YAZ User's Guide and Reference

COLLABORATORS

	TITLE : YAZ User's Guide and Reference	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Sebastian Hammer, Adam Dickmeiss, Mike Taylor, Heikki Levanto, and Dennis Schafroth	May 30, 2017	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

List of Figures

List of Tables

Abstract

This document is the programmer's guide and reference to the YAZ package version 5.22.0. YAZ is a compact toolkit that provides access to the Z39.50 and SRU/Solr protocols, as well as a set of higher-level tools for implementing the server and client roles, respectively. The documentation can be used on its own, or as a reference when looking at the example applications provided with the package.

Chapter 1

Introduction

YAZ is a C/C++ library for information retrieval applications using the Z39.50/SRU/Solr protocols for information retrieval.

Properties of YAZ:

- Complete Z39.50 version 3 support. Amendments and Z39.50-2002 revision is supported.
- Supports SRU GET/POST/SOAP version 1.1, 1.2 and 2.0 (over HTTP and HTTPS).
- Includes BER encoders/decoders for the ISO ILL protocol.
- Supports Apache Solr Web Service version 1.4.x (client side only)
- Supports the following transports: BER over TCP/IP (RFC1729), BER over unix local socket, and HTTP 1.1.
- Secure Socket Layer support using GnuTLS. If enabled, YAZ uses HTTPS transport (for SOAP) or "Secure BER" (for Z39.50).
- Offers ZOOM C API implementing Z39.50, SRU and Solr Web Service.
- The YAZ library offers a set of useful utilities related to the protocols, such as MARC (ISO2709) parser, CCL (ISO8777) parser, CQL parser, memory management routines, character set conversion.
- Portable code. YAZ compiles out-of-the box on most Unixes and on Windows using Microsoft Visual C++.
- Fast operation. The C based BER encoders/decoders as well as the server component of YAZ is very fast.
- Liberal license that allows for commercial use of YAZ.

Reading this Manual

Most implementors only need to read a fraction of the material in this manual, so a quick walkthrough of the chapters is in order.

- Chapter 2 contains installation instructions for YAZ. You don't need to read this if you expect to download YAZ binaries. However, the chapter contains information about how to make *your* application link with YAZ.

- Chapter 3 describes the ZOOM API of YAZ. This is definitely worth reading if you wish to develop a Z39.50/SRU client.

- Chapter 4 describes the generic frontend server and explains how to develop server Z39.50/SRU applications for YAZ. Obviously worth reading if you're to develop a server.

- yaz-client(1) describes how to use the YAZ Z39.50 client. If you're a developer and wish to test your server or a server from another party, you might find this chapter useful.

- Chapter 5 documents the most commonly used Z39.50 C data structures offered by the YAZ API. Client developers using ZOOM and non-Z39.50 implementors may skip this.

- Chapter 6 describes how SRU and SOAP is used in YAZ. Only if you're developing SRU applications this section is a must.

- Chapter 7 contains sections for the various tools offered by YAZ. Scan through the material quickly and see what's relevant to you! SRU implementors might find the CQL section particularly useful.

- Chapter 8 goes through the details of the ODR module which is the work horse that encodes and decodes BER packages. Implementors using ZOOM only, do *not* need to read this. Most other Z39.50 implementors only need to read the first two sections (Section 8.1 and Section 8.2).

- Chapter 9 describes the network layer module COMSTACK. Implementors using ZOOM or the generic frontend server may skip this. Others, presumably, handling client/server communication on their own should read this.

The API

The YAZ toolkit offers several different levels of access to the ISO23950/Z39.50, ILL and SRU protocols. The level that you need to use depends on your requirements, and the role (server or client) that you want to implement. If you're developing a client application you should consider the ZOOM API. It is, by far, the easiest way to develop clients in C. Server implementers should consider the generic frontend server. None of those high-level APIs support the whole protocol, but they do include most facilities used in existing Z39.50 applications.

If you're using 'exotic' functionality (meaning anything not included in the high-level APIs), developing non-standard extensions to Z39.50 or you're going to develop an ILL application, you'll have to learn the lower level APIs of YAZ.

The YAZ toolkit modules are shown in figure Figure 1.1.

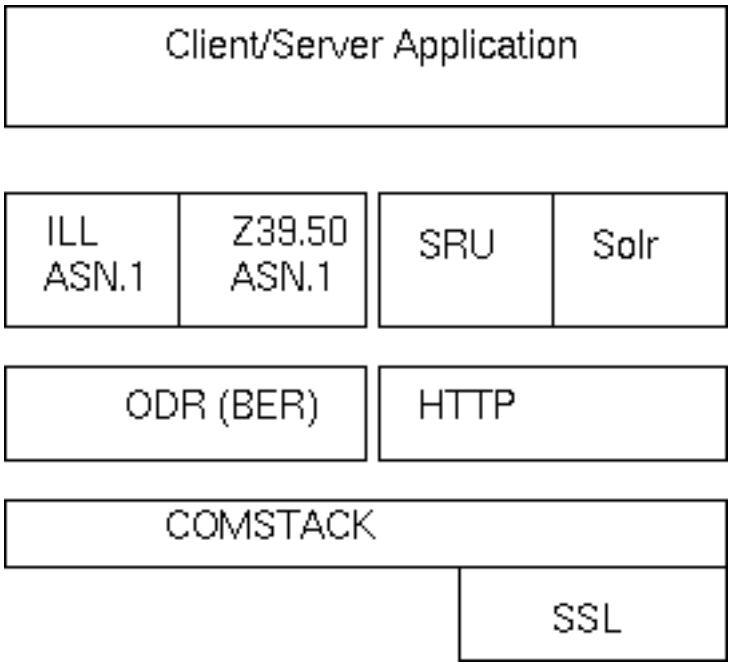

Figure 1.1: YAZ layers

There are four layers.

- A client or server application (or both). This layer includes ZOOM and the generic frontend server.
- The second layer provides a C representation of the protocol units (packages) for Z39.50 ASN.1, ILL ASN.1, SRU.
- The third layer encodes and decodes protocol data units to simple packages (buffer with certain length). The ODR module encodes and decodes BER whereas the HTTP modules encodes and decodes HTTP requests/responses.
- The lowest layer is COMSTACK which exchanges the encoded packages with a peer process over a network.

The Z39.50 ASN.1 module represents the ASN.1 definition of the Z39.50 protocol. It establishes a set of type and structure definitions, with one structure for each of the top-level PDUs, and one structure or type for each of the contained ASN.1 types. For primitive types, or other types that are defined by the ASN.1 standard itself (such as the EXTERNAL type), the C representation is provided by the ODR (Open Data Representation) subsystem.

ODR is a basic mechanism for representing an ASN.1 type in the C programming language, and for implementing BER encoders and decoders for values of that type. The types defined in the Z39.50 ASN.1 module generally have the prefix `Z_`, and a suffix corresponding to the name of the type in the ASN.1 specification of the protocol (generally Z39.50-1995). In the case of base types (those originating in the ASN.1 standard itself), the prefix `Odr_` is sometimes seen. Either way, look for the actual definition in either `z-core.h` (for the types from the protocol), `odr.h` (for the primitive ASN.1 types). The Z39.50 ASN.1 library also provides functions (which are, in turn, defined using ODR primitives) for encoding and decoding data values. Their general form is

int **z_*xxx***(ODR o, Z_*xxx* **p, int optional, const char *name);

(note the lower-case "z" in the function name)

Note

If you are using the premade definitions of the Z39.50 ASN.1 module, and you are not adding a new protocol of your own, the only parts of ODR that you need to worry about are documented in Section 8.2.

When you have created a BER-encoded buffer, you can use the COMSTACK subsystem to transmit (or receive) data over the network. The COMSTACK module provides simple functions for establishing a connection (passively or actively, depending on the role of your application), and for exchanging BER-encoded PDUs over that connection. When you create a connection endpoint, you need to specify what transport to use (TCP/IP, SSL or UNIX sockets). For the remainder of the connection's lifetime, you don't have to worry about the underlying transport protocol at all - the COMSTACK will ensure that the correct mechanism is used.

We call the combined interfaces to ODR, Z39.50 ASN.1, and COMSTACK the service level API. It's the API that most closely models the Z39.50 service/protocol definition, and it provides unlimited access to all fields and facilities of the protocol definitions.

The reason that the YAZ service-level API is a conglomerate of the APIs from three different submodules is twofold. First, we wanted to allow the user a choice of different options for each major task. For instance, if you don't like the protocol API provided by ODR/Z39.50 ASN.1, you can use SNACC or BERUtils instead, and still have the benefits of the transparent transport approach of the COMSTACK module. Secondly, we realize that you may have to fit the toolkit into an existing event-processing structure, in a way that is incompatible with the COMSTACK interface or some other part of YAZ.

Chapter 2

Compilation and Installation

Introduction

The latest version of the software will generally be found at:

http://ftp.indexdata.com/pub/yaz/

We have tried our best to keep the software portable, and on many platforms, you should be able to compile everything with little or no changes.

The software is regularly tested on Debian GNU/Linux, CentOS, Ubuntu Linux, FreeBSD (i386), MAC OSX, Windows 10.

Some versions have be known to work on Windows XP, Solaris, HP/UX, DEC Unix, NetBSD, OpenBSD, IBM AIX, Data General DG/UX (with some CFLAGS tinkering), SGI/IRIX, DDE Supermax, Apple Macintosh (using the Codewarrior programming environment and the GUSI socket libraries), IBM AS/400 .

If you move the software to other platforms, we'd be grateful if you'd let us know about it. If you run into difficulties, we will try to help if we can, and if you solve the problems, we would be happy to include your fixes in the next release. So far, we have mostly avoided `#ifdefs` for individual platforms, and we'd like to keep it that way as far as it makes sense.

We maintain a mailing-list for the purpose of announcing new releases and bug-fixes, as well as general discussion. Subscribe by filling-in the form here. General questions and problems can be directed at mailto:yaz-help@indexdata.dk, or the address given at the top of this document.

UNIX

We provide Debian GNU/Linux (i386 and amd64), Ubuntu (i386 and amd64) and CentOS (amd64 only) packages for YAZ. You should be able to create packages for other CPUs by building them from the source package.

YAZ is also part of several packages repositories. Some of them are

- Solaris CSW: http://www.opencsw.org/packages/yaz/

- Solaris: http://unixpackages.com
- FreeBSD: http://www.freshports.org/net/yaz
- Debian: http://packages.debian.org/search?keywords=yaz
- Ubuntu: https://launchpad.net/ubuntu/+source/yaz
- NetBSD: http://ftp.netbsd.org/pub/pkgsrc/current/pkgsrc/net/yaz/README.html

Compiling from source on Unix

You can choose to compile YAZ from official tar releases from http://ftp.indexdata.com/pub/yaz/ or clone it via Github https://github.com/indexdata/yaz.git.

If you wish to use character set conversion facilities in YAZ or if you are compiling YAZ for use with Zebra, it is a good idea to ensure that the iconv library is installed. Some Unixes today already have it - if not, we suggest GNU libiconv.

YAZ 3.0.16 and later includes a wrapper for the ICU (International Components for Unicode). In order to use this, the developer version of the ICU library must be available. ICU support is recommended for applications such as Pazpar2 and Zebra.

The libxslt, libxml2 librararies are required if YAZ is to support SRU/Solr. These libraries are very portable and should compile out-of-the box on virtually all Unix platforms. It is available in binary forms for Linux and others.

The GNU tools Autoconf, Automake and Libtool are used to generate Makefiles and configure YAZ for the system. You do *not* need these tools unless you're using the Git version of YAZ.

The CQL parser for YAZ is built using GNU Bison. This tool is only needed if you're using the Git version of YAZ.

YAZ includes a tiny ASN.1 compiler. This compiler is written in Tcl. But as for Bison you do not need it unless you're using Git version of YAZ or you're using the compiler to build your own codecs for private ASN.1.

If you are checking out from Git, run:

```
./buildconf.sh
```

This will create the `configure` script and Makefiles.

