specifiy how to index the contents of records in those schemas, and what format to use when presenting records in those schemas to client software.)
- What attribute sets to recognise in searches. (Subsidiary files specify how to interpret the attributes in terms of the indexes that are created on the records.)
- Policy details such as what type of input format to expect when adding new records, what low-level indexing algorithm to use, how to identify potential duplicate records, etc.

Now let's see what goes in the `zebra.cfg` file for some example configurations.

Example 1: XML Indexing And Searching

This example shows how Zebra can be used with absolutely minimal configuration to index a body of XML (http://www.w3.org/XML/) documents, and search them using XPath (http://www.w3.org/TR/xpath) expressions to specify access points.

Go to the `examples/zthes` subdirectory of the distribution archive. There you will find a `Makefile` that will populate the `records` subdirectory with a file of Zthes (http://zthes.z3950.org/) records representing a taxonomic hierarchy of dinosaurs. (The records are generated from the family tree in the file `dino.tree`.) Type `make records/dino.xml` to make the XML data file. (Or you could just type `make dino` to build the XML data file, create the database and populate it with the taxonomic records all in one shot - but then you wouldn't learn anything, would you? :-)

Now we need to create a Zebra database to hold and index the XML records. We do this with the Zebra indexer, `zebraidx`, which is driven by the `zebra.cfg` configuration file. For our purposes, we don't need any special behaviour - we can use the defaults - so we can start with a minimal file that just tells `zebraidx` where to find the default indexing rules, and how to parse the records:

```
profilePath: .:../../tab
recordType: grs.sgml
```

That's all you need for a minimal Zebra configuration. Now you can roll the XML records into the database and build the indexes:

```
zebraidx update records
```

Now start the server. Like the indexer, its behaviour is controlled by the `zebra.cfg` file; and like the indexer, it works just fine with this minimal configuration.

```
zebrasrv
```

By default, the server listens on IP port number 9999, although this can easily be changed - see the Section called *Running the Z39.50 Server (zebrasrv)* in Chapter 7.

Now you can use the Z39.50 client program of your choice to execute XPath-based boolean queries and fetch the XML records that satisfy them:

```
$ yaz-client @:9999
Connecting...Ok.
Z> find @attr 1=/Zthes/termName Sauroposeidon
Number of hits: 1
Z> format xml
Z> show 1
<Zthes>
 <termId>22</termId>
 <termName>Sauroposeidon</termName>
 <termType>PT</termType>
 <termNote>The tallest known dinosaur (18m)</termNote>
 <relation>
  <relationType>BT</relationType>
  <termId>21</termId>
  <termName>Brachiosauridae</termName>
  <termType>PT</termType>
 </relation>

  <idzebra xmlns="http://www.indexdata.dk/zebra/">
<size>300</size>
<localnumber>23</localnumber>
<filename>records/dino.xml</filename>
  </idzebra>
</Zthes>
```

Now wasn't that nice and easy?

Example 2: Supporting Interoperable Searches

The problem with the previous example is that you need to know the structure of the documents in order to find them. For example, when we wanted to find the record for the taxon *Sauroposeidon*, we had to formulate a complex XPath `/Zthes/termName` which embodies the knowledge that taxon names are specified in a `<termName>` element inside the top-level `<Zthes>` element.

This is bad not just because it requires a lot of typing, but more significantly because it ties searching semantics to the physical structure of the searched records. You can't use the same search specification to search two databases if their internal representations are different. Consider an different taxonomy database in which the records have taxon names specified inside a `<name>` element nested within a `<identification>` element inside a top-level `<taxon>` element: then you'd need to search for them using `1=/taxon/identification/name`

How, then, can we build broadcasting Information Retrieval applications that look for records in many different databases? The Z39.50 protocol offers a powerful and general solution to this: abstract "access points". In the Z39.50 model, an access point is simply a point at which searches can be directed. Nothing is said about implementation: in a given database, an access point might be implemented as an index, a path into physical records, an algorithm for interrogating relational tables or whatever works. The only important thing point is that the semantics of an access point are fixed and well defined.

For convenience, access points are gathered into *attribute sets*. For example, the BIB-1 attribute set is supposed to contain bibliographic access points such as author, title, subject and ISBN; the GEO attribute set contains access points pertaining to geospatial information (bounding coordinates, stratum, latitude resolution, etc.); the CIMI attribute set contains access points to do with museum collections (provenance, inscriptions, etc.)

In practice, the BIB-1 attribute set has tended to be a dumping ground for all sorts of access points, so that, for example, it includes some geospatial access points as well as strictly bibliographic ones. Nevertheless, this model allows a layer of abstraction over the physical representation of records in databases.

In the BIB-1 attribute set, a taxon name is probably best interpreted as a title - that is, a phrase that identifies the item in question. BIB-1 represents title searches by access point 4. (See The BIB-1 Attribute Set Semantics (ftp://ftp.loc.gov/pub/z3950/defs/bib1.txt)) So we need to configure our dinosaur database so that searches for BIB-1 access point 4 look in the `<termName>` element, inside the top-level `<Zthes>` element.

This is a two-step process. First, we need to tell Zebra that we want to support the BIB-1 attribute set. Then we need to tell it which elements of its record pertain to access point 4.

We need to create an Abstract Syntax file named after the document element of the records we're working with, plus a `.abs` suffix - in this case, `Zthes.abs` - as follows:

```
attset zthes.att                                    ❶
attset bib1.att                                     ❷
xpath enable
systag sysno none

xelm /Zthes/termId              termId:w            ❸
xelm /Zthes/termName            termName:w,title:w  ❹
xelm /Zthes/termQualifier       termQualifier:w
xelm /Zthes/termType            termType:w
```

```
xelm /Zthes/termLanguage        termLanguage:w
xelm /Zthes/termNote            termNote:w
xelm /Zthes/termCreatedDate     termCreatedDate:w
xelm /Zthes/termCreatedBy       termCreatedBy:w
xelm /Zthes/termModifiedDate    termModifiedDate:w
xelm /Zthes/termModifiedBy      termModifiedBy:w
```

❶ Declare Thesausus attribute set. See `zthes.att`.

❷ Declare Bib-1 attribute set. See `bib1.att` in Zebra's `tab` directory.

❸ This xelm directive selects contents of nodes by XPath expression `/Zthes/termId`. The contents (CDATA) will be word searchable by Zthes attribute termId (value 1001).

❹ Make `termName` word searchable by both Zthes attribute termName (1002) and Bib-1 atttribute title (4).

After re-indexing, we can search the database using Bib-1 attribute, title, as follows:

```
Z> form xml
Z> f @attr 1=4 Eoraptor
Sent searchRequest.
Received SearchResponse.
Search was a success.
Number of hits: 1, setno 1
SearchResult-1: Eoraptor(1)
records returned: 0
Elapsed: 0.106896
Z> s
Sent presentRequest (1+1).
Records: 1
[Default]Record type: XML
<Zthes>
 <termId>2</termId>
 <termName>Eoraptor</termName>
 <termType>PT</termType>
 <termNote>The most basal known dinosaur</termNote>
 ...
```

Chapter 5. Administrating Zebra

Unlike many simpler retrieval systems, Zebra supports safe, incremental updates to an existing index.

Normally, when Zebra modifies the index it reads a number of records that you specify. Depending on your specifications and on the contents of each record one the following events take place for each record:

Insert

: The record is indexed as if it never occurred before. Either the Zebra system doesn't know how to identify the record or Zebra can identify the record but didn't find it to be already indexed.

Modify

: The record has already been indexed. In this case either the contents of the record or the location (file) of the record indicates that it has been indexed before.

Delete

: The record is deleted from the index. As in the update-case it must be able to identify the record.

Please note that in both the modify- and delete- case the Zebra indexer must be able to generate a unique key that identifies the record in question (more on this below).

To administrate the Zebra retrieval system, you run the `zebraidx` program. This program supports a number of options which are preceded by a dash, and a few commands (not preceded by dash).

Both the Zebra administrative tool and the Z39.50 server share a set of index files and a global configuration file. The name of the configuration file defaults to `zebra.cfg`. The configuration file includes specifications on how to index various kinds of records and where the other configuration files are located. `zebrasrv` and `zebraidx` *must* be run in the directory where the configuration file lives unless you indicate the location of the configuration file by option `-c`.