The next step is always:

```
./configure
```

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

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

To customize YAZ, the configure script also accepts a set of options. The most important are:

--prefix=*prefix* Specifies installation prefix for YAZ. This is only needed if you run `make install` later to perform a "system" installation. The prefix is `/usr/local` if not specified.

--enable-tcpd The front end server will be built using Wietse's TCP wrapper library. It allows you to allow/deny clients depending on IP number. The TCP wrapper library is often used in GNU/Linux and BSD distributions. See hosts_access(5) and tcpd(8).

--enable-threads YAZ will be built using POSIX threads. Specifically, `_REENTRANT` will be defined during compilation.

--disable-shared The make process will not create shared libraries (also known as shared objects `.so`). By default, shared libraries are created - equivalent to `--enable-shared`.

--disable-shared The make process will not create static libraries (`.a`). By default, static libraries are created - equivalent to `--enable-static`.

--with-iconv[=*prefix*] Compile YAZ with iconv library in directory *prefix*. By default configure will search for iconv on the system. Use this option if it doesn't find iconv. Alternatively, `--without-iconv`, can be used to force YAZ not to use iconv.

--with-xslt[=*prefix*] Compile YAZ with libxslt in directory *prefix*. Use this option if you want XSLT and XML support. By default, configure will search for libxslt on the system. Use this option if libxslt is not found automatically. Alternatively, `--without-xslt`, can be used to force YAZ not to use libxslt.

--with-xml2[=*prefix*] Compile YAZ with libxml2 in directory *prefix*. Use this option if you want YAZ to use XML and support SRU/Solr. By default, configure will search for libxml2 on the system. Use this option if libxml2 is not found automatically. Alternatively, `--without-xml2`, can be used to force YAZ not to use libxml2.

Note that option `--with-xslt` also enables libxml2.

--with-gnutls[=*prefix*] YAZ will be linked with the GNU TLS libraries and an SSL COMSTACK will be provided. By default configure enables SSL support for YAZ if the GNU TLS development libraries are found on the system.

--with-icu[=*prefix*] YAZ will be linked the ICU library in the prefix if given. If prefix is not given, the libraries exposed by the script icu-config will be used if found.

--with-memcached YAZ will be linked with libMemcached to allow for result-set caching for ZOOM. The prefix can not be given. Note that 0.40 of libmemcached is required.

--with-redis YAZ will be linked with the hiredis C library to allow for result-set caching for ZOOM on a redis server. The prefix can not be given.

When configured, build the software by typing:

```
make
```

The following files are generated by the make process:

src/libyaz.la Main YAZ library. This is no ordinary library. It's a Libtool archive. By default, YAZ creates a static library in `lib/.libs/libyaz.a`.

src/libyaz_server.la Generic Frontend server. This is an add-on for libyaz.la. Code in this library uses POSIX threads functions - if POSIX threads are available on the platform.

src/libyaz_icu.la Functions that wrap the ICU library.

ztest/yaz-ztest Test Z39.50 server.

client/yaz-client Z39.50 client for testing the protocol. See chapter YAZ client for more information.

util/yaz-config A Bourne-shell script, generated by configure, that specifies how external applications should compile - and link with YAZ.

util/yaz-asncomp The ASN.1 compiler for YAZ. Requires the Tcl Shell, tclsh, in `PATH` to operate.

util/yaz-iconv This program converts data in one character set to another. This command exercises the YAZ character set conversion API.

util/yaz-marcdump This program parses ISO2709 encoded MARC records and prints them in line-format or XML.

util/yaz-icu This program exposes the ICU wrapper library if that is enabled for YAZ. Only if ICU is available this program is useful.

util/yaz-url This program is a simple HTTP page fetcher ala wget or curl.

zoom/zoomsh A simple shell implemented on top of the ZOOM functions. The shell is a command line application that allows you to enter simple commands to perform ZOOM operations.

zoom/zoomtst1, zoom/zoomtst2, .. Several small applications that demonstrate the ZOOM API.

If you wish to install YAZ in system directories `/usr/local/bin`, `/usr/local/lib` .. etc, you can type:

```
make install
```

You probably need to have root access in order to perform this. You must specify the `--prefix` option for configure if you wish to install YAZ in other directories than the default `/usr/local/`.

If you wish to perform an un-installation of YAZ, use:

```
make uninstall
```

This will only work if you haven't reconfigured YAZ (and therefore changed installation prefix). Note that uninstall will not remove directories created by make install, e.g. `/usr/local/include/yaz`.

How to make apps using YAZ on UNIX

This section describes how to compile - and link your own applications using the YAZ toolkit. If you're used to Makefiles this shouldn't be hard. As for other libraries you have used before, you need to set a proper include path for your C/C++ compiler and specify the location of YAZ libraries. You can do it by hand, but generally we suggest you use the `yaz-config` that is generated by `configure`. This is especially important if you're using the threaded version of YAZ which require you to pass more options to your linker/compiler.

The `yaz-config` script accepts command line options that makes the `yaz-config` script print options that you should use in your make process. The most important ones are: `--cflags`, `--libs` which prints C compiler flags, and linker flags respectively.

A small and complete `Makefile` for a C application consisting of one source file, `myprog.c`, may look like this:

```
YAZCONFIG=/usr/local/bin/yaz-config
CFLAGS=`$(YAZCONFIG) --cflags`
LIBS=`$(YAZCONFIG) --libs`
myprog: myprog.o
   $(CC) $(CFLAGS) -o myprog myprog.o $(LIBS)
```

The CFLAGS variable consists of a C compiler directive that will set the include path to the *parent* directory of `yaz`. That is, if YAZ header files were installed in `/usr/local/include/yaz`, then include path is set to `/usr/local/include`. Therefore, in your applications you should use

```
#include <yaz/proto.h>
```

and *not*

```
#include <proto.h>
```

For Libtool users, the `yaz-config` script provides a different variant of option `--libs`, called `--lal ibs` that returns the name of the Libtool archive(s) for YAZ rather than the ordinary ones.

For applications using the threaded version of YAZ, specify `threads` after the other options. When `threads` is given, more flags and linker flags will be printed by `yaz-config`. If our previous example was using threads, you'd have to modify the lines that set `CFLAGS` and `LIBS` as follows:

```
CFLAGS=`$(YAZCONFIG) --cflags threads`
LIBS=`$(YAZCONFIG) --libs threads`
```

There is no need specify POSIX thread libraries in your Makefile. The `LIBS` variable includes that as well.

Windows

The easiest way to install YAZ on Windows is by downloading an installer from Index Data's Windows support area . The installer comes with source too - in case you wish to compile YAZ with different compiler options, etc.

Compiling from Source on Windows

YAZ is shipped with "makefiles" for the NMAKE tool that comes with Microsoft Visual Studio. It has been tested with Microsoft Visual Studio 2015.

Start a command prompt and switch the sub directory `WIN` where the file `makefile` is located. Customize the installation by editing the `makefile` file (for example by using notepad). The following summarizes the most important settings in that file:

`DEBUG` If set to 1, the software is compiled with debugging libraries (code generation is multi-threaded debug DLL). If set to 0, the software is compiled with release libraries (code generation is multi-threaded DLL).

`HAVE_TCL`, `TCL` If `HAVE_TCL` is set to 1, nmake will use the ASN.1 compiler (Tcl based). You must set `TCL` to the full path of the Tcl interpreter. A Windows version of Tcl is part of Git for Windows.

If you do not have Tcl installed, set `HAVE_TCL` to 0.

`HAVE_BISON`, `BISON` If GNU Bison is present, you might set `HAVE_BISON` to 1 and specify the Bison executable in `BISON`. Bison is only required if you use the Git version of YAZ or if you modify the grammar for CQL (`cql.y`).

A Windows version of GNU Bison can be fetched from here: Index Data's Windows support area .

`HAVE_ICONV`, `ICONV_DIR` If `HAVE_ICONV` is set to 1, YAZ is compiled with iconv support. In this configuration, set `ICONV_DIR` to the iconv source directory.

`HAVE_LIBXML2`, `LIBXML2_DIR` If `HAVE_LIBXML2` is set to 1, YAZ is compiled with SRU support. In this configuration, set `LIBXML2_DIR` to the libxml2 source directory.

You can get pre-compiled Libxml2+Libxslt DLLs and headers from here. Should you with to compile those libraries yourself, refer to to Section 2.3.3

`HAVE_LIBXSLT`, `LIBXSLT_DIR` If `HAVE_LIBXSLT` is set to 1, YAZ is compiled with XSLT support. In this configuration, set `LIBXSLT_DIR` to the libxslt source directory.

> **Note**
> libxslt depends on libxml2.

`HAVE_ICU`, `ICU_DIR` If `HAVE_ICU` is set to 1, YAZ is compiled with ICU support. In this configuration, set `ICU_DIR` to the ICU source directory.

Pre-compiled ICU libraries for various versions of Visual Studio can be found here or from Index Data's Windows support site.

When satisfied with the settings in the makefile, type

```
nmake
```

Note
If the `nmake` command is not found on your system you probably haven't defined the environment variables required to use that tool. To fix that, find and run the batch file `vcvarsall.bat`. You need to run it from within the command prompt or set the environment variables "globally"; otherwise it doesn't work.

If you wish to recompile YAZ - for example if you modify settings in the `makefile` you can delete object files, etc by running.

```
nmake clean
```

The following files are generated upon successful compilation:

`bin/yaz5.dll` / `bin/yaz5d.dll` YAZ Release/Debug DLL.

`lib/yaz5.lib` / `lib/yaz5d.lib` Import library for `yaz5.dll` / `yaz5d.dll`.

`bin/yaz_cond5.dll` / `bin/yaz_cond5d.dll` Release/Debug DLL for condition variable utilities (condvar.c).

`lib/yaz_cond5.lib` / `lib/yaz_cond5d.lib` Import library for `yaz_cond5.dll` / `yaz_cond5d.dll`.

`bin/yaz_icu5.dll` / `bin/yaz_icu5d.dll` Release/Debug DLL for the ICU wrapper utility. Only build if HAVE_ICU is 1.

`lib/yaz_icu5.lib` / `lib/yaz_icu5d.lib` Import library for `yaz_icu5.dll` / `yaz_icu5d.dll`.

`bin/yaz-ztest.exe` Z39.50 multi-threaded test/example server. It's a WIN32 console application.

`bin/yaz-client.exe` YAZ Z39.50 client application. It's a WIN32 console application. See chapter YAZ client for more information.

`bin/yaz-icu.exe` This program exposes the ICU wrapper library if that is enabled for YAZ. Only if ICU is available this program is built.

`bin/zoomsh.exe` Simple console application implemented on top of the ZOOM functions. The application is a command line shell that allows you to enter simple commands to perform ZOOM operations.

`bin/zoomtst1.exe`, `bin/zoomtst2.exe`, .. Several small applications that demonstrate the ZOOM API.

How to make apps using YAZ on Windows

This section will go though the process of linking your Windows applications with YAZ.

Some people are confused by the fact that we use the nmake tool to build YAZ. They think they have to do that too - in order to make their Windows applications work with YAZ. The good news is that you don't have to. You can use the integrated environment of Visual Studio if desired for your own application.

When setting up a project or Makefile you have to set the following:

include path Set it to the `include` directory of YAZ.

import library `yaz5.lib` You must link with this library. It's located in the sub directory `lib` of YAZ. If you want to link with the debug version of YAZ, you must link against `yaz5d.lib` instead.

dynamic link library `yaz5.dll` This DLL must be in your execution path when you invoke your application. Specifically, you should distribute this DLL with your application.

Compiling Libxml2 and Libxslt on windows

Download libxml2 and Libxslt source and unpack it. In the example below we install Libxml2 2.9.2 and Libxslt 1.1.28 for 32-bit, so we use the destination directories libxml2.2.9.2.win32 and libxslt-1.1.28.win32 to reflect both version and architecture.

```
cd win32
cscript configure.js prefix=c:\libxml2-2.9.2.win32 iconv=no
nmake
nmake install
```

Note
There's an error in `configure.js` for Libxml2 2.9.2. Line 17 should be assigned to `configure.ac` rather than `configure.in`.

For Libxslt it is similar. We must ensure that compilation of Libxslt links against the already installed libxml2.

```
cd win32
cscript configure.js prefix=c:\libxslt-1.1.28.win32 iconv=no \
    lib=c:\libxml2-2.9.2.win32\lib \
include=c:\libxml2-2.9.2.win32\include\libxml2
nmake
nmake install
```

Chapter 3

ZOOM

ZOOM is an acronym for 'Z39.50 Object-Orientation Model' and is an initiative started by Mike Taylor (Mike is from the UK, which explains the peculiar name of the model). The goal of ZOOM is to provide a common Z39.50 client API not bound to a particular programming language or toolkit.

From YAZ version 2.1.12, SRU is supported. You can make SRU ZOOM connections by specifying scheme `http://` for the hostname for a connection. The dialect of SRU used is specified by the value of the connection's `sru` option, which may be SRU over HTTP GET (`get`), SRU over HTTP POST (`post`), (SRU over SOAP) (`soap`) or `solr` (Solr Web Service). Using the facility for embedding options in target strings, a connection can be forced to use SRU rather the SRW (the default) by prefixing the target string with `sru=get,`, like this: `sru=get,http://sru.miketaylor.org.uk:80/sru.pl`

Solr protocol support was added to YAZ in version 4.1.0, as a dialect of a SRU protocol, since both are HTTP based protocols.

The lack of a simple Z39.50 client API for YAZ has become more and more apparent over time. So when the first ZOOM specification became available, an implementation for YAZ was quickly developed. For the first time, it is now as easy (or easier!) to develop clients as it is to develop servers with YAZ. This chapter describes the ZOOM C binding. Before going further, please reconsider whether C is the right programming language for the job. There are other language bindings available for YAZ, and still more are in active development. See the ZOOM web-site for more information.

In order to fully understand this chapter you should read and try the example programs `zoomtst1.c`, `zoomtst2.c`, .. in the `zoom` directory.

The C language misses features found in object oriented languages such as C++, Java, etc. For example, you'll have to manually, destroy all objects you create, even though you may think of them as temporary. Most objects have a `_create` - and a `_destroy` variant. All objects are in fact pointers to internal stuff, but you don't see that because of typedefs. All destroy methods should gracefully ignore a `NULL` pointer.

In each of the sections below you'll find a sub section called protocol behavior, that describes how the API maps to the Z39.50 protocol.

Connections

The Connection object is a session with a target.

```
#include <yaz/zoom.h>

ZOOM_connection ZOOM_connection_new(const char *host, int portnum);

ZOOM_connection ZOOM_connection_create(ZOOM_options options);

void ZOOM_connection_connect(ZOOM_connection c, const char *host,
                             int portnum);
void ZOOM_connection_destroy(ZOOM_connection c);
```

Connection objects are created with either function `ZOOM_connection_new` or `ZOOM_connection_create`. The former creates and automatically attempts to establish a network connection with the target. The latter doesn't establish a connection immediately, thus allowing you to specify options before establishing network connection using the function `ZOOM_connection_connect`. If the port number, `portnum`, is zero, the `host` is consulted for a port specification. If no port is given, 210 is used. A colon denotes the beginning of a port number in the host string. If the host string includes a slash, the following part specifies a database for the connection.

You can prefix the host with a scheme followed by colon. The default scheme is `tcp` (Z39.50 protocol). The scheme `http` selects SRU/get over HTTP by default, but can overridden to use SRU/post, SRW, and the Solr protocol.

You can prefix the scheme-qualified host-string with one or more comma-separated *`key=value`* sequences, each of which represents an option to be set into the connection structure *before* the protocol-level connection is forged and the initialization handshake takes place. This facility can be used to provide authentication credentials, as in host-strings such as: `user=admin,password=halfAm4n,tcp:localhost:8017/db`

Connection objects should be destroyed using the function `ZOOM_connection_destroy`.

```
void ZOOM_connection_option_set(ZOOM_connection c,
                                const char *key, const char *val);

void ZOOM_connection_option_setl(ZOOM_connection c,
                                 const char *key,
                                 const char *val, int len);

const char *ZOOM_connection_option_get(ZOOM_connection c,
                                       const char *key);
const char *ZOOM_connection_option_getl(ZOOM_connection c,
                                        const char *key,
                                        int *lenp);
```

The functions `ZOOM_connection_option_set` and `ZOOM_connection_option_setl` allows you to set an option given by *`key`* to the value *`value`* for the connection. For `ZOOM_connection_option_set`, the value is assumed to be a 0-terminated string. Function `ZOOM_connection_option_setl` specifies a value of a certain size (len).

Functions ZOOM_connection_option_get and ZOOM_connection_option_getl returns the value for an option given by *key*.

If either option lang or charset is set, then Character Set and Language Negotiation is in effect.

```
int ZOOM_connection_error(ZOOM_connection c, const char **cp,
                          const char **addinfo);
int ZOOM_connection_error_x(ZOOM_connection c, const char **cp,
                            const char **addinfo, const char **dset);
```

Function ZOOM_connection_error checks for errors for the last operation(s) performed. The function returns zero if no errors occurred; non-zero otherwise indicating the error. Pointers *cp* and *addinfo* holds messages for the error and additional-info if passed as non-NULL. Function ZOOM_connection_error_x is an extended version of ZOOM_connection_error that is capable of returning name of diagnostic set in *dset*.

Z39.50 Protocol behavior

The calls ZOOM_connection_new and ZOOM_connection_connect establishes a TCP/IP connection and sends an Initialize Request to the target if possible. In addition, the calls wait for an Initialize Response from the target and the result is inspected (OK or rejected).

If proxy is set then the client will establish a TCP/IP connection with the peer as specified by the proxy host and the hostname as part of the connect calls will be set as part of the Initialize Request. The proxy server will then "forward" the PDUs transparently to the target behind the proxy.

For the authentication parameters, if option user is set and both options group and pass are unset, then Open style authentication is used (Version 2/3) in which case the username is usually followed by a slash, then by a password. If either group or pass is set then idPass authentication (Version 3 only) is used. If none of the options are set, no authentication parameters are set as part of the Initialize Request (obviously).

When option async is 1, it really means that all network operations are postponed (and queued) until the function ZOOM_event is invoked. When doing so it doesn't make sense to check for errors after ZOOM_connection_new is called since that operation "connecting - and init" is still incomplete and the API cannot tell the outcome (yet).

SRU/Solr Protocol behavior

The HTTP based protocols (SRU, SRW, Solr) do not feature an Inititialize Request, so the connection phase merely establishes a TCP/IP connection with the HTTP server.

Most of the ZOOM connection options do not affect SRU/Solr and they are ignored. However, future versions of YAZ might honor implementationName and put that as part of User-Agent header for HTTP requests.

The charset is used in the Content-Type header of HTTP requests.

Setting authentcationMode specifies how authentication parameters are encoded for HTTP. The default is "basic" where user and password are encoded by using HTTP basic authentication.

Option	Description	Default
implementationName	Name of your client	none
user	Authentication user name	none
group	Authentication group name	none
password	Authentication password.	none
authenticationMode	How authentication is encoded.	basic
host	Target host. This setting is "read-only". It's automatically set internally when connecting to a target.	none
proxy	Proxy host. If set, the logical host is encoded in the otherInfo area of the Z39.50 Init PDU with OID 1.2.840.10003.10.1000.81.1.	none
clientIP	Client IP. If set, is encoded in the otherInfo area of a Z39.50 PDU with OID 1.2.840.10003.10.1000.81.3. Holds the original IP addresses of a client. Is used if ZOOM is used in a gateway of some sort.	none
async	If true (1) the connection operates in asynchronous operation which means that all calls are non-blocking except ZOOM_event.	0
maximumRecordSize	Maximum size of single record.	1 MB
preferredMessageSize	Maximum size of multiple records.	1 MB
lang	Language for negotiation.	none
charset	Character set for negotiation.	none
serverImplementationId	Implementation ID of server. (The old targetImplementationId option is also supported for the benefit of old applications.)	none
targetImplementationName	Implementation Name of server. (The old targetImplementationName option is also supported for the benefit of old applications.)	none
serverImplementationVersion	Implementation Version of server. (The old targetImplementationVersion option is also supported for the benefit of old applications.)	none
databaseName	One or more database names separated by character plus (+), which is to be used by subsequent search requests on this Connection.	Default
piggyback	True (1) if piggyback should be used in searches; false (0) if not.	1
smallSetUpperBound	If hits is less than or equal to this value, then target will return all records using small element set name	0
largeSetLowerBound	If hits is greater than this value, the target will return no records.	1
mediumSetPresentNumber	This value represents the number of records to be returned as part of a search when hits is less than or equal to large set lower bound and if hits is greater than small set upper bound.	0
smallSetElementSetName	The element set name to be used for small result sets.	none
mediumSetElementSetName	The element set name to be used for medium-sized result sets.	none
init_opt_search, init_opt_present,	After a successful Init, these options may be interrogated to discover whether the server	none

If authentcationMode is "url", then user and password are encoded in the URL by parameters x-username and x-password as given by the SRU standard.

Queries

Query objects represents queries.

```
ZOOM_query ZOOM_query_create(void);

void ZOOM_query_destroy(ZOOM_query q);

int ZOOM_query_prefix(ZOOM_query q, const char *str);

int ZOOM_query_cql(ZOOM_query s, const char *str);

int ZOOM_query_sortby(ZOOM_query q, const char *criteria);

int ZOOM_query_sortby2(ZOOM_query q, const char *strategy,
                       const char *criteria);
```

Create query objects using ZOOM_query_create and destroy them by calling ZOOM_query_destroy. RPN-queries can be specified in PQF notation by using the function ZOOM_query_prefix. The ZOOM_query_cql specifies a CQL query to be sent to the server/target. More query types will be added in future versions of YAZ, such as CCL to RPN-mapping, native CCL query, etc. In addition to a search, a sort criteria may be set. Function ZOOM_query_sortby enables Z39.50 sorting and it takes sort criteria using the same string notation as yaz-client's sort command.

ZOOM_query_sortby2 is similar to ZOOM_query_sortby but allows a strategy for sorting. The reason for the strategy parameter is that some protocols offer multiple ways of performing sorting. For example, Z39.50 has the standard sort, which is performed after search on an existing result set. It's also possible to use CQL in Z39.50 as the query type and use CQL's SORTBY keyword. Finally, Index Data's Zebra server also allows sorting to be specified as part of RPN (Type 7).

Name	Description
z39.50	Z39.50 resultset sort
type7	Sorting embedded in RPN(Type-7)
cql	CQL SORTBY
sru11	SRU sortKeys parameter
solr	Solr sort
embed	type7 for Z39.50, cql for SRU, solr for Solr protocol

Table 3.2: ZOOM sort strategy

Result sets

The result set object is a container for records returned from a target.

```
ZOOM_resultset ZOOM_connection_search(ZOOM_connection, ZOOM_query q);

ZOOM_resultset ZOOM_connection_search_pqf(ZOOM_connection c,
                                          const char *q);
void ZOOM_resultset_destroy(ZOOM_resultset r);
```

Function `ZOOM_connection_search` creates a result set, given a connection and query. Destroy a result set by calling `ZOOM_resultset_destroy`. Simple clients using PQF only, may use the function `ZOOM_connection_search_pqf` in which case creating query objects is not necessary.

```
void ZOOM_resultset_option_set(ZOOM_resultset r,
                               const char *key, const char *val);

const char *ZOOM_resultset_option_get(ZOOM_resultset r, const char *key

size_t ZOOM_resultset_size(ZOOM_resultset r);
```

Functions `ZOOM_resultset_options_set` and `ZOOM_resultset_get` sets and gets an option for a result set similar to `ZOOM_connection_option_get` and `ZOOM_connection_option_set`.

The number of hits, also called result-count, is returned by function `ZOOM_resultset_size`.

For servers that support Search Info report, the following options may be read using `ZOOM_resultset_get`. This detailed information is read after a successful search has completed.