Record Types

Indexing is a per-record process, in which either insert/modify/delete will occur. Before a record is indexed search keys are extracted from whatever might be the layout the original record (sgml,html,text, etc..). The Zebra system currently supports two fundamental types of records: structured and simple text. To specify a particular extraction process, use either the command line option `-t` or specify a `recordType` setting in the configuration file.

The Zebra Configuration File

The Zebra configuration file, read by `zebraidx` and `zebrasrv` defaults to `zebra.cfg` unless specified by `-c` option.

You can edit the configuration file with a normal text editor. parameter names and values are separated by colons in the file. Lines starting with a hash sign (#) are treated as comments.

If you manage different sets of records that share common characteristics, you can organize the configuration settings for each type into "groups". When `zebraidx` is run and you wish to address a given group you specify the group name with the `-g` option. In this case settings that have the group name as their prefix will be used by `zebraidx`. If no `-g` option is specified, the settings without prefix are used.

In the configuration file, the group name is placed before the option name itself, separated by a dot (.). For instance, to set the record type for group `public` to `grs.sgml` (the SGML-like format for structured records) you would write:

```
public.recordType: grs.sgml
```

To set the default value of the record type to `text` write:

```
recordType: text
```

The available configuration settings are summarized below. They will be explained further in the following sections.

group .recordType[.*name*]: *`type`*

Specifies how records with the file extension *name* should be handled by the indexer. This option may also be specified as a command line option (`-t`). Note that if you do not specify a *name*, the setting applies to all files. In general, the record type specifier consists of the elements (each element separated by dot), *fundamental-type*, *file-read-type* and arguments. Currently, two fundamental types exist, `text` and `grs`.

group.recordId: *`record-id-spec`*

Specifies how the records are to be identified when updated. See the Section called *Locating Records*.

group.database: *`database`*

Specifies the Z39.50 database name.

group.storeKeys: *`boolean`*

Specifies whether key information should be saved for a given group of records. If you plan to update/delete this type of records later this should be specified as 1; otherwise it should be 0 (default), to save register space. See the Section called *Indexing with File Record IDs*.

group.storeData: *`boolean`*

Specifies whether the records should be stored internally in the Zebra system files. If you want to maintain the raw records yourself, this option should be false (0). If you want Zebra to take care of the records for you, it should be true(1).

register: *`register-location`*

Specifies the location of the various register files that Zebra uses to represent your databases. See the Section called *Register Location*.

shadow: *`register-location`*

Enables the *safe update* facility of Zebra, and tells the system where to place the required, temporary files. See the Section called *Safe Updating - Using Shadow Registers*.

lockDir: *`directory`*

Directory in which various lock files are stored.

keyTmpDir: *`directory`*

Directory in which temporary files used during zebraidx's update phase are stored.

setTmpDir: *`directory`*

Specifies the directory that the server uses for temporary result sets. If not specified `/tmp` will be used.

profilePath: *`path`*

Specifies a path of profile specification files. The path is composed of one or more directories separated by colon. Similar to PATH for UNIX systems.

attset: *`filename`*

Specifies the filename(s) of attribute set files for use in searching. At least the Bib-1 set should be loaded (`bib1.att`). The `profilePath` setting is used to look for the specified files. See the Section called *The Attribute Set (.att) Files* in Chapter 8

memMax: *`size`*

Specifies *`size`* of internal memory to use for the zebraidx program. The amount is given in megabytes - default is 4 (4 MB).

root: *`dir`*

Specifies a directory base for Zebra. All relative paths given (in profilePath, register, shadow) are based on this directory. This setting is useful if your Zebra server is running in a different directory from where `zebra.cfg` is located.

Locating Records

The default behavior of the Zebra system is to reference the records from their original location, i.e. where they were found when you ran `zebraidx`. That is, when a client wishes to retrieve a record following a search operation, the files are accessed from the place where you originally put them - if you remove the files (without running `zebraidx` again, the server will return diagnostic number 14 ("System error in presenting records") to the client.

If your input files are not permanent - for example if you retrieve your records from an outside source, or if they were temporarily mounted on a CD-ROM drive, you may want Zebra to make an internal copy of them. To do this, you specify 1 (true) in the `storeData` setting. When the Z39.50 server retrieves the records they will be read from the internal file structures of the system.

Indexing with no Record IDs (Simple Indexing)

If you have a set of records that are not expected to change over time you may can build your database without record IDs. This indexing method uses less space than the other methods and is simple to use.

To use this method, you simply omit the `recordId` entry for the group of files that you index. To add a set of records you use `zebraidx` with the `update` command. The `update` command will always add all of the records that it encounters to the index - whether they have already been indexed or not. If the set of indexed files change, you should delete all of the index files, and build a new index from scratch.

Consider a system in which you have a group of text files called `simple`. That group of records should belong to a Z39.50 database called `textbase`. The following `zebra.cfg` file will suffice:

```
profilePath: /usr/local/idzebra/tab
attset: bib1.att
simple.recordType: text
simple.database: textbase
```

Since the existing records in an index can not be addressed by their IDs, it is impossible to delete or modify records when using this method.

Indexing with File Record IDs

If you have a set of files that regularly change over time: Old files are deleted, new ones are added, or existing files are modified, you can benefit from using the *file ID* indexing methodology. Examples of this type of database might include an index of WWW resources, or a USENET news spool area. Briefly speaking, the file key methodology uses the directory paths of the individual records as a unique identifier for each record. To perform indexing of a directory with file keys, again, you specify the top-level directory after the `update` command. The command will recursively traverse the directories and compare each one with whatever have been indexed before in that same directory. If a file is new (not in the previous version of the directory) it is inserted into the registers; if a file was already indexed and it has been modified since the last update, the index is also modified; if a file has been removed since the last visit, it is deleted from the index.

The resulting system is easy to administrate. To delete a record you simply have to delete the corresponding file (say, with the `rm` command). And to add records you create new files (or directories with files). For your changes to take effect in the register you must run `zebraidx update` with the same directory root again. This mode of operation requires more disk space than simpler indexing methods, but it makes it easier for you to keep the index in sync with a frequently changing set of data. If you combine this system with the *safe update* facility (see below), you never have to take your server off-line for maintenance or register updating purposes.

To enable indexing with pathname IDs, you must specify `file` as the value of `recordId` in the configuration file. In addition, you should set `storeKeys` to `1`, since the Zebra indexer must save additional information about the contents of each record in order to modify the indexes correctly at a later time.

For example, to update records of group `esdd` located below `/data1/records/` you should type:

```
$ zebraidx -g esdd update /data1/records
```

The corresponding configuration file includes:

```
esdd.recordId: file
esdd.recordType: grs.sgml
esdd.storeKeys: 1
```

Note: You cannot start out with a group of records with simple indexing (no record IDs as in the previous section) and then later enable file record Ids. Zebra must know from the first time that you index the group that the files should be indexed with file record IDs.

You cannot explicitly delete records when using this method (using the `delete` command to `zebraidx`. Instead you have to delete the files from the file system (or move them to a different location) and then run `zebraidx` with the `update` command.

Indexing with General Record IDs

When using this method you construct an (almost) arbitrary, internal record key based on the contents of the record itself and other system information. If you have a group of records that explicitly associates an ID with each record, this method is convenient. For example, the record format may contain a title or a ID-number - unique within the group. In either case you specify the Z39.50 attribute set and use-attribute location in which this information is stored, and the system looks at that field to determine the identity of the record.

As before, the record ID is defined by the `recordId` setting in the configuration file. The value of the record ID specification consists of one or more tokens separated by whitespace. The resulting ID is represented in the index by concatenating the tokens and separating them by ASCII value (1).

There are three kinds of tokens:

Internal record info

The token refers to a key that is extracted from the record. The syntax of this token is `(` *set* `,` *use* `)`, where *set* is the attribute set name *use* is the name or value of the attribute.

System variable

The system variables are preceded by

```
$
```

and immediately followed by the system variable name, which may one of

group

Group name.

database

Current database specified.

type

Record type.