This information is a list of of items, where each item is information about a term or subquery. All items in the list are prefixed by `SearchResult.`*`no`* where no presents the item number (0=first, 1=second). Read `searchresult.size` to determine the number of items.

Z39.50 Result-set Sort

```
void ZOOM_resultset_sort(ZOOM_resultset r,
                         const char *sort_type, const char *sort_spec);

int ZOOM_resultset_sort1(ZOOM_resultset r,
                         const char *sort_type, const char *sort_spec);
```

`ZOOM_resultset_sort` and `ZOOM_resultset_sort1` both sort an existing result-set. The sort_type parameter is not used. Set it to "yaz". The sort_spec is same notation as ZOOM_query_sortby and identical to that offered by yaz-client's sort command.

These functions only work for Z39.50. Use the more generic utility ZOOM_query_sortby2 for other protocols (and even Z39.50).

Option	Description	Default
start	Offset of first record to be retrieved from target. First record has offset 0 unlike the protocol specifications where first record has position 1. This option affects ZOOM_resultset_search and ZOOM_resultset_search_pqf and must be set before any of these functions are invoked. If a range of records must be fetched manually after search, function ZOOM_resultset_records should be used.	0
count	Number of records to be retrieved. This option affects ZOOM_resultset_search and ZOOM_resultset_search_pqf and must be set before any of these functions are invoked.	0
presentChunk	The number of records to be requested from the server in each chunk (present request). The value 0 means to request all the records in a single chunk. (The old `step` option is also supported for the benefit of old applications.)	0
elementSetName	Element-Set name of records. Most targets should honor element set name `B` and `F` for brief and full respectively.	none
preferredRecordSyntax	Preferred Syntax, such as `USMARC`, `SUTRS`, etc.	none
schema	Schema for retrieval, such as `Gils-schema`, `Geo-schema`, etc.	none
setname	Name of Result Set (Result Set ID). If this option isn't set, the ZOOM module will automatically allocate a result set name.	default
rpnCharset	Character set for RPN terms. If this is set, ZOOM C will assume that the ZOOM application is running UTF-8. Terms in RPN queries are then converted to the rpnCharset. If this is unset, ZOOM C will not assume any encoding of RPN terms and no conversion is performed.	none

Table 3.3: ZOOM Result set Options

Option	Description
searchresult.size	number of search result entries. This option is non-existent if no entries are returned by the server.
searchresult.*no*.id	sub query ID
searchresult.*no*.count	result count for item (number of hits)
searchresult.*no*.subquery.term	subquery term
searchresult.*no*.interpretation.term	interpretation term
searchresult.*no*.recommendation.term	recommendation term

Table 3.4: Search Info Report Options

Z39.50 Protocol behavior

The creation of a result set involves at least a SearchRequest - SearchResponse protocol handshake. Following that, if a sort criteria was specified as part of the query, a SortRequest - SortResponse handshake takes place. Note that it is necessary to perform sorting before any retrieval takes place, so no records will be returned from the target as part of the SearchResponse because these would be unsorted. Hence, piggyback is disabled when sort criteria are set. Following Search - and a possible sort - Retrieval takes place - as one or more Present Requests/Response pairs being transferred.

The API allows for two different modes for retrieval. A high level mode which is somewhat more powerful and a low level one. The low level is enabled when searching on a Connection object for which the settings `smallSetUpperBound`, `mediumSetPresentNumber` and `largeSetLowerBound` are set. The low level mode thus allows you to precisely set how records are returned as part of a search response as offered by the Z39.50 protocol. Since the client may be retrieving records as part of the search response, this mode doesn't work well if sorting is used.

The high-level mode allows you to fetch a range of records from the result set with a given start offset. When you use this mode the client will automatically use piggyback if that is possible with the target, and perform one or more present requests as needed. Even if the target returns fewer records as part of a present response because of a record size limit, etc. the client will repeat sending present requests. As an example, if option `start` is 0 (default) and `count` is 4, and `piggyback` is 1 (default) and no sorting criteria is specified, then the client will attempt to retrieve the 4 records as part the search response (using piggyback). On the other hand, if either `start` is positive or if a sorting criteria is set, or if `piggyback` is 0, then the client will not perform piggyback but send Present Requests instead.

If either of the options `mediumSetElementSetName` and `smallSetElementSetName` are unset, the value of option `elementSetName` is used for piggyback searches. This means that for the high-level mode you only have to specify one elementSetName option rather than three.

SRU Protocol behavior

Current version of YAZ does not take advantage of a result set id returned by the SRU server. Future versions might do, however. Since the ZOOM driver does not save result set IDs, any present (retrieval) is transformed to a SRU SearchRetrieveRequest with same query but, possibly, different offsets.

Option `schema` specifies SRU schema for retrieval. However, options `elementSetName` and `preferredRecordSyntax` are ignored.

Options `start` and `count` are supported by SRU. The remaining options `piggyback`, `smallSetUpperBound`, `largeSetLowerBound`, `mediumSetPresentNumber`, `mediumSetElementSetName`, `smallSetElementSetName` are unsupported.

SRU supports CQL queries, *not* PQF. If PQF is used, however, the PQF query is transferred anyway using non-standard element `pQuery` in SRU SearchRetrieveRequest.

Solr queries need to be done in Solr query format.

Unfortunately, SRU and Solr do not define a database setting. Hence, `databaseName` is unsupported and ignored. However, the path part in host parameter for functions `ZOOM_connecton_new` and `ZOOM_connection_connect` acts as a database (at least for the YAZ SRU server).

Records

A record object is a retrieval record on the client side - created from result sets.

```
void ZOOM_resultset_records(ZOOM_resultset r,
                            ZOOM_record *recs,
                            size_t start, size_t count);
ZOOM_record ZOOM_resultset_record(ZOOM_resultset s, size_t pos);

const char *ZOOM_record_get(ZOOM_record rec, const char *type,
                            size_t *len);

int ZOOM_record_error(ZOOM_record rec, const char **msg,
                      const char **addinfo, const char **diagset);

ZOOM_record ZOOM_record_clone(ZOOM_record rec);

void ZOOM_record_destroy(ZOOM_record rec);
```

References to temporary records are returned by functions `ZOOM_resultset_records` or `ZOOM_resultset_record`.

If a persistent reference to a record is desired `ZOOM_record_clone` should be used. It returns a record reference that should be destroyed by a call to `ZOOM_record_destroy`.

A single record is returned by function `ZOOM_resultset_record` that takes a position as argument. First record has position zero. If no record could be obtained `NULL` is returned.

Error information for a record can be checked with `ZOOM_record_error` which returns non-zero (error code) if record is in error, called *Surrogate Diagnostics* in Z39.50.

Function `ZOOM_resultset_records` retrieves a number of records from a result set. Parameter `start` and `count` specifies the range of records to be returned. Upon completion, the array `recs[0], ..recs[count-1]` holds record objects for the records. The array of records `recs` should be allocated prior the call `ZOOM_resultset_records`. Note that for those records that couldn't be retrieved from the target, `recs[ ..]` is set to `NULL`.

In order to extract information about a single record, `ZOOM_record_get` is provided. The function returns a pointer to certain record information. The nature (type) of the pointer depends on the parameter, *`type`*.

The *`type`* is a string of the format:

`format`[;charset=*`from`*[/*`opacfrom`*][,*`to`*]][;format=*`v`*][;base64=*`xpath`*]

If `charset` is given, then *`from`* specifies the character set of the record in its original form (as returned by the server), *`to`* specifies the output (returned) character set encoding. If *`to`* is omitted, then UTF-8 is assumed. If charset is not given, then no character set conversion takes place. OPAC records may be returned in a different set from the bibliographic MARC record. If this is this the case, *`opacfrom`* should be set to the character set of the OPAC record part.

The `format` is generic but can only be used to specify XML indentation when the value *v* is 1 (`format=1`).

The `base64` allows a full record to be extracted from base64-encoded string in an XML document.

Note
Specifying the OPAC record character set requires YAZ 4.1.5 or later.
Specifying the base64 parameter requires YAZ 4.2.35 or later.

The format argument controls whether record data should be XML pretty-printed (post process operation). It is enabled only if format value *v* is `1` and the record content is XML well-formed.

In addition, for certain types, the length `len` passed will be set to the size in bytes of the returned information.

The following are the supported values for *form*.

database The Database of the record is returned as a C null-terminated string. Return type `const char *`.

syntax The transfer syntax of the record is returned as a C null-terminated string containing the symbolic name of the record syntax, e.g. `Usmarc`. Return type is `const char *`.

schema The schema of the record is returned as a C null-terminated string. Return type is `const char *`.

render The record is returned in a display friendly format. Upon completion, buffer is returned (type `const char *`) and length is stored in `*len`.

raw The record is returned in the internal YAZ specific format. For GRS-1, Explain, and others, the raw data is returned as type `Z_External *` which is just the type for the member `retrievalRecord` in type `NamePlusRecord`. For SUTRS and octet aligned record (including all MARCs) the octet buffer is returned and the length of the buffer.

xml The record is returned in XML if possible. SRU, Solr and Z39.50 records with transfer syntax XML are returned verbatim. MARC records are returned in MARCXML (converted from ISO2709 to MARCXML by YAZ). OPAC records are also converted to XML and the bibliographic record is converted to MARCXML (when possible). GRS-1 records are not supported for this form. Upon completion, the XML buffer is returned (type `const char *`) and length is stored in `*len`.

opac OPAC information for record is returned in XML if an OPAC record is present at the position given. If no OPAC record is present, a NULL pointer is returned.

txml The record is returned in TurboMARC if possible. SRU and Z39.50 records with transfer syntax XML are returned verbatim. MARC records are returned in TurboMARC (converted from ISO2709 to TurboMARC by YAZ). Upon completion, the XML buffer is returned (type `const char *`) and length is stored in `*len`.

json Like xml, but MARC records are converted to MARC-in-JSON.

Most MARC21 records uses the MARC-8 character set encoding. An application that wishes to display in Latin-1 would use

```
render; charset=marc8,iso-8859-1
```

Z39.50 Protocol behavior

The functions `ZOOM_resultset_record` and `ZOOM_resultset_records` inspects the client-side record cache. Records not found in cache are fetched using Present. The functions may block (and perform network I/O) - even though option `async` is 1, because they return records objects. (And there's no way to return records objects without retrieving them!)

There is a trick, however, in the usage of function `ZOOM_resultset_records` that allows for delayed retrieval (and makes it non-blocking). By using a null pointer for *`recs`* you're indicating you're not interested in getting records objects *now*.

SRU/Solr Protocol behavior

The ZOOM driver for SRU/Solr treats records returned by a SRU/Solr server as if they where Z39.50 records with transfer syntax XML and no element set name or database name.

ZOOM Facets

Facets are only supported for a few Z39.50 targets. It is a relatively new non-standard Z39.50 extension (see facets.asn in the YAZ source). However, facets are usually supported for Solr and SRU 2.0 targets.

Facets may be specified by the facets option before a search is sent. See Section 7.8 for the notation. For inspection of the returned facets, the following functions are available:

```
ZOOM_facet_field *ZOOM_resultset_facets(ZOOM_resultset r);

size_t ZOOM_resultset_facets_size(ZOOM_resultset r);

ZOOM_facet_field ZOOM_resultset_get_facet_field(ZOOM_resultset r,
                                                const char *facet_name)

ZOOM_facet_field ZOOM_resultset_get_facet_field_by_index(ZOOM_resultset
                                                         int pos);

const char *ZOOM_facet_field_name(ZOOM_facet_field facet_field);

size_t ZOOM_facet_field_term_count(ZOOM_facet_field facet_field);

const char *ZOOM_facet_field_get_term(ZOOM_facet_field facet_field,
                                      size_t idx, int *freq);
```

References to temporary structures are returned by all functions. They are only valid as long the Result set is valid.

All facet fields may be returned by a call to `ZOOM_resultset_facets`. The length of the array is given by `ZOOM_resultset_facets_size`. The array is zero-based and the last entry will be at `ZOOM_resultset_facets_size(result_set)-1`.