Constant string

A string used as part of the ID — surrounded by single- or double quotes.

For instance, the sample GILS records that come with the Zebra distribution contain a unique ID in the data tagged Control-Identifier. The data is mapped to the Bib-1 use attribute Identifier-standard (code 1007). To use this field as a record id, specify `(bib1,Identifier-standard)` as the value of the `recordId` in the configuration file. If you have other record types that uses the same field for a different purpose, you might add the record type (or group or database name) to the record id of the gils records as well, to prevent matches with other types of records. In this case the recordId might be set like this:

```
gils.recordId: $type (bib1,Identifier-standard)
```

(see the Section called *Configuring Your Data Model* in Chapter 8 for details of how the mapping between elements of your records and searchable attributes is established).

As for the file record ID case described in the previous section, updating your system is simply a matter of running `zebraidx` with the `update` command. However, the update with general keys is considerably slower than with file record IDs, since all files visited must be (re)read to discover their IDs.

As you might expect, when using the general record IDs method, you can only add or modify existing records with the `update` command. If you wish to delete records, you must use the, `delete` command, with a directory as a parameter. This will remove all records that match the files below that root directory.

Register Location

Normally, the index files that form dictionaries, inverted files, record info, etc., are stored in the directory where you run `zebraidx`. If you wish to store these, possibly large, files somewhere else, you must add

the `register` entry to the `zebra.cfg` file. Furthermore, the Zebra system allows its file structures to span multiple file systems, which is useful for managing very large databases.

The value of the `register` setting is a sequence of tokens. Each token takes the form:

dir`:`*size*`.`

The *dir* specifies a directory in which index files will be stored and the *size* specifies the maximum size of all files in that directory. The Zebra indexer system fills each directory in the order specified and use the next specified directories as needed. The *size* is an integer followed by a qualifier code, `b` for bytes, `k` for kilobytes. `M` for megabytes, `G` for gigabytes.

For instance, if you have allocated two disks for your register, and the first disk is mounted on `/d1` and has 2GB of free space and the second, mounted on `/d2` has 3.6 GB, you could put this entry in your configuration file:

```
register: /d1:2G /d2:3600M
```

Note that Zebra does not verify that the amount of space specified is actually available on the directory (file system) specified - it is your responsibility to ensure that enough space is available, and that other applications do not attempt to use the free space. In a large production system, it is recommended that you allocate one or more file system exclusively to the Zebra register files.

Safe Updating - Using Shadow Registers

Description

The Zebra server supports *updating* of the index structures. That is, you can add, modify, or remove records from databases managed by Zebra without rebuilding the entire index. Since this process involves modifying structured files with various references between blocks of data in the files, the update process is inherently sensitive to system crashes, or to process interruptions: Anything but a successfully completed update process will leave the register files in an unknown state, and you will essentially have no recourse but to re-index everything, or to restore the register files from a backup medium. Further, while the update process is active, users cannot be allowed to access the system, as the contents of the register files may change unpredictably.

You can solve these problems by enabling the shadow register system in Zebra. During the updating procedure, `zebraidx` will temporarily write changes to the involved files in a set of "shadow files", without modifying the files that are accessed by the active server processes. If the update procedure is interrupted by a system crash or a signal, you simply repeat the procedure - the register files have not been changed or damaged, and the partially written shadow files are automatically deleted before the new updating procedure commences.

At the end of the updating procedure (or in a separate operation, if you so desire), the system enters a "commit mode". First, any active server processes are forced to access those blocks that have been changed from the shadow files rather than from the main register files; the unmodified blocks are still accessed at their normal location (the shadow files are not a complete copy of the register files - they only

contain those parts that have actually been modified). If the commit process is interrupted at any point during the commit process, the server processes will continue to access the shadow files until you can repeat the commit procedure and complete the writing of data to the main register files. You can perform multiple update operations to the registers before you commit the changes to the system files, or you can execute the commit operation at the end of each update operation. When the commit phase has completed successfully, any running server processes are instructed to switch their operations to the new, operational register, and the temporary shadow files are deleted.

How to Use Shadow Register Files

The first step is to allocate space on your system for the shadow files. You do this by adding a `shadow` entry to the `zebra.cfg` file. The syntax of the `shadow` entry is exactly the same as for the `register` entry (see the Section called *Register Location*). The location of the shadow area should be *different* from the location of the main register area (if you have specified one - remember that if you provide no `register` setting, the default register area is the working directory of the server and indexing processes).

The following excerpt from a `zebra.cfg` file shows one example of a setup that configures both the main register location and the shadow file area. Note that two directories or partitions have been set aside for the shadow file area. You can specify any number of directories for each of the file areas, but remember that there should be no overlaps between the directories used for the main registers and the shadow files, respectively.

```
register: /d1:500M

shadow: /scratch1:100M /scratch2:200M
```

When shadow files are enabled, an extra command is available at the `zebraidx` command line. In order to make changes to the system take effect for the users, you'll have to submit a "commit" command after a (sequence of) update operation(s).

```
$ zebraidx update /d1/records
$ zebraidx commit
```

Or you can execute multiple updates before committing the changes:

```
$ zebraidx -g books update /d1/records  /d2/more-records
$ zebraidx -g fun update /d3/fun-records
$ zebraidx commit
```

If one of the update operations above had been interrupted, the commit operation on the last line would fail: `zebraidx` will not let you commit changes that would destroy the running register. You'll have to

rerun all of the update operations since your last commit operation, before you can commit the new changes.

Similarly, if the commit operation fails, `zebraidx` will not let you start a new update operation before you have successfully repeated the commit operation. The server processes will keep accessing the shadow files rather than the (possibly damaged) blocks of the main register files until the commit operation has successfully completed.

You should be aware that update operations may take slightly longer when the shadow register system is enabled, since more file access operations are involved. Further, while the disk space required for the shadow register data is modest for a small update operation, you may prefer to disable the system if you are adding a very large number of records to an already very large database (we use the terms *large* and *modest* very loosely here, since every application will have a different perception of size). To update the system without the use of the the shadow files, simply run `zebraidx` with the `-n` option (note that you do not have to execute the *commit* command of `zebraidx` when you temporarily disable the use of the shadow registers in this fashion. Note also that, just as when the shadow registers are not enabled, server processes will be barred from accessing the main register while the update procedure takes place.

Chapter 6. Running the Maintenance Interface (zebraidx)

The following is a complete reference to the command line interface to the `zebraidx` application.

Syntax

```
$ zebraidx [options] command [directory] ...
```

Options:

-t *`type`*

Update all files as *`type`*. Currently, the types supported are `text` and `grs.`*`subtype`*. If no *`subtype`* is provided for the GRS (General Record Structure) type, the canonical input format is assumed (see the Section called *Local Representation* in Chapter 8). Generally, it is probably advisable to specify the record types in the `zebra.cfg` file (see the Section called *Record Types* in Chapter 5), to avoid confusion at subsequent updates.

-c *`config-file`*

Read the configuration file *`config-file`* instead of `zebra.cfg`.

-g *`group`*

Update the files according to the group settings for *`group`* (see the Section called *The Zebra Configuration File* in Chapter 5).

-d *`database`*

The records located should be associated with the database name *`database`* for access through the Z39.50 server.

-l *`file`*

Write log messages to *`file`* instead of `stderr`.

-m *`mbytes`*

Use *`mbytes`* of memory before flushing keys to background storage. This setting affects performance when updating large databases.

-n

Disable the use of shadow registers for this operation (see the Section called *Safe Updating - Using Shadow Registers* in Chapter 5).

-s

Show analysis of the indexing process. The maintenance program works in a read-only mode and doesn’t change the state of the index. This options is very useful when you wish to test a new profile.

-V

Show Zebra version.

-v *`level`*

Set the log level to *`level`*. *`level`* should be one of `none`, `debug`, and `all`.

Commands

update *`directory`*

Update the register with the files contained in *`directory`*. If no directory is provided, a list of files is read from `stdin`. See Chapter 5.

delete *`directory`*

Remove the records corresponding to the files found under *`directory`* from the register.

commit

Write the changes resulting from the last `update` commands to the register. This command is only available if the use of shadow register files is enabled (see the Section called *Safe Updating - Using Shadow Registers* in Chapter 5).