Facet fields can also be fetched via its name using `ZOOM_resultset_get_facet_field`. Or by its index (starting from 0) by a call to `ZOOM_resultset_get_facet_field_by_index`. Both of these functions return NULL if name is not found or index is out of bounds.

Function `ZOOM_facet_field_name` gets the request facet name from a returned facet field.

Function `ZOOM_facet_field_get_term` returns the idx'th term and term count for a facet field. Idx must between 0 and `ZOOM_facet_field_term_count-1`, otherwise the returned reference will be `NULL`. On a valid idx, the value of the freq reference will be the term count. The `freq` parameter must be valid pointer to integer.

Scan

This section describes an interface for Scan. Scan is not an official part of the ZOOM model yet. The result of a scan operation is the `ZOOM_scanset` which is a set of terms returned by a target.

The Scan interface is supported for both Z39.50, SRU and Solr.

```
ZOOM_scanset ZOOM_connection_scan(ZOOM_connection c,
                                  const char *startpqf);

ZOOM_scanset ZOOM_connection_scan1(ZOOM_connection c,
                                   ZOOM_query q);

size_t ZOOM_scanset_size(ZOOM_scanset scan);

const char *ZOOM_scanset_term(ZOOM_scanset scan, size_t pos,
                              size_t *occ, size_t *len);

const char *ZOOM_scanset_display_term(ZOOM_scanset scan, size_t pos,
                                      size_t *occ, size_t *len);

void ZOOM_scanset_destroy(ZOOM_scanset scan);

const char *ZOOM_scanset_option_get(ZOOM_scanset scan,
                                    const char *key);

void ZOOM_scanset_option_set(ZOOM_scanset scan, const char *key,
                             const char *val);
```

The scan set is created by function `ZOOM_connection_scan` which performs a scan operation on the connection using the specified *`startpqf`*. If the operation was successful, the size of the scan set can be retrieved by a call to `ZOOM_scanset_size`. Like result sets, the items are numbered 0..size-1. To obtain information about a particular scan term, call function `ZOOM_scanset_term`. This function takes a scan set offset `pos` and returns a pointer to a *raw term* or `NULL` if non-present. If present, the `occ` and `len` are set to the number of occurrences and the length of the actual term respectively. `ZOOM_scanset_display_term` is similar to `ZOOM_scanset_term` except that it returns the *display term* rather than the raw term. In a few cases, the term is different from display term. Always use the display term for display and the raw term for subsequent scan operations (to get more terms, next scan result, etc).

A scan set may be freed by a call to function `ZOOM_scanset_destroy`. Functions `ZOOM_scanset_option_get` and `ZOOM_scanset_option_set` retrieves and sets an option respectively.

The *`startpqf`* is a subset of PQF, namely the Attributes+Term part. Multiple `@attr` can be used. For example to scan in title (complete) phrases:

```
@attr 1=4 @attr 6=2 "science o"
```

The `ZOOM_connecton_scan1` is a newer and more generic alternative to `ZOOM_connection_scan` which allows to use both CQL and PQF for Scan.

Option	Description	Default
number	Number of Scan Terms requested in next scan. After scan it holds the actual number of terms returned.	20
position	Preferred Position of term in response in next scan; actual position after completion of scan.	1
stepSize	Step Size	0
scanStatus	An integer indicating the Scan Status of last scan.	0
rpnCharset	Character set for RPN terms. If this is set, ZOOM C will assume that the ZOOM application is running UTF-8. Terms in RPN queries are then converted to the rpnCharset. If this is unset, ZOOM C will not assume any encoding of RPN terms and no conversion is performed.	none

Table 3.5: ZOOM Scan Set Options

Extended Services

ZOOM offers an interface to a subset of the Z39.50 extended services as well as a few privately defined ones:

- Z39.50 Item Order (ILL). See Section 3.7.1.
- Record Update. This allows a client to insert, modify or delete records. See Section 3.7.2.

- Database Create. This a non-standard feature. Allows a client to create a database. See Section 3.7.3.
- Database Drop. This a non-standard feature. Allows a client to delete/drop a database. See Section 3.7.4.
- Commit operation. This a non-standard feature. Allows a client to commit operations. See Section 3.7.5.

To create an extended service operation, a `ZOOM_package` must be created. The operation is a five step operation. The package is created, package is configured by means of options, the package is sent, result is inspected (by means of options), the package is destroyed.

```
ZOOM_package ZOOM_connection_package(ZOOM_connection c,
                                     ZOOM_options options);

const char *ZOOM_package_option_get(ZOOM_package p,
                                    const char *key);
void ZOOM_package_option_set(ZOOM_package p, const char *key,
                             const char *val);
void ZOOM_package_send(ZOOM_package p, const char *type);

void ZOOM_package_destroy(ZOOM_package p);
```

The `ZOOM_connection_package` creates a package for the connection given using the options specified.

Functions `ZOOM_package_option_get` and `ZOOM_package_option_set` gets and sets options.

`ZOOM_package_send` sends the package the via connection specified in `ZOOM_connection_package`. The *`type`* specifies the actual extended service package type to be sent.

Type	**Description**
itemorder	Item Order
update	Record Update
create	Database Create
drop	Database Drop
commit	Commit Operation

Table 3.6: Extended Service Type

Item Order

For Item Order, `type` must be set to `itemorder` in `ZOOM_package_send`.

There are two variants of item order: ILL-variant and XML document variant. In order to use the XML variant the setting `doc` must hold the XML item order document. If that setting is unset, the ILL-variant is used.

Option	Description	Default
package-name	Extended Service Request package name. Must be specified as part of a request.	none
user-id	User ID of Extended Service Package. Is a request option.	none
function	Function of package - one of `create`, `delete`, `modify`. Is a request option.	`create`
waitAction	Wait action for package. Possible values: `wait`, `waitIfPossible`, `dontWait` or `dontReturnPackage`.	`waitIfPossible`
targetReference	Target Reference. This is part of the response as returned by the server. Read it after a successful operation.	`none`

Table 3.7: Extended Service Common Options

Option	Description	Default
contact-name	ILL contact name	none
contact-phone	ILL contact phone	none
contact-email	ILL contact email	none
itemorder-setname	Name of result set for record	default
itemorder-item	Position for item (record) requested. An integer	1

Table 3.8: Item Order Options

Option
protocol-version-num
transaction-id,initial-requester-id,person-or-institution-symbol,person
transaction-id,initial-requester-id,person-or-institution-symbol,institution
transaction-id,initial-requester-id,name-of-person-or-institution,name-of-person
transaction-id,initial-requester-id,name-of-person-or-institution,name-of-institution
transaction-id,transaction-group-qualifier
transaction-id,transaction-qualifier
transaction-id,sub-transaction-qualifier
service-date-time,this,date
service-date-time,this,time
service-date-time,original,date
service-date-time,original,time
requester-id,person-or-institution-symbol,person
requester-id,person-or-institution-symbol,institution
requester-id,name-of-person-or-institution,name-of-person
requester-id,name-of-person-or-institution,name-of-institution
responder-id,person-or-institution-symbol,person
responder-id,person-or-institution-symbol,institution
responder-id,name-of-person-or-institution,name-of-person
responder-id,name-of-person-or-institution,name-of-institution
transaction-type
delivery-address,postal-address,name-of-person-or-institution,name-of-person
delivery-address,postal-address,name-of-person-or-institution,name-of-institution
delivery-address,postal-address,extended-postal-delivery-address
delivery-address,postal-address,street-and-number
delivery-address,postal-address,post-office-box
delivery-address,postal-address,city
delivery-address,postal-address,region
delivery-address,postal-address,country
delivery-address,postal-address,postal-code
delivery-address,electronic-address,telecom-service-identifier
delivery-address,electronic-address,telecom-service-addreess
billing-address,postal-address,name-of-person-or-institution,name-of-person
billing-address,postal-address,name-of-person-or-institution,name-of-institution
billing-address,postal-address,extended-postal-delivery-address
billing-address,postal-address,street-and-number
billing-address,postal-address,post-office-box
billing-address,postal-address,city
billing-address,postal-address,region
billing-address,postal-address,country
billing-address,postal-address,postal-code
billing-address,electronic-address,telecom-service-identifier
billing-address,electronic-address,telecom-service-addreess
ill-service-type
requester-optional-messages,can-send-RECEIVED
requester-optional-messages,can-send-RETURNED
requester-optional-messages,requester-SHIPPED
requester-optional-messages,requester-CHECKED-IN
search-type,level-of-service
search-type,need-before-date
search-type,expiry-date
search-type,expiry-flag

Option	Description	Default
action	The update action. One of `specialUpdate`, `recordInsert`, `recordReplace`, `recordDelete`, `elementUpdate`.	`specialUpdate (recordInsert for updateVersion=1 which does not support specialUpdate)`
recordIdOpaque	Opaque Record ID	none
recordIdNumber	Record ID number	none
record	The record itself	none
recordOpaque	Specifies an opaque record which is encoded as an ASN.1 ANY type with the OID as given by option `syntax` (see below). Option `recordOpaque` is an alternative to record - and `record` option (above) is ignored if recordOpaque is set. This option is only available in YAZ 3.0.35 and later, and is meant to facilitate Updates with servers from OCLC.	none
syntax	The record syntax (transfer syntax). Is a string that is a known record syntax.	no syntax
databaseName	Database from connection object	Default
correlationInfo.note	Correlation Info Note (string)	none
correlationInfo.id	Correlation Info ID (integer)	none
elementSetName	Element Set for Record	none
updateVersion	Record Update version which holds one of the values 1, 2 or 3. Each version has a distinct OID: 1.2.840.10003.9.5 (first version) , 1.2.840.10003.9.5.1 (second version) and 1.2.840.10003.9.5.1.1 (third and newest version).	3

Table 3.10: Record Update Options

Record Update

For Record Update, `type` must be set to `update` in `ZOOM_package_send`.

Database Create

For Database Create, `type` must be set to `create` in `ZOOM_package_send`.

Option	Description	Default
databaseName	Database from connection object	Default

Table 3.11: Database Create Options

Database Drop

For Database Drop, `type` must be set to `drop` in `ZOOM_package_send`.

Option	Description	Default
databaseName	Database from connection object	Default

Table 3.12: Database Drop Options

Commit Operation

For Commit, `type` must be set to `commit` in `ZOOM_package_send`.

Protocol behavior

All the extended services are Z39.50-only.

Note
The database create, drop, and commit services are privately defined operations. Refer to `esadmin.asn` in YAZ for the ASN.1 definitions.

Options

Most ZOOM objects provide a way to specify options to change behavior. From an implementation point of view, a set of options is just like an associative array / hash.

```
ZOOM_options ZOOM_options_create(void);

ZOOM_options ZOOM_options_create_with_parent(ZOOM_options parent);

void ZOOM_options_destroy(ZOOM_options opt);


const char *ZOOM_options_get(ZOOM_options opt, const char *name);

void ZOOM_options_set(ZOOM_options opt, const char *name,
                      const char *v);


typedef const char *(*ZOOM_options_callback)
                       (void *handle, const char *name);

ZOOM_options_callback
        ZOOM_options_set_callback(ZOOM_options opt,
                                  ZOOM_options_callback c,
                                  void *handle);
```

Query conversions

```
int ZOOM_query_cql2rpn(ZOOM_query s, const char *cql_str,
                       ZOOM_connection conn);

int ZOOM_query_ccl2rpn(ZOOM_query s, const char *ccl_str,
                       const char *config,
                       int *ccl_error, const char **error_string,
                       int *error_pos);
```

`ZOOM_query_cql2rpn` translates the CQL string, client-side, into RPN which may be passed to the server. This is useful for servers that don't themselves support CQL, for which `ZOOM_query_cql` is useless. 'conn' is used only as a place to stash diagnostics if compilation fails; if this information is not needed, a null pointer may be used. The CQL conversion is driven by option `cqlfile` from connection conn. This specifies a conversion file (e.g. pqf.properties) which *must* be present.

`ZOOM_query_ccl2rpn` translates the CCL string, client-side, into RPN which may be passed to the server. The conversion is driven by the specification given by `config`. Upon completion 0 is returned on success; -1 is returned on failure. On failure `error_string` and `error_pos` hold the error message and position of first error in original CCL string.