Chapter 7. The Z39.50 Server

Running the Z39.50 Server (zebrasrv)

Syntax

```
zebrasrv [options] [listener-address ...]
```

Options

-a *APDU file*

Specify a file for dumping PDUs (for diagnostic purposes). The special name "-" sends output to `stderr`.

-c *config-file*

Read configuration information from *config-file*. The default configuration is `./zebra.cfg`.

-S

Don't fork on connection requests. This can be useful for symbolic-level debugging. The server can only accept a single connection in this mode.

-z

Use the Z39.50 protocol. Currently the only protocol supported. The option is retained for historical reasons, and for future extensions.

-l *logfile*

Specify an output file for the diagnostic messages. The default is to write this information to `stderr`.

-v *log-level*

The log level. Use a comma-separated list of members of the set {fatal,debug,warn,log,all,none}.

-u *username*

Set user ID. Sets the real UID of the server process to that of the given *username*. It's useful if you aren't comfortable with having the server run as root, but you need to start it as such to bind a privileged port.

-w *working-directory*

Change working directory.

-i

Run under the Internet superserver, `inetd`. Make sure you use the logfile option `-l` in conjunction with this mode and specify the `-l` option before any other options.

-t *timeout*

Set the idle session timeout (default 60 minutes).

-k *kilobytes*

Set the (approximate) maximum size of present response messages. Default is 1024 KB (1 MB).

A *listener-address* consists of an optional transport mode followed by a colon (:) followed by a listener address. The transport mode is either `ssl` or `tcp` (default).

For TCP, an address has the form

```
hostname | IP-number [: portnumber]
```

The port number defaults to 210 (standard Z39.50 port) for privileged users (root), and 9999 for normal users.

Examples

```
tcp:@

ssl:@:3000
```

In both cases, the special hostname "@" is mapped to the address INADDR_ANY, which causes the server to listen on any local interface. To start the server listening on the registered port for Z39.50, and to drop root privileges once the ports are bound, execute the server like this (from a root shell):

```
zebrasrv -u daemon @
```

You can replace `daemon` with another user, eg. your own account, or a dedicated IR server account.

The default behavior for `zebrasrv` is to establish a single TCP/IP listener, for the Z39.50 protocol, on port 9999.

Z39.50 Protocol Support and Behavior

Initialization

During initialization, the server will negotiate to version 3 of the Z39.50 protocol, and the option bits for Search, Present, Scan, NamedResultSets, and concurrentOperations will be set, if requested by the client. The maximum PDU size is negotiated down to a maximum of 1 MB by default.

Search

The supported query type are 1 and 101. All operators are currently supported with the restriction that only proximity units of type "word" are supported for the proximity operator. Queries can be arbitrarily complex. Named result sets are supported, and result sets can be used as operands without limitations. Searches may span multiple databases.

The server has full support for piggy-backed retrieval (see also the following section).

Use attributes are interpreted according to the attribute sets which have been loaded in the `zebra.cfg` file, and are matched against specific fields as specified in the `.abs` file which describes the profile of the records which have been loaded. If no Use attribute is provided, a default of Bib-1 Any is assumed.

If a *Structure* attribute of *Phrase* is used in conjunction with a *Completeness* attribute of *Complete (Sub)field*, the term is matched against the contents of the phrase (long word) register, if one exists for the given *Use* attribute. A phrase register is created for those fields in the `.abs` file that contains a `p`-specifier.

If *Structure=Phrase* is used in conjunction with *Incomplete Field* - the default value for *Completeness*, the search is directed against the normal word registers, but if the term contains multiple words, the term will only match if all of the words are found immediately adjacent, and in the given order. The word search is performed on those fields that are indexed as type `w` in the `.abs` file.

If the *Structure* attribute is *Word List*, *Free-form Text*, or *Document Text*, the term is treated as a natural-language, relevance-ranked query. This search type uses the word register, i.e. those fields that are indexed as type `w` in the `.abs` file.

If the *Structure* attribute is *Numeric String* the term is treated as an integer. The search is performed on those fields that are indexed as type `n` in the `.abs` file.

If the *Structure* attribute is *URx* the term is treated as a URX (URL) entity. The search is performed on those fields that are indexed as type `u` in the `.abs` file.

If the *Structure* attribute is *Local Number* the term is treated as native Zebra Record Identifier.

If the *Relation* attribute is *Equals* (default), the term is matched in a normal fashion (modulo truncation and processing of individual words, if required). If *Relation* is *Less Than*, *Less Than or Equal*, *Greater than*, or *Greater than or Equal*, the term is assumed to be numerical, and a standard regular expression is constructed to match the given expression. If *Relation* is *Relevance*, the standard natural-language query processor is invoked.

For the *Truncation* attribute, *No Truncation* is the default. *Left Truncation* is not supported. *Process # in search term* is supported, as is *Regxp-1*. *Regxp-2* enables the fault-tolerant (fuzzy) search. As a default, a single error (deletion, insertion, replacement) is accepted when terms are matched against the register contents.

Regular expressions

Each term in a query is interpreted as a regular expression if the truncation value is either *Regxp-1* (102) or *Regxp-2* (103). Both query types follow the same syntax with the operands:

x

Matches the character *x*.

.

Matches any character.

`[..]`

Matches the set of characters specified; such as `[abc]` or `[a-c]`.

and the operators:

x*

Matches *x* zero or more times. Priority: high.

x+

Matches *x* one or more times. Priority: high.

x?

Matches *x* zero or once. Priority: high.

xy

Matches *x*, then *y*. Priority: medium.

x|y

Matches either *x* or *y*. Priority: low.

The order of evaluation may be changed by using parentheses.

If the first character of the *Regxp-2* query is a plus character (+) it marks the beginning of a section with non-standard specifiers. The next plus character marks the end of the section. Currently Zebra only supports one specifier, the error tolerance, which consists one digit.

Since the plus operator is normally a suffix operator the addition to the query syntax doesn't violate the syntax for standard regular expressions.

Query examples

Phrase search for *information retrieval* in the title-register:

```
@attr 1=4 "information retrieval"
```

Ranked search for the same thing:

```
@attr 1=4 @attr 2=102 "Information retrieval"
```

Phrase search with a regular expression:

```
@attr 1=4 @attr 5=102 "informat.* retrieval"
```

Ranked search with a regular expression:

```
@attr 1=4 @attr 5=102 @attr 2=102 "informat.* retrieval"
```

In the GILS schema (`gils.abs`), the west-bounding-coordinate is indexed as type `n`, and is therefore searched by specifying *structure=Numeric String*. To match all those records with west-bounding-coordinate greater than -114 we use the following query:

```
@attr 4=109 @attr 2=5 @attr gils 1=2038 -114
```

Present

The present facility is supported in a standard fashion. The requested record syntax is matched against the ones supported by the profile of each record retrieved. If no record syntax is given, SUTRS is the default. The requested element set name, again, is matched against any provided by the relevant record profiles.

Scan

The attribute combinations provided with the termListAndStartPoint are processed in the same way as operands in a query (see above). Currently, only the term and the globalOccurrences are returned with the termInfo structure.

Sort

Z39.50 specifies three different types of sort criteria. Of these Zebra supports the attribute specification type in which case the use attribute specifies the "Sort register". Sort registers are created for those fields that are of type "sort" in the default.idx file. The corresponding character mapping file in default.idx specifies the ordinal of each character used in the actual sort.

Z39.50 allows the client to specify sorting on one or more input result sets and one output result set. Zebra supports sorting on one result set only which may or may not be the same as the output result set.

Close

If a Close PDU is received, the server will respond with a Close PDU with reason=FINISHED, no matter which protocol version was negotiated during initialization. If the protocol version is 3 or more, the server will generate a Close PDU under certain circumstances, including a session timeout (60 minutes by default), and certain kinds of protocol errors. Once a Close PDU has been sent, the protocol

association is considered broken, and the transport connection will be closed immediately upon receipt of further data, or following a short timeout.

Chapter 8. The Record Model

The Zebra system is designed to support a wide range of data management applications. The system can be configured to handle virtually any kind of structured data. Each record in the system is associated with a *record schema* which lends context to the data elements of the record. Any number of record schemas can coexist in the system. Although it may be wise to use only a single schema within one database, the system poses no such restrictions.

The record model described in this chapter applies to the fundamental, structured record type `grs`, introduced in the Section called *Record Types* in Chapter 5.

Records pass through three different states during processing in the system.

- When records are accessed by the system, they are represented in their local, or native format. This might be SGML or HTML files, News or Mail archives, MARC records. If the system doesn't already know how to read the type of data you need to store, you can set up an input filter by preparing conversion rules based on regular expressions and possibly augmented by a flexible scripting language (Tcl). The input filter produces as output an internal representation, a tree structure.
- When records are processed by the system, they are represented in a tree-structure, constructed by tagged data elements hanging off a root node. The tagged elements may contain data or yet more tagged elements in a recursive structure. The system performs various actions on this tree structure (indexing, element selection, schema mapping, etc.),
- Before transmitting records to the client, they are first converted from the internal structure to a form suitable for exchange over the network - according to the Z39.50 standard.

Local Representation

As mentioned earlier, Zebra places few restrictions on the type of data that you can index and manage. Generally, whatever the form of the data, it is parsed by an input filter specific to that format, and turned into an internal structure that Zebra knows how to handle. This process takes place whenever the record is accessed - for indexing and retrieval.

The RecordType parameter in the `zebra.cfg` file, or the `-t` option to the indexer tells Zebra how to process input records. Two basic types of processing are available - raw text and structured data. Raw text is just that, and it is selected by providing the argument *text* to Zebra. Structured records are all handled internally using the basic mechanisms described in the subsequent sections. Zebra can read structured records in many different formats. How this is done is governed by additional parameters after the "grs" keyword, separated by "." characters.

Four basic subtypes to the *grs* type are currently available:

grs.sgml

This is the canonical input format — described below. It is a simple SGML-like syntax.

grs.regx.*filter*

This enables a user-supplied input filter. The mechanisms of these filters are described below.

grs.tcl.*filter*

Similar to grs.regx but using Tcl for rules.

grs.marc.*abstract syntax*

This allows Zebra to read records in the ISO2709 (MARC) encoding standard. In this case, the last parameter *abstract syntax* names the `.abs` file (see below) which describes the specific MARC structure of the input record as well as the indexing rules.

grs.xml

This filter reads XML records. Only one record per file is supported. The filter is only available if Zebra/YAZ is compiled with EXPAT support.

Canonical Input Format

Although input data can take any form, it is sometimes useful to describe the record processing capabilities of the system in terms of a single, canonical input format that gives access to the full spectrum of structure and flexibility in the system. In Zebra, this canonical format is an "SGML-like" syntax.

To use the canonical format specify `grs.sgml` as the record type.

Consider a record describing an information resource (such a record is sometimes known as a *locator record*). It might contain a field describing the distributor of the information resource, which might in turn be partitioned into various fields providing details about the distributor, like this:

```
<Distributor>
  <Name> USGS/WRD </Name>
  <Organization> USGS/WRD </Organization>
  <Street-Address>
    U.S. GEOLOGICAL SURVEY, 505 MARQUETTE, NW
  </Street-Address>
  <City> ALBUQUERQUE </City>
  <State> NM </State>
  <Zip-Code> 87102 </Zip-Code>
  <Country> USA </Country>
  <Telephone> (505) 766-5560 </Telephone>
</Distributor>
```

The keywords surrounded by <...> are *tags*, while the sections of text in between are the *data elements*. A data element is characterized by its location in the tree that is made up by the nested elements. Each element is terminated by a closing tag - beginning with </, and containing the same symbolic tag-name as the corresponding opening tag. The general closing tag - `</>` - terminates the element started by the last opening tag. The structuring of elements is significant. The element *Telephone*, for instance, may be

indexed and presented to the client differently, depending on whether it appears inside the *Distributor* element, or some other, structured data element such a *Supplier* element.

Record Root

The first tag in a record describes the root node of the tree that makes up the total record. In the canonical input format, the root tag should contain the name of the schema that lends context to the elements of the record (see the Section called *Internal Representation*). The following is a GILS record that contains only a single element (strictly speaking, that makes it an illegal GILS record, since the GILS profile includes several mandatory elements - Zebra does not validate the contents of a record against the Z39.50 profile, however - it merely attempts to match up elements of a local representation with the given schema):

```
<gils>
   <title>Zen and the Art of Motorcycle Maintenance</title>
</gils>
```

Variants

Zebra allows you to provide individual data elements in a number of *variant forms*. Examples of variant forms are textual data elements which might appear in different languages, and images which may appear in different formats or layouts. The variant system in Zebra is essentially a representation of the variant mechanism of Z39.50-1995.

The following is an example of a title element which occurs in two different languages.

```
<title>
<var lang lang "eng">
Zen and the Art of Motorcycle Maintenance</>
<var lang lang "dan">
Zen og Kunsten at Vedligeholde en Motorcykel</>
</title>
```

The syntax of the *variant element* is `<var class type value>`. The available values for the *class* and *type* fields are given by the variant set that is associated with the current schema (see the Section called *The Variant Set (.var) Files*).

Variant elements are terminated by the general end-tag </>, by the variant end-tag </var>, by the appearance of another variant tag with the same *class* and *value* settings, or by the appearance of another, normal tag. In other words, the end-tags for the variants used in the example above could have been omitted.

Variant elements can be nested. The element

```
<title>
<var lang lang "eng"><var body iana "text/plain">
Zen and the Art of Motorcycle Maintenance
```

```
</title>
```

Associates two variant components to the variant list for the title element.

Given the nesting rules described above, we could write

```
<title>
<var body iana "text/plain>
<var lang lang "eng">
Zen and the Art of Motorcycle Maintenance
<var lang lang "dan">
Zen og Kunsten at Vedligeholde en Motorcykel
</title>
```

The title element above comes in two variants. Both have the IANA body type "text/plain", but one is in English, and the other in Danish. The client, using the element selection mechanism of Z39.50, can retrieve information about the available variant forms of data elements, or it can select specific variants based on the requirements of the end-user.

Input Filters

In order to handle general input formats, Zebra allows the operator to define filters which read individual records in their native format and produce an internal representation that the system can work with.

Input filters are ASCII files, generally with the suffix `.flt`. The system looks for the files in the directories given in the *profilePath* setting in the `zebra.cfg` files. The record type for the filter is `grs.regx.`*filter-filename* (fundamental type `grs`, file read type `regx`, argument *filter-filename*).

Generally, an input filter consists of a sequence of rules, where each rule consists of a sequence of expressions, followed by an action. The expressions are evaluated against the contents of the input record, and the actions normally contribute to the generation of an internal representation of the record.

An expression can be either of the following:

INIT

: The action associated with this expression is evaluated exactly once in the lifetime of the application, before any records are read. It can be used in conjunction with an action that initializes tables or other resources that are used in the processing of input records.

BEGIN

: Matches the beginning of the record. It can be used to initialize variables, etc. Typically, the *BEGIN* rule is also used to establish the root node of the record.

END

Matches the end of the record - when all of the contents of the record has been processed.

/pattern/

Matches a string of characters from the input record.

BODY

This keyword may only be used between two patterns. It matches everything between (not including) those patterns.

FINISH

The expression associated with this pattern is evaluated once, before the application terminates. It can be used to release system resources - typically ones allocated in the *INIT* step.

An action is surrounded by curly braces ({...}), and consists of a sequence of statements. Statements may be separated by newlines or semicolons (;). Within actions, the strings that matched the expressions immediately preceding the action can be referred to as $0, $1, $2, etc.

The available statements are:

begin *type [parameter ...]*

Begin a new data element. The type is one of the following:

record

Begin a new record. The following parameter should be the name of the schema that describes the structure of the record, eg. `gils` or `wais` (see below). The `begin record` call should precede any other use of the *begin* statement.

element

Begin a new tagged element. The parameter is the name of the tag. If the tag is not matched anywhere in the tagsets referenced by the current schema, it is treated as a local string tag.

variant

Begin a new node in a variant tree. The parameters are *class type value*.

data

Create a data element. The concatenated arguments make up the value of the data element. The option `-text` signals that the layout (whitespace) of the data should be retained for transmission. The option `-element` *tag* wraps the data up in the *tag*. The use of the `-element` option is equivalent to preceding the command with a *begin element* command, and following it with the *end* command.

end *[type]*

Close a tagged element. If no parameter is given, the last element on the stack is terminated. The first parameter, if any, is a type name, similar to the *begin* statement. For the *element* type, a tag name can be provided to terminate a specific tag.