Events

If you're developing non-blocking applications, you have to deal with events.

```
int ZOOM_event(int no, ZOOM_connection *cs);
```

The `ZOOM_event` executes pending events for a number of connections. Supply the number of connections in `no` and an array of connections in `cs` (`cs[0] ...cs[no-1]`). A pending event could be sending a search, receiving a response, etc. When an event has occurred for one of the connections, this function returns a positive integer `n` denoting that an event occurred for connection `cs[n-1]`. When no events are pending for the connections, a value of zero is returned. To ensure that all outstanding requests are performed, call this function repeatedly until zero is returned.

If `ZOOM_event` returns, and returns non-zero, the last event that occurred can be expected.

```
int ZOOM_connection_last_event(ZOOM_connection cs);
```

`ZOOM_connection_last_event` returns an event type (integer) for the last event.

Event	Description
ZOOM_EVENT_NONE	No event has occurred
ZOOM_EVENT_CONNECT	TCP/IP connect has initiated
ZOOM_EVENT_SEND_DATA	Data has been transmitted (sending)
ZOOM_EVENT_RECV_DATA	Data has been received
ZOOM_EVENT_TIMEOUT	Timeout
ZOOM_EVENT_UNKNOWN	Unknown event
ZOOM_EVENT_SEND_APDU	An APDU has been transmitted (sending)
ZOOM_EVENT_RECV_APDU	An APDU has been received
ZOOM_EVENT_RECV_RECORD	A result-set record has been received
ZOOM_EVENT_RECV_SEARCH	A search result has been received

Table 3.13: ZOOM Event IDs

Chapter 4

Generic server

Introduction

If you aren't into documentation, a good way to learn how the back end interface works is to look at the `backend.h` file. Then, look at the small dummy-server in `ztest/ztest.c`. The `backend.h` file also makes a good reference, once you've chewed your way through the prose of this file.

If you have a database system that you would like to make available by means of Z39.50 or SRU, YAZ basically offers two options. You can use the APIs provided by the Z39.50 ASN.1, ODR, and COMSTACK modules to create and decode PDUs, and exchange them with a client. Using this low-level interface gives you access to all fields and options of the protocol, and you can construct your server as close to your existing database as you like. It is also a fairly involved process, requiring you to set up an event-handling mechanism, protocol state machine, etc. To simplify server implementation, we have implemented a compact and simple, but reasonably full-functioned server-frontend that will handle most of the protocol mechanics, while leaving you to concentrate on your database interface.

Note
The backend interface was designed in anticipation of a specific integration task, while still attempting to achieve some degree of generality. We realize fully that there are points where the interface can be improved significantly. If you have specific functions or parameters that you think could be useful, send us a mail (or better, sign on to the mailing list referred to in the top-level README file). We will try to fit good suggestions into future releases, to the extent that it can be done without requiring too many structural changes in existing applications.

Note
The YAZ server does not support XCQL.

The Database Frontend

We refer to this software as a generic database frontend. Your database system is the *backend database*, and the interface between the two is called the *backend API*. The backend API consists of a small number of

function handlers and structure definitions. You are required to provide the `main()` routine for the server (which can be quite simple), as well as a set of handlers to match each of the prototypes. The interface functions that you write can use any mechanism you like to communicate with your database system: You might link the whole thing together with your database application and access it by function calls; you might use IPC to talk to a database server somewhere; or you might link with third-party software that handles the communication for you (like a commercial database client library). At any rate, the handlers will perform the tasks of:

- Initialization.
- Searching.
- Fetching records.
- Scanning the database index (optional - if you wish to implement SCAN).
- Extended Services (optional).
- Result-Set Delete (optional).
- Result-Set Sort (optional).
- Return Explain for SRU (optional).

(more functions will be added in time to support as much of Z39.50-1995 as possible).

The Backend API

The header file that you need to use the interface are in the `include/yaz` directory. It's called `backend.h`. It will include other files from the `include/yaz` directory, so you'll probably want to use the -I option of your compiler to tell it where to find the files. When you run `make` in the top-level YAZ directory, everything you need to create your server is to link with the `lib/libyaz.la` library.

Your main() Routine

As mentioned, your `main()` routine can be quite brief. If you want to initialize global parameters, or read global configuration tables, this is the place to do it. At the end of the routine, you should call the function

```
int statserv_main(int argc, char **argv,
                  bend_initresult *(*bend_init)(bend_initrequest *r),
                  void (*bend_close)(void *handle));
```

The third and fourth arguments are pointers to handlers. Handler `bend_init` is called whenever the server receives an Initialize Request, so it serves as a Z39.50 session initializer. The `bend_close` handler is called when the session is closed.

`statserv_main` will establish listening sockets according to the parameters given. When connection requests are received, the event handler will typically `fork()` and create a sub-process to handle a new connection. Alternatively the server may be setup to create threads for each connection. If you do use global variables and forking, you should be aware, then, that these cannot be shared between associations, unless you explicitly disable forking by command line parameters.

The server provides a mechanism for controlling some of its behavior without using command-line options. The function

```
statserv_options_block *statserv_getcontrol(void);
```

will return a pointer to a `struct statserv_options_block` describing the current default settings of the server. The structure contains these elements:

`int dynamic` A boolean value, which determines whether the server will fork on each incoming request (TRUE), or not (FALSE). Default is TRUE. This flag is only read by UNIX-based servers (WIN32-based servers do not fork).

`int threads` A boolean value, which determines whether the server will create a thread on each incoming request (TRUE), or not (FALSE). Default is FALSE. This flag is only read by UNIX-based servers that offer POSIX Threads support. WIN32-based servers always operate in threaded mode.

`int inetd` A boolean value, which determines whether the server will operate under a UNIX INET daemon (inetd). Default is FALSE.

`char logfile[ODR_MAXNAME+1]` File for diagnostic output ("": stderr).

`char apdufile[ODR_MAXNAME+1]` Name of file for logging incoming and outgoing APDUs ("": don't log APDUs, "-": `stderr`).

`char default_listen[1024]` Same form as the command-line specification of listener address. "": no default listener address. Default is to listen at "tcp:@:9999". You can only specify one default listener address in this fashion.

`enum oid_proto default_proto;` Either `PROTO_Z3950` or `PROTO_SR`. Default is `PROTO_Z39_50`.

`int idle_timeout;` Maximum session idle-time, in minutes. Zero indicates no (infinite) timeout. Default is 15 minutes.

`int maxrecordsize;` Maximum permissible record (message) size. Default is 64 MB. This amount of memory will only be allocated if a client requests a very large amount of records in one operation (or a big record). Set it to a lower number if you are worried about resource consumption on your host system.

`char configname[ODR_MAXNAME+1]` Passed to the backend when a new connection is received.

char setuid[ODR_MAXNAME+1] Set user id to the user specified, after binding the listener addresses.

void (*bend_start)(struct statserv_options_block *p) Pointer to function which is called after the command line options have been parsed - but before the server starts listening. For forked UNIX servers, this handler is called in the mother process; for threaded servers, this handler is called in the main thread. The default value of this pointer is NULL in which case it isn't invoked by the frontend server. When the server operates as an NT service, this handler is called whenever the service is started.

void (*bend_stop)(struct statserv_options_block *p) Pointer to function which is called whenever the server has stopped listening for incoming connections. This function pointer has a default value of NULL in which case it isn't called. When the server operates as an NT service, this handler is called whenever the service is stopped.

void *handle User defined pointer (default value NULL). This is a per-server handle that can be used to specify "user-data". Do not confuse this with the session-handle as returned by bend_init.

The pointer returned by `statserv_getcontrol` points to a static area. You are allowed to change the contents of the structure, but the changes will not take effect until you call

```
void statserv_setcontrol(statserv_options_block *block);
```

Note
You should generally update this structure before calling `statserv_main()`.

The Backend Functions

For each service of the protocol, the backend interface declares one or two functions. You are required to provide implementations of the functions representing the services that you wish to implement.

Init

```
bend_initresult (*bend_init)(bend_initrequest *r);
```

This handler is called once for each new connection request, after a new process/thread has been created, and an Initialize Request has been received from the client. The pointer to the `bend_init` handler is passed in the call to `statserv_start`.

This handler is also called when operating in SRU mode - when a connection has been made (even though SRU does not offer this service).

Unlike previous versions of YAZ, the `bend_init` also serves as a handler that defines the Z39.50 services that the backend intends to support. Pointers to *all* service handlers, including search - and fetch must be specified here in this handler.

The request - and result structures are defined as

```
typedef struct bend_initrequest
{
    /** \brief user/name/password to be read */
    Z_IdAuthentication *auth;
    /** \brief encoding stream (for results) */
    ODR stream;
    /** \brief printing stream */
    ODR print;
    /** \brief decoding stream (use stream for results) */
    ODR decode;
    /** \brief reference ID */
    Z_ReferenceId *referenceId;
    /** \brief peer address of client */
    char *peer_name;

    /** \brief character set and language negotiation

    see include/yaz/z-charneg.h
    */
    Z_CharSetandLanguageNegotiation *charneg_request;

    /** \brief character negotiation response */
    Z_External *charneg_response;

    /** \brief character set (encoding) for query terms

    This is NULL by default. It should be set to the native character
    set that the backend assumes for query terms */
    char *query_charset;

    /** \brief whether query_charset also applies to records

    Is 0 (No) by default. Set to 1 (yes) if records is in the same
    character set as queries. If in doubt, use 0 (No).
    */
    int records_in_same_charset;

    char *implementation_id;
    char *implementation_name;
    char *implementation_version;

    /** \brief Z39.50 sort handler */
    int (*bend_sort)(void *handle, bend_sort_rr *rr);
    /** \brief SRU/Z39.50 search handler */
    int (*bend_search)(void *handle, bend_search_rr *rr);
    /** \brief SRU/Z39.50 fetch handler */
    int (*bend_fetch)(void *handle, bend_fetch_rr *rr);
```

```
    /** \brief SRU/Z39.50 present handler */
    int (*bend_present)(void *handle, bend_present_rr *rr);
    /** \brief Z39.50 extended services handler */
    int (*bend_esrequest) (void *handle, bend_esrequest_rr *rr);
    /** \brief Z39.50 delete result set handler */
    int (*bend_delete)(void *handle, bend_delete_rr *rr);
    /** \brief Z39.50 scan handler */
    int (*bend_scan)(void *handle, bend_scan_rr *rr);
    /** \brief Z39.50 segment facility handler */
    int (*bend_segment)(void *handle, bend_segment_rr *rr);
    /** \brief SRU explain handler */
    int (*bend_explain)(void *handle, bend_explain_rr *rr);
    /** \brief SRU scan handler */
    int (*bend_srw_scan)(void *handle, bend_scan_rr *rr);
    /** \brief SRU record update handler */
    int (*bend_srw_update)(void *handle, bend_update_rr *rr);

    /** \brief whether named result sets are supported (0=disable, 1=enable)
    int named_result_sets;
} bend_initrequest;

typedef struct bend_initresult
{
    int errcode;                /* 0==OK */
    char *errstring;            /* system error string or NULL */
    void *handle;               /* private handle to the backend module */
} bend_initresult;
```

In general, the server frontend expects that the `bend_*result` pointer that you return is valid at least until the next call to a `bend_* function`. This applies to all of the functions described herein. The parameter structure passed to you in the call belongs to the server frontend, and you should not make assumptions about its contents after the current function call has completed. In other words, if you want to retain any of the contents of a request structure, you should copy them.

The `errcode` should be zero if the initialization of the backend went well. Any other value will be interpreted as an error. The `errstring` isn't used in the current version, but one option would be to stick it in the initResponse as a VisibleString. The `handle` is the most important parameter. It should be set to some value that uniquely identifies the current session to the backend implementation. It is used by the frontend server in any future calls to a backend function. The typical use is to set it to point to a dynamically allocated state structure that is private to your backend module.

The `auth` member holds the authentication information part of the Z39.50 Initialize Request. Interpret this if your server requires authentication.