The following input filter reads a Usenet news file, producing a record in the WAIS schema. Note that the body of a news posting is separated from the list of headers by a blank line (or rather a sequence of two newline characters.

```
BEGIN               { begin record wais }

/^From:/ BODY /$/    { data -element name $1 }
/^Subject:/ BODY /$/ { data -element title $1 }
/^Date:/ BODY /$/    { data -element lastModified $1 }
/\n\n/ BODY END      {
begin element bodyOfDisplay
begin variant body iana "text/plain"
data -text $1
end record
}
```

If Zebra is compiled with support for Tcl (Tool Command Language) enabled, the statements described above are supplemented with a complete scripting environment, including control structures (conditional expressions and loop constructs), and powerful string manipulation mechanisms for modifying the elements of a record. Tcl is a popular scripting environment, with several tutorials available both online and in hardcopy.

Internal Representation

When records are manipulated by the system, they’re represented in a tree-structure, with data elements at the leaf nodes, and tags or variant components at the non-leaf nodes. The root-node identifies the schema that lends context to the tagging and structuring of the record. Imagine a simple record, consisting of a ’title’ element and an ’author’ element:

```
ROOT
   TITLE     "Zen and the Art of Motorcycle Maintenance"
   AUTHOR    "Robert Pirsig"
```

A slightly more complex record would have the author element consist of two elements, a surname and a first name:

```
ROOT
```

```
TITLE  "Zen and the Art of Motorcycle Maintenance"
AUTHOR
   FIRST-NAME "Robert"
   SURNAME    "Pirsig"
```

The root of the record will refer to the record schema that describes the structuring of this particular record. The schema defines the element tags (TITLE, FIRST-NAME, etc.) that may occur in the record, as well as the structuring (SURNAME should appear below AUTHOR, etc.). In addition, the schema establishes element set names that are used by the client to request a subset of the elements of a given record. The schema may also establish rules for converting the record to a different schema, by stating, for each element, a mapping to a different tag path.

Tagged Elements

A data element is characterized by its tag, and its position in the structure of the record. For instance, while the tag "telephone number" may be used different places in a record, we may need to distinguish between these occurrences, both for searching and presentation purposes. For instance, while the phone numbers for the "customer" and the "service provider" are both representatives for the same type of resource (a telephone number), it is essential that they be kept separate. The record schema provides the structure of the record, and names each data element (defined by the sequence of tags - the tag path - by which the element can be reached from the root of the record).

Variants

The children of a tag node may be either more tag nodes, a data node (possibly accompanied by tag nodes), or a tree of variant nodes. The children of variant nodes are either more variant nodes or a data node (possibly accompanied by more variant nodes). Each leaf node, which is normally a data node, corresponds to a *variant form* of the tagged element identified by the tag which parents the variant tree. The following title element occurs in two different languages:

```
VARIANT LANG=ENG  "War and Peace"
TITLE
VARIANT LANG=DAN  "Krig og Fred"
```

Which of the two elements are transmitted to the client by the server depends on the specifications provided by the client, if any.

In practice, each variant node is associated with a triple of class, type, value, corresponding to the variant mechanism of Z39.50.

Data Elements

Data nodes have no children (they are always leaf nodes in the record tree).

Configuring Your Data Model

The following sections describe the configuration files that govern the internal management of data records. The system searches for the files in the directories specified by the *profilePath* setting in the `zebra.cfg` file.

The Abstract Syntax

The abstract syntax definition (also known as an Abstract Record Structure, or ARS) is the focal point of the record schema description. For a given schema, the ABS file may state any or all of the following:

- The object identifier of the Z39.50 schema associated with the ARS, so that it can be referred to by the client.
- The attribute set (which can possibly be a compound of multiple sets) which applies in the profile. This is used when indexing and searching the records belonging to the given profile.
- The tag set (again, this can consist of several different sets). This is used when reading the records from a file, to recognize the different tags, and when transmitting the record to the client - mapping the tags to their numerical representation, if they are known.
- The variant set which is used in the profile. This provides a vocabulary for specifying the *forms* of data that appear inside the records.
- Element set names, which are a shorthand way for the client to ask for a subset of the data elements contained in a record. Element set names, in the retrieval module, are mapped to *element specifications*, which contain information equivalent to the *Espec-1* syntax of Z39.50.
- Map tables, which may specify mappings to *other* database profiles, if desired.
- Possibly, a set of rules describing the mapping of elements to a MARC representation.
- A list of element descriptions (this is the actual ARS of the schema, in Z39.50 terms), which lists the ways in which the various tags can be used and organized hierarchically.

Several of the entries above simply refer to other files, which describe the given objects.

The Configuration Files

This section describes the syntax and use of the various tables which are used by the retrieval module.

The number of different file types may appear daunting at first, but each type corresponds fairly clearly to a single aspect of the Z39.50 retrieval facilities. Further, the average database administrator, who is simply reusing an existing profile for which tables already exist, shouldn’t have to worry too much about the contents of these tables.

Generally, the files are simple ASCII files, which can be maintained using any text editor. Blank lines, and lines beginning with a (#) are ignored. Any characters on a line followed by a (#) are also ignored. All other lines contain *directives*, which provide some setting or value to the system. Generally, settings are characterized by a single keyword, identifying the setting, followed by a number of parameters. Some settings are repeatable (r), while others may occur only once in a file. Some settings are optional (o), while others again are mandatory (m).

The Abstract Syntax (.abs) Files

The name of this file type is slightly misleading in Z39.50 terms, since, apart from the actual abstract syntax of the profile, it also includes most of the other definitions that go into a database profile.

When a record in the canonical, SGML-like format is read from a file or from the database, the first tag of the file should reference the profile that governs the layout of the record. If the first tag of the record is, say, `<gils>`, the system will look for the profile definition in the file `gils.abs`. Profile definitions are cached, so they only have to be read once during the lifespan of the current process.

When writing your own input filters, the *record-begin* command introduces the profile, and should always be called first thing when introducing a new record.

The file may contain the following directives:

name *`symbolic-name`*

(m) This provides a shorthand name or description for the profile. Mostly useful for diagnostic purposes.

reference *`OID-name`*

(m) The reference name of the OID for the profile. The reference names can be found in the *util* module of YAZ.

attset *`filename`*

(m) The attribute set that is used for indexing and searching records belonging to this profile.

tagset *`filename`*

(o) The tag set (if any) that describe that fields of the records.

varset *`filename`*

(o) The variant set used in the profile.

maptab *`filename`*

(o,r) This points to a conversion table that might be used if the client asks for the record in a different schema from the native one.

marc *`filename`*

(o) Points to a file containing parameters for representing the record contents in the ISO2709 syntax. Read the description of the MARC representation facility below.

esetname *`name filename`*

(o,r) Associates the given element set name with an element selection file. If an (@) is given in place of the filename, this corresponds to a null mapping for the given element set name.

any *`tags`*

(o) This directive specifies a list of attributes which should be appended to the attribute list given for each element. The effect is to make every single element in the abstract syntax searchable by way of the given attributes. This directive provides an efficient way of supporting free-text

searching across all elements. However, it does increase the size of the index significantly. The attributes can be qualified with a structure, as in the *elm* directive below.

elm *path name attributes*

(o,r) Adds an element to the abstract record syntax of the schema. The *path* follows the syntax which is suggested by the Z39.50 document - that is, a sequence of tags separated by slashes (/). Each tag is given as a comma-separated pair of tag type and -value surrounded by parenthesis. The *name* is the name of the element, and the *attributes* specifies which attributes to use when indexing the element in a comma-separated list. A ! in place of the attribute name is equivalent to specifying an attribute name identical to the element name. A - in place of the attribute name specifies that no indexing is to take place for the given element. The attributes can be qualified with *field types* to specify which character set should govern the indexing procedure for that field. The same data element may be indexed into several different fields, using different character set definitions. See the the Section called *Field Structure and Character Sets*. The default field type is `w` for *word*.

xelm *xpath attributes*

Specifies indexing for record nodes given by *xpath*. Unlike directive elm, this directive allows you to index attribute contents. The *xpath* uses a syntax similar to XPath. The *attributes* have same syntax and meaning as directive elm, except that operator ! refers to the nodes selected by *xpath*.

encoding *encodingname*

This directive specifies character encoding for external records. For records such as XML that specifies encoding within the file via a header this directive is ignored. If neither this directive is given, nor an encoding is set within external records, ISO-8859-1 encoding is assumed.

xpath `enable`/`disable`

If this directive is followed by `enable`, then extra indexing is performed to allow for XPath-like queries. If this directive is not specified - equivalent to `disable` - no extra XPath-indexing is performed.

systag *systemTag actualTag*

Specifies what information, if any, Zebra should automatically include in retrieval records for the “system fields” that it supports. *systemTag* may be any of the following:

`rank`

An integer indicating the relevance-ranking score assigned to the record.

`sysno`

An automatically generated identifier for the record, unique within this database. It is represented by the `<localControlNumber>` element in XML and the `(1,14)` tag in GRS-1.

`size`

The size, in bytes, of the retrieved record.

The *actualTag* parameter may be none to indicate that the named element should be omitted from retrieval records.

Note: The mechanism for controlling indexing is not adequate for complex databases, and will probably be moved into a separate configuration table eventually.

The following is an excerpt from the abstract syntax file for the GILS profile.

```
name gils
reference GILS-schema
attset gils.att
tagset gils.tag
varset var1.var

maptab gils-usmarc.map

# Element set names

esetname VARIANT gils-variant.est  # for WAIS-compliance
esetname B gils-b.est
esetname G gils-g.est
esetname F @

elm (1,10)              rank                       -
elm (1,12)              url                        -
elm (1,14)              localControlNumber     Local-number
elm (1,16)              dateOfLastModification Date/time-last-modified
elm (2,1)               title                      w:!,p:!
elm (4,1)               controlIdentifier      Identifier-standard
elm (2,6)               abstract               Abstract
elm (4,51)              purpose                    !
elm (4,52)              originator                 -
elm (4,53)              accessConstraints          !
elm (4,54)              useConstraints             !
elm (4,70)              availability               -
elm (4,70)/(4,90)       distributor                -
elm (4,70)/(4,90)/(2,7) distributorName            !
elm (4,70)/(4,90)/(2,10 distributorOrganization    !
elm (4,70)/(4,90)/(4,2) distributorStreetAddress   !
elm (4,70)/(4,90)/(4,3) distributorCity            !
```

The Attribute Set (.att) Files

This file type describes the *Use* elements of an attribute set. It contains the following directives.

name *symbolic-name*

(m) This provides a shorthand name or description for the attribute set. Mostly useful for diagnostic purposes.

reference *OID-name*

(m) The reference name of the OID for the attribute set. The reference names can be found in the *util* module of *YAZ*.

include *filename*

(o,r) This directive is used to include another attribute set as a part of the current one. This is used when a new attribute set is defined as an extension to another set. For instance, many new attribute sets are defined as extensions to the *bib-1* set. This is an important feature of the retrieval system of Z39.50, as it ensures the highest possible level of interoperability, as those access points of your database which are derived from the external set (say, bib-1) can be used even by clients who are unaware of the new set.

att *att-value att-name [local-value]*

(o,r) This repeatable directive introduces a new attribute to the set. The attribute value is stored in the index (unless a *local-value* is given, in which case this is stored). The name is used to refer to the attribute from the *abstract syntax*.

This is an excerpt from the GILS attribute set definition. Notice how the file describing the *bib-1* attribute set is referenced.

```
name gils
reference GILS-attset
include bib1.att

att 2001 distributorName
att 2002 indextermsControlled
att 2003 purpose
att 2004 accessConstraints
att 2005 useConstraints
```

The Tag Set (.tag) Files

This file type defines the tagset of the profile, possibly by referencing other tag sets (most tag sets, for instance, will include tagsetG and tagsetM from the Z39.50 specification. The file may contain the following directives.

name *symbolic-name*

(m) This provides a shorthand name or description for the tag set. Mostly useful for diagnostic purposes.

reference *OID-name*

(o) The reference name of the OID for the tag set. The reference names can be found in the *util* module of *YAZ*. The directive is optional, since not all tag sets are registered outside of their schema.

type *integer*

(m) The type number of the tagset within the schema profile (note: this specification really should belong to the .abs file. This will be fixed in a future release).

include *filename*

(o,r) This directive is used to include the definitions of other tag sets into the current one.

tag *number names type*

(o,r) Introduces a new tag to the set. The *number* is the tag number as used in the protocol (there is currently no mechanism for specifying string tags at this point, but this would be quick work to add). The *names* parameter is a list of names by which the tag should be recognized in the input file format. The names should be separated by slashes (/). The *type* is the recommended data type of the tag. It should be one of the following:

- structured
- string
- numeric
- bool
- oid
- generalizedtime
- intunit
- int
- octetstring
- null

The following is an excerpt from the TagsetG definition file.

```
name tagsetg
reference TagsetG
type 2

tag 1 title string
tag 2 author string
tag 3 publicationPlace string
tag 4 publicationDate string
```

```
tag 5 documentId string
tag 6 abstract string
tag 7 name string
tag 8 date generalizedtime
tag 9 bodyOfDisplay string
tag 10 organization string
```

The Variant Set (.var) Files

The variant set file is a straightforward representation of the variant set definitions associated with the protocol. At present, only the *Variant-1* set is known.

These are the directives allowed in the file.

name *symbolic-name*

(m) This provides a shorthand name or description for the variant set. Mostly useful for diagnostic purposes.

reference *OID-name*

(o) The reference name of the OID for the variant set, if one is required. The reference names can be found in the *util* module of *YAZ*.

class *integer class-name*

(m,r) Introduces a new class to the variant set.

type *integer type-name datatype*

(m,r) Addes a new type to the current class (the one introduced by the most recent *class* directive). The type names belong to the same name space as the one used in the tag set definition file.

The following is an excerpt from the file describing the variant set *Variant-1*.

```
name variant-1
reference Variant-1

class 1 variantId

type 1 variantId octetstring

class 2 body

type 1 iana string
type 2 z39.50 string
type 3 other string
```

The Element Set (.est) Files

The element set specification files describe a selection of a subset of the elements of a database record. The element selection mechanism is equivalent to the one supplied by the *Espec-1* syntax of the Z39.50 specification. In fact, the internal representation of an element set specification is identical to the *Espec-1* structure, and we'll refer you to the description of that structure for most of the detailed semantics of the directives below.

> **Note:** Not all of the Espec-1 functionality has been implemented yet. The fields that are mentioned below all work as expected, unless otherwise is noted.

The directives available in the element set file are as follows:

defaultVariantSetId *OID-name*

(o) If variants are used in the following, this should provide the name of the variantset used (it's not currently possible to specify a different set in the individual variant request). In almost all cases (certainly all profiles known to us), the name `Variant-1` should be given here.

defaultVariantRequest *variant-request*

(o) This directive provides a default variant request for use when the individual element requests (see below) do not contain a variant request. Variant requests consist of a blank-separated list of variant components. A variant compont is a comma-separated, parenthesized triple of variant class, type, and value (the two former values being represented as integers). The value can currently only be entered as a string (this will change to depend on the definition of the variant in question). The special value (@) is interpreted as a null value, however.

simpleElement *path ['variant' variant-request]*

(o,r) This corresponds to a simple element request in *Espec-1*. The path consists of a sequence of tag-selectors, where each of these can consist of either:

- A simple tag, consisting of a comma-separated type-value pair in parenthesis, possibly followed by a colon (:) followed by an occurrences-specification (see below). The tag-value can be a number or a string. If the first character is an apostrophe ('), this forces the value to be interpreted as a string, even if it appears to be numerical.
- A WildThing, represented as a question mark (?), possibly followed by a colon (:) followed by an occurrences specification (see below).
- A WildPath, represented as an asterisk (*). Note that the last element of the path should not be a wildPath (wildpaths don't work in this version).