The members `peer_name`, `implementation_id`, `implementation_name` and `implementation_version` holds DNS of client, ID of implementor, name of client (Z39.50) implementation - and version.

The bend_ - members are set to NULL when bend_init is called. Modify the pointers by setting them to point to backend functions.

Search and Retrieve

We now describe the handlers that are required to support search - and retrieve. You must support two functions - one for search - and one for fetch (retrieval of one record). If desirable you can provide a third handler which is called when a present request is received which allows you to optimize retrieval of multiple-records.

```
int (*bend_search) (void *handle, bend_search_rr *rr);

typedef struct {
    char *setname;              /* name to give to this set */
    int replace_set;            /* replace set, if it already exists */
    int num_bases;              /* number of databases in list */
    char **basenames;           /* databases to search */
    Z_ReferenceId *referenceId;/* reference ID */
    Z_Query *query;             /* query structure */
    ODR stream;                 /* encode stream */
    ODR decode;                 /* decode stream */
    ODR print;                  /* print stream */

    bend_request request;
    bend_association association;
    int *fd;
    int hits;                   /* number of hits */
    int errcode;                /* 0==OK */
    char *errstring;            /* system error string or NULL */
    Z_OtherInformation *search_info; /* additional search info */
    char *srw_sortKeys;         /* holds SRU/SRW sortKeys info */
    char *srw_setname;          /* holds SRU/SRW generated resultsetID */
    int *srw_setnameIdleTime;   /* holds SRU/SRW life-time */
    int estimated_hit_count;    /* if hit count is estimated */
    int partial_resultset;      /* if result set is partial */
} bend_search_rr;
```

The bend_search handler is a fairly close approximation of a protocol Z39.50 Search Request - and Response PDUs. The setname is the resultSetName from the protocol. You are required to establish a mapping between the set name and whatever your backend database likes to use. Similarly, the replace_set is a boolean value corresponding to the resultSetIndicator field in the protocol. num_bases/basenames is a length of/array of character pointers to the database names provided by the client. The query is the full query structure as defined in the protocol ASN.1 specification. It can be either of the possible query types, and it's up to you to determine if you can handle the provided query type. Rather than reproduce the C interface here, we'll refer you to the structure definitions in the file include/yaz/

z-core.h. If you want to look at the attributeSetId OID of the RPN query, you can either match it against your own internal tables, or you can use the OID tools.

The structure contains a number of hits, and an errcode/errstring pair. If an error occurs during the search, or if you're unhappy with the request, you should set the errcode to a value from the BIB-1 diagnostic set. The value will then be returned to the user in a nonsurrogate diagnostic record in the response. The errstring, if provided, will go in the addinfo field. Look at the protocol definition for the defined error codes, and the suggested uses of the addinfo field.

The bend_search handler is also called when the frontend server receives a SRU SearchRetrieveRequest. For SRU, a CQL query is usually provided by the client. The CQL query is available as part of Z_Query structure (note that CQL is now part of Z39.50 via an external). To support CQL in existing implementations that only do Type-1, we refer to the CQL-to-PQF tool described here.

To maintain backwards compatibility, the frontend server of yaz always assume that error codes are BIB-1 diagnostics. For SRU operation, a Bib-1 diagnostic code is mapped to SRU diagnostic.

```
int (*bend_fetch) (void *handle, bend_fetch_rr *rr);

typedef struct bend_fetch_rr {
    char *setname;              /* set name */
    int number;                 /* record number */
    Z_ReferenceId *referenceId;/* reference ID */
    Odr_oid *request_format;    /* format, transfer syntax (OID) */
    Z_RecordComposition *comp; /* Formatting instructions */
    ODR stream;                 /* encoding stream - memory source if req */
    ODR print;                  /* printing stream */

    char *basename;             /* name of database that provided record */
    int len;                    /* length of record or -1 if structured */
    char *record;               /* record */
    int last_in_set;            /* is it?  */
    Odr_oid *output_format;     /* response format/syntax (OID) */
    int errcode;                /* 0==success */
    char *errstring;            /* system error string or NULL */
    int surrogate_flag;         /* surrogate diagnostic */
    char *schema;               /* string record schema input/output */
} bend_fetch_rr;
```

The frontend server calls the bend_fetch handler when it needs database records to fulfill a Z39.50 Search Request, a Z39.50 Present Request or a SRU SearchRetrieveRequest. The setname is simply the name of the result set that holds the reference to the desired record. The number is the offset into the set (with 1 being the first record in the set). The format field is the record format requested by the client (See Section 7.2). A value of NULL for format indicates that the client did not request a specific format. The stream argument is an ODR stream which should be used for allocating space for structured data records. The stream will be reset when all records have been assembled, and the response package has been transmitted. For unstructured data, the backend is responsible for maintaining a static or dynamic buffer for the record between calls.

If a SRU SearchRetrieveRequest is received by the frontend server, the `referenceId` is NULL and the `format` (transfer syntax) is the OID for XML. The schema for SRU is stored in both the `Z_RecordComposition` structure and `schema` (simple string).

In the structure, the `basename` is the name of the database that holds the record. `len` is the length of the record returned, in bytes, and `record` is a pointer to the record. `last_in_set` should be nonzero only if the record returned is the last one in the given result set. `errcode` and `errstring`, if given, will be interpreted as a global error pertaining to the set, and will be returned in a non-surrogate-diagnostic. If you wish to return the error as a surrogate-diagnostic (local error) you can do this by setting `surrogate_flag` to 1 also.

If the `len` field has the value -1, then `record` is assumed to point to a constructed data type. The `format` field will be used to determine which encoder should be used to serialize the data.

Note
If your backend generates structured records, it should use `odr_malloc()` on the provided stream for allocating data: This allows the frontend server to keep track of the record sizes.

The `format` field is mapped to an object identifier in the direct reference of the resulting EXTERNAL representation of the record.

Note
The current version of YAZ only supports the direct reference mode.

```
int (*bend_present) (void *handle, bend_present_rr *rr);

typedef struct {
    char *setname;              /* set name */
    int start;
    int number;                 /* record number */
    Odr_oid *format;            /* format, transfer syntax (OID) */
    Z_ReferenceId *referenceId;/* reference ID */
    Z_RecordComposition *comp; /* Formatting instructions */
    ODR stream;                 /* encoding stream - memory source if requir
    ODR print;                  /* printing stream */
    bend_request request;
    bend_association association;

    int hits;                   /* number of hits */
    int errcode;                /* 0==OK */
    char *errstring;            /* system error string or NULL */
} bend_present_rr;
```

The `bend_present` handler is called when the server receives a Z39.50 Present Request. The `setname`, `start` and `number` is the name of the result set - start position - and number of records to be retrieved

respectively. `format` and `comp` is the preferred transfer syntax and element specifications of the present request.

Note that this is handler serves as a supplement for `bend_fetch` and need not to be defined in order to support search - and retrieve.

Delete

For back-ends that supports delete of a result set, only one handler must be defined.

```
int (*bend_delete)(void *handle, bend_delete_rr *rr);

typedef struct bend_delete_rr {
    int function;
    int num_setnames;
    char **setnames;
    Z_ReferenceId *referenceId;
    int delete_status;      /* status for the whole operation */
    int *statuses;          /* status each set - indexed as setnames */
    ODR stream;
    ODR print;
} bend_delete_rr;
```

Note

The delete set function definition is rather primitive, mostly because we have had no practical need for it as of yet. If someone wants to provide a full delete service, we'd be happy to add the extra parameters that are required. Are there clients out there that will actually delete sets they no longer need?

Scan

For servers that wish to offer the scan service one handler must be defined.

```
int (*bend_scan)(void *handle, bend_scan_rr *rr);

typedef enum {
    BEND_SCAN_SUCCESS,  /* ok */
    BEND_SCAN_PARTIAL   /* not all entries could be found */
} bend_scan_status;

typedef struct bend_scan_rr {
    int num_bases;      /* number of elements in databaselist */
    char **basenames;   /* databases to search */
    Odr_oid *attributeset;
    Z_ReferenceId *referenceId; /* reference ID */
```

```
    Z_AttributesPlusTerm *term;
    ODR stream;          /* encoding stream - memory source if required */
    ODR print;           /* printing stream */

    int *step_size;      /* step size */
    int term_position;   /* desired index of term in result list/returned */
    int num_entries;     /* number of entries requested/returned */

    /* scan term entries. The called handler does not have
       to allocate this. Size of entries is num_entries (see above) */
    struct scan_entry *entries;
    bend_scan_status status;
    int errcode;
    char *errstring;
    char *scanClause;    /* CQL scan clause */
    char *setname;       /* Scan in result set (NULL if omitted) */
} bend_scan_rr;
```

This backend server handles both Z39.50 scan and SRU scan. In order for a handler to distinguish between SRU (CQL) scan Z39.50 Scan, it must check for a non-NULL value of `scanClause`.

Note

If designed today, it would be a choice using a union or similar, but that would break binary compatibility with existing servers.

Application Invocation

The finished application has the following invocation syntax (by way of `statserv_main()`):

`application` [`-install`] [`-installa`] [`-remove`] [`-a` *`file`*] [`-v` *`level`*] [`-l` *`file`*] [`-u` *`uid`*] [`-c` *`config`*] [`-f` *`vconfig`*] [`-C` *`fname`*] [`-t` *`minutes`*] [`-k` *`kilobytes`*] [`-K`] [`-d` *`daemon`*] [`-w` *`dir`*] [`-p` *`pidfile`*] [`-r` *`kilobytes`*] [`-ziDSTV1`] [listener-spec...]

The options are:

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

`-S` Don't fork or make threads on connection requests. This is good for debugging, but not recommended for real operation: Although the server is asynchronous and non-blocking, it can be nice to keep a software malfunction (okay then, a crash) from affecting all current users.

`-1` Like `-S` but after one session the server exits. This mode is for debugging *only*.

`-T` Operate the server in threaded mode. The server creates a thread for each connection rather than fork a process. Only available on UNIX systems that offer POSIX threads.

-s Use the SR protocol (obsolete).

-z Use the Z39.50 protocol (default). This option and `-s` complement each other. You can use both multiple times on the same command line, between listener-specifications (see below). This way, you can set up the server to listen for connections in both protocols concurrently, on different local ports.

-l ***file*** The logfile.

-c ***config*** A user option that serves as a specifier for some sort of configuration, usually a filename. The argument to this option is transferred to member `configname` of the `statserv_options_block`.

-f ***vconfig*** This specifies an XML file that describes one or more YAZ frontend virtual servers.

-C ***fname*** Sets SSL certificate file name for server (PEM).

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

-u ***uid*** Set user ID. Sets the real UID of the server process to that of the given user. 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 ***dir*** The server changes to this directory before listening to incoming connections. This option is useful when the server is operating from the inetd daemon (see `-i`).

-p ***pidfile*** Specifies that the server should write its Process ID to the file given by *`pidfile`*. A typical location would be `/var/run/yaz-ztest.pid`.

-i Use this to make the the server run from the inetd server (UNIX only).

-D Use this to make the server put itself in the background and run as a daemon. If neither `-i` nor `-D` is given, the server starts in the foreground.

-install Use this to install the server as an NT service (Windows NT/2000/XP only). Control the server by going to the Services in the Control Panel.

-installa Use this to install the server as an NT service and mark it as "auto-start. Control the server by going to the Services in the Control Panel.

-remove Use this to remove the server from the NT services (Windows NT/2000/XP only).

-t ***minutes*** Idle session timeout, in minutes.

-k ***size*** Maximum record size/message size, in kilobytes.

-K Forces no-keepalive for HTTP sessions. By default GFS will keep sessions alive for HTTP 1.1 sessions (as defined by the standard). Using this option will force GFS to close the connection for each operation.

-r ***size*** Maximum size of log file before rotation occurs, in kilobytes. Default size is 1048576 k (=1 GB).

-d ***daemon*** Set name of daemon to be used in hosts access file. See hosts_access(5) and tcpd(8).

-m ***time-format*** Sets the format of time-stamps in the log-file. Specify a string in the input format to `strftime()`.