The occurrences-specification can be either the string `all`, the string `last`, or an explicit value-range. The value-range is represented as an integer (the starting point), possibly followed by a plus (+) and a second integer (the number of elements, default being one).

The variant-request has the same syntax as the defaultVariantRequest above. Note that it may sometimes be useful to give an empty variant request, simply to disable the default for a specific set of fields (we aren't certain if this is proper *Espec-1*, but it works in this implementation).

The following is an example of an element specification belonging to the GILS profile.

```
simpleelement (1,10)
simpleelement (1,12)
simpleelement (2,1)
simpleelement (1,14)
simpleelement (4,1)
simpleelement (4,52)
```

The Schema Mapping (.map) Files

Sometimes, the client might want to receive a database record in a schema that differs from the native schema of the record. For instance, a client might only know how to process WAIS records, while the database record is represented in a more specific schema, such as GILS. In this module, a mapping of data to one of the MARC formats is also thought of as a schema mapping (mapping the elements of the record into fields consistent with the given MARC specification, prior to actually converting the data to the ISO2709). This use of the object identifier for USMARC as a schema identifier represents an overloading of the OID which might not be entirely proper. However, it represents the dual role of schema and record syntax which is assumed by the MARC family in Z39.50.

These are the directives of the schema mapping file format:

targetName *name*

(m) A symbolic name for the target schema of the table. Useful mostly for diagnostic purposes.

targetRef *OID-name*

(m) An OID name for the target schema. This is used, for instance, by a server receiving a request to present a record in a different schema from the native one. The name, again, is found in the *oid* module of *YAZ*.

map *element-name target-path*

(o,r) Adds an element mapping rule to the table.

The MARC (ISO2709) Representation (.mar) Files

This file provides rules for representing a record in the ISO2709 format. The rules pertain mostly to the values of the constant-length header of the record.

Field Structure and Character Sets

In order to provide a flexible approach to national character set handling, Zebra allows the administrator to configure the set up the system to handle any 8-bit character set — including sets that require multi-octet diacritics or other multi-octet characters. The definition of a character set includes a specification of the permissible values, their sort order (this affects the display in the SCAN function), and relationships between upper- and lowercase characters. Finally, the definition includes the specification of space characters for the set.

The operator can define different character sets for different fields, typical examples being standard text fields, numerical fields, and special-purpose fields such as WWW-style linkages (URx).

The field types, and hence character sets, are associated with data elements by the .abs files (see above). The file `default.idx` provides the association between field type codes (as used in the .abs files) and the character map files (with the .chr suffix). The format of the .idx file is as follows

index *field type code*

This directive introduces a new search index code. The argument is a one-character code to be used in the .abs files to select this particular index type. An index, roughly, corresponds to a particular structure attribute during search. Refer to the Section called *Search* in Chapter 7.

sort *field code type*

This directive introduces a sort index. The argument is a one-character code to be used in the .abs fie to select this particular index type. The corresponding use attribute must be used in the sort request to refer to this particular sort index. The corresponding character map (see below) is used in the sort process.

completeness *boolean*

This directive enables or disables complete field indexing. The value of the *boolean* should be 0 (disable) or 1. If completeness is enabled, the index entry will contain the complete contents of the field (up to a limit), with words (non-space characters) separated by single space characters (normalized to " " on display). When completeness is disabled, each word is indexed as a separate entry. Complete subfield indexing is most useful for fields which are typically browsed (eg. titles, authors, or subjects), or instances where a match on a complete subfield is essential (eg. exact title searching). For fields where completeness is disabled, the search engine will interpret a search containing space characters as a word proximity search.

charmap *filename*

This is the filename of the character map to be used for this index for field type.

The contents of the character map files are structured as follows:

lowercase *value-set*

This directive introduces the basic value set of the field type. The format is an ordered list (without spaces) of the characters which may occur in "words" of the given type. The order of the entries in the list determines the sort order of the index. In addition to single characters, the following combinations are legal:

- Backslashes may be used to introduce three-digit octal, or two-digit hex representations of single characters (preceded by `x`). In addition, the combinations \\, \\r, \\n, \\t, \\s (space — remember that real space-characters may not occur in the value definition), and \\ are recognized, with their usual interpretation.
- Curly braces {} may be used to enclose ranges of single characters (possibly using the escape convention described in the preceding point), eg. {a-z} to introduce the standard range of ASCII characters. Note that the interpretation of such a range depends on the concrete representation in your local, physical character set.
- paranthesises () may be used to enclose multi-byte characters - eg. diacritics or special national combinations (eg. Spanish "ll"). When found in the input stream (or a search term), these characters are viewed and sorted as a single character, with a sorting value depending on the position of the group in the value statement.

uppercase *value-set*

This directive introduces the upper-case equivalencis to the value set (if any). The number and order of the entries in the list should be the same as in the `lowercase` directive.

space *value-set*

This directive introduces the character which separate words in the input stream. Depending on the completeness mode of the field in question, these characters either terminate an index entry, or delimit individual "words" in the input stream. The order of the elements is not significant — otherwise the representation is the same as for the `uppercase` and `lowercase` directives.

map *value-set target*

This directive introduces a mapping between each of the members of the value-set on the left to the character on the right. The character on the right must occur in the value set (the `lowercase` directive) of the character set, but it may be a paranthesis-enclosed multi-octet character. This directive may be used to map diacritics to their base characters, or to map HTML-style character-representations to their natural form, etc.

Exchange Formats

Converting records from the internal structure to en exchange format is largely an automatic process. Currently, the following exchange formats are supported:

- GRS-1. The internal representation is based on GRS-1/XML, so the conversion here is straightforward. The system will create applied variant and supported variant lists as required, if a record contains variant information.
- XML. The internal representation is based on GRS-1/XML so the mapping is trivial. Note that XML schemas, preprocessing instructions and comments are not part of the internal representation and therefore will never be part of a generated XML record. Future versions of the Zebra will support that.
- SUTRS. Again, the mapping is fairly straightforward. Indentation is used to show the hierarchical structure of the record. All "GRS" type records support both the GRS-1 and SUTRS representations.
- ISO2709-based formats (USMARC, etc.). Only records with a two-level structure (corresponding to fields and subfields) can be directly mapped to ISO2709. For records with a different structuring (eg., GILS), the representation in a structure like USMARC involves a schema-mapping (see the Section called *The Schema Mapping (.map) Files*), to an "implied" USMARC schema (implied, because there is no formal schema which specifies the use of the USMARC fields outside of ISO2709). The resultant, two-level record is then mapped directly from the internal representation to ISO2709. See the GILS schema definition files for a detailed example of this approach.
- Explain. This representation is only available for records belonging to the Explain schema.
- Summary. This ASN-1 based structure is only available for records belonging to the Summary schema - or schema which provide a mapping to this schema (see the description of the schema mapping facility above).
- SOIF. Support for this syntax is experimental, and is currently keyed to a private Index Data OID (1.2.840.10003.5.1000.81.2). All abstract syntaxes can be mapped to the SOIF format, although nested elements are represented by concatenation of the tag names at each level.

Appendix A. License

GNU General Public License

gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections

1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further

restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free

Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Appendix B. About Index Data and the Zebra Server

Index Data is a consulting and software-development enterprise that specializes in library and information management systems. Our interests and expertise span a broad range of related fields, and one of our primary, long-term objectives is the development of a powerful information management system with open network interfaces and hyper-media capabilities.

We make this software available free of charge, on a fairly unrestrictive license; as a service to the networking community, and to further the development of quality software for open network communication.

We'll be happy to answer questions about the software, and about ourselves in general.

Index Data Aps
Købmagergade 43
1150 Copenhagen K
Denmark
Phone +45 3341 0100
Fax +45 3341 0101
Email <info@indexdata.dk>

indexdata.dk (http://indexdata.dk/)

The *Random House College Dictionary*, 1975 edition offers this definition of the word "Zebra":

[Zebra, n., any of several horselike, African mammals of the genus Equus, having a characteristic pattern of black or dark-brown stripes on a whitish background.]