-V Display YAZ version and exit.

A listener specification consists of a transport mode followed by a colon (:) followed by a listener address. The transport mode is either `tcp`, `unix:` or `ssl`.

For TCP and SSL, an address has the form

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

The port number defaults to 210 (standard Z39.50 port).

For UNIX, the address is the filename of socket.

For TCP/IP and SSL, the special hostnames `@`, maps to `IN6ADDR_ANY_INIT` with IPV4 binding as well (bindv6only=0), The special hostname `@4` binds to `INADDR_ANY` (IPV4 only listener). The special hostname `@6` binds to `IN6ADDR_ANY_INIT` with bindv6only=1 (IPV6 only listener).

Example 4.1 Running the GFS on Unix

Assuming the server application *appname* is started as root, the following will make it listen on port 210. The server will change identity to `nobody` and write its log to `/var/log/app.log`.

```
application -l /var/log/app.log -u nobody tcp:@:210
```

The server will accept Z39.50 requests and offer SRU service on port 210.

Example 4.2 Setting up Apache as SRU Frontend

If you use Apache as your public web server and want to offer HTTP port 80 access to the YAZ server on 210, you can use the `ProxyPass` directive. If you have virtual host `srw.mydomain` you can use the following directives in Apache's httpd.conf:

```
<VirtualHost *>
 ErrorLog /home/srw/logs/error_log
 TransferLog /home/srw/logs/access_log
 ProxyPass / http://srw.mydomain:210/
</VirtualHost>
```

The above is for the Apache 1.3 series.

Example 4.3 Running a server with local access only

A server that is only being accessed from the local host should listen on UNIX file socket rather than an Internet socket. To listen on `/tmp/mysocket` start the server as follows:

```
application unix:/tmp/mysocket
```

GFS Configuration and Virtual Hosts

The Virtual hosts mechanism allows a YAZ frontend server to support multiple backends. A backend is selected on the basis of the TCP/IP binding (port+listening adddress) and/or the virtual host.

A backend can be configured to execute in a particular working directory. Or the YAZ frontend may perform CQL to RPN conversion, thus allowing traditional Z39.50 backends to be offered as a SRW/SRU service. SRW/SRU Explain information for a particular backend may also be specified.

For the HTTP protocol, the virtual host is specified in the Host header. For the Z39.50 protocol, the virtual host is specified as in the Initialize Request in the OtherInfo, OID 1.2.840.10003.10.1000.81.1.

Note
Not all Z39.50 clients allow the VHOST information to be set. For those, the selection of the backend must rely on the TCP/IP information alone (port and address).

The YAZ frontend server uses XML to describe the backend configurations. Command-line option `-f` specifies filename of the XML configuration.

The configuration uses the root element `yazgfs`. This element includes a list of `listen` elements, followed by one or more `server` elements.

The `listen` describes listener (transport end point), such as TCP/IP, Unix file socket or SSL server. Content for a listener:

CDATA (required) The CDATA for the `listen` element holds the listener string, such as `tcp:@:210`, `tcp:server1:2100`, etc.

attribute `id` (optional) Identifier for this listener. This may be referred to from server sections.

Note
We expect more information to be added for the listen section in a future version, such as CERT file for SSL servers.

The `server` describes a server and the parameters for this server type. Content for a server:

attribute `id` (optional) Identifier for this server. Currently not used for anything, but it might be for logging purposes.

attribute `listenref` (optional) Specifies one or more listeners for this server. Each server ID is separated by a comma. If this attribute is not given, the server is accessible from all listeners. In order for the server to be used for real, however, the virtual host must match if specified in the configuration.

element `config` (optional) Specifies the server configuration. This is equivalent to the config specified using command line option `-c`.

element `directory` (optional) Specifies a working directory for this backend server. If specified, the YAZ frontend changes current working directory to this directory whenever a backend of this type is started (backend handler bend_start), stopped (backend handler hand_stop) and initialized (bend_init).

element `host` (optional) Specifies the virtual host for this server. If this is specified a client *must* specify this host string in order to use this backend.

element `cql2rpn` (optional) Specifies a filename that includes CQL to RPN conversion for this backend server. See Section 7.1.3.4. If given, the backend server will only "see" a Type-1/RPN query.

element `ccl2rpn` (optional) Specifies a filename that includes CCL to RPN conversion for this backend server. See Section 7.1.2.2. If given, the backend server will only "see" a Type-1/RPN query.

element `stylesheet` (optional) Specifies the stylesheet reference to be part of SRU HTTP responses when the client does not specify one. If none is given, then if the client does not specify one, then no stylesheet reference is part of the SRU HTTP response.

element `client_query_charset` (optional) If specified, a conversion from the character set given to UTF-8 is performed by the generic frontend server. It is only executed for Z39.50 search requests (SRU/Solr are assumed to be UTF-8 encoded already).

element `docpath` (optional) Specifies a path for local file access using HTTP. All URLs with a leading prefix (/ excluded) that matches the value of `docpath` are used for file access. For example, if the server is to offer access in directory `xsl`, the docpath would be `xsl` and all URLs of the form `http://host/xsl` will result in a local file access.

element `explain` (optional) Specifies SRW/SRU ZeeRex content for this server. Copied verbatim to the client. As things are now, some of the Explain content seem redundant because host information, etc. is also stored elsewhere.

element `maximumrecordsize` (optional) Specifies maximum record size/message size, in bytes. This value also serves as the maximum size of *incoming* packages (for Record Updates etc). It's the same value as that given by the `-k` option.

element `retrievalinfo` (optional) Enables the retrieval facility to support conversions and specifications of record formats/types. See Section 7.6 for more information.

The XML below configures a server that accepts connections from two ports, TCP/IP port 9900 and a local UNIX file socket. We name the TCP/IP server `public` and the other server `internal`.

```
<yazgfs>
 <listen id="public">tcp:@:9900</listen>
 <listen id="internal">unix:/var/tmp/socket</listen>
 <server id="server1">
   <host>server1.mydomain</host>
   <directory>/var/www/s1</directory>
   <config>config.cfg</config>
 </server>
 <server id="server2" listenref="public,internal">
   <host>server2.mydomain</host>
```

```
    <directory>/var/www/s2</directory>
    <config>config.cfg</config>
    <cql2rpn>../etc/pqf.properties</cql2rpn>
    <explain xmlns="http://explain.z3950.org/dtd/2.0/">
      <serverInfo>
        <host>server2.mydomain</host>
        <port>9900</port>
        <database>a</database>
      </serverInfo>
    </explain>
  </server>
  <server id="server3" listenref="internal">
    <directory>/var/www/s3</directory>
    <config>config.cfg</config>
  </server>
</yazgfs>
```

There are three configured backend servers. The first two servers, "server1" and "server2", can be reached by both listener addresses. "server1" is reached by all (two) since no listenref attribute is specified. "server2" is reached by the two listeners specified. In order to distinguish between the two, a virtual host has been specified for each server in the host elements.

For "server2" elements for CQL to RPN conversion is supported and explain information has been added (a short one here to keep the example small).

The third server, "server3" can only be reached via listener "internal".

Chapter 5

The Z39.50 ASN.1 Module

Introduction

The Z39.50 ASN.1 module provides you with a set of C struct definitions for the various PDUs of the Z39.50 protocol, as well as for the complex types appearing within the PDUs. For the primitive data types, the C representation often takes the form of an ordinary C language type, such as `Odr_int` which is equivalent to an integral C integer. For ASN.1 constructs that have no direct representation in C, such as general octet strings and bit strings, the ODR module (see section The ODR Module) provides auxiliary definitions.

The Z39.50 ASN.1 module is located in sub directory `z39.50`. There you'll find C files that implement encoders and decoders for the Z39.50 types. You'll also find the protocol definitions: `z3950v3.asn`, `esupdate.asn`, and others.

Preparing PDUs

A structure representing a complex ASN.1 type doesn't in itself contain the members of that type. Instead, the structure contains *pointers* to the members of the type. This is necessary, in part, to allow a mechanism for specifying which of the optional structure (SEQUENCE) members are present, and which are not. It follows that you will need to somehow provide space for the individual members of the structure, and set the pointers to refer to the members.

The conversion routines don't care how you allocate and maintain your C structures - they just follow the pointers that you provide. Depending on the complexity of your application, and your personal taste, there are at least three different approaches that you may take when you allocate the structures.

You can use static or automatic local variables in the function that prepares the PDU. This is a simple approach, and it provides the most efficient form of memory management. While it works well for flat PDUs like the InitRequest, it will generally not be sufficient for say, the generation of an arbitrarily complex RPN query structure.

You can individually create the structure and its members using the `malloc(2)` function. If you want to ensure that the data is freed when it is no longer needed, you will have to define a function that individually releases each member of a structure before freeing the structure itself.

You can use the `odr_malloc()` function (see Section 8.2 for details). When you use `odr_malloc()`, you can release all of the allocated data in a single operation, independent of any pointers and relations between the data. The `odr_malloc()` function is based on a "nibble-memory" scheme, in which large portions of memory are allocated, and then gradually handed out with each call to `odr_malloc()`. The next time you call `odr_reset()`, all of the memory allocated since the last call is recycled for future use (actually, it is placed on a free-list).

You can combine all of the methods described here. This will often be the most practical approach. For instance, you might use `odr_malloc()` to allocate an entire structure and some of its elements, while you leave other elements pointing to global or per-session default variables.

The Z39.50 ASN.1 module provides an important aid in creating new PDUs. For each of the PDU types (say, `Z_InitRequest`), a function is provided that allocates and initializes an instance of that PDU type for you. In the case of the InitRequest, the function is simply named `zget_InitRequest()`, and it sets up reasonable default value for all of the mandatory members. The optional members are generally initialized to null pointers. This last aspect is very important: it ensures that if the PDU definitions are extended after you finish your implementation (to accommodate new versions of the protocol, say), you won't get into trouble with uninitialized pointers in your structures. The functions use `odr_malloc()` to allocate the PDUs and its members, so you can free everything again with a single call to `odr_reset()`. We strongly recommend that you use the `zget_*` functions whenever you are preparing a PDU (in a C++ API, the `zget_` functions would probably be promoted to constructors for the individual types).

The prototype for the individual PDU types generally look like this:

```
Z_<type> *zget_<type>(ODR o);
```

e.g.:

```
Z_InitRequest *zget_InitRequest(ODR o);
```

The ODR handle should generally be your encoding stream, but it needn't be.

As well as the individual PDU functions, a function `zget_APDU()` is provided, which allocates a top-level Z-APDU of the type requested:

```
Z_APDU *zget_APDU(ODR o, int which);
```

The `which` parameter is (of course) the discriminator belonging to the `Z_APDU CHOICE` type. All of the interface described here is provided by the Z39.50 ASN.1 module, and you access it through the `proto.h` header file.

EXTERNAL Data

In order to achieve extensibility and adaptability to different application domains, the new version of the protocol defines many structures outside of the main ASN.1 specification, referencing them through ASN.1 EXTERNAL constructs. To simplify the construction and access to the externally referenced data, the Z39.50 ASN.1 module defines a specialized version of the EXTERNAL construct, called `Z_External`.It is defined thus:

```
typedef struct Z_External
{
    Odr_oid *direct_reference;
    int *indirect_reference;
    char *descriptor;
    enum
    {
        /* Generic types */
        Z_External_single = 0,
        Z_External_octet,
        Z_External_arbitrary,

        /* Specific types */
        Z_External_SUTRS,
        Z_External_explainRecord,
        Z_External_resourceReport1,
        Z_External_resourceReport2

    ...

    } which;
    union
    {
        /* Generic types */
        Odr_any *single_ASN1_type;
        Odr_oct *octet_aligned;
        Odr_bitmask *arbitrary;

        /* Specific types */
        Z_SUTRS *sutrs;
        Z_ExplainRecord *explainRecord;
        Z_ResourceReport1 *resourceReport1;
        Z_ResourceReport2 *resourceReport2;

        ...

    } u;
} Z_External;
```

When decoding, the Z39.50 ASN.1 module will attempt to determine which syntax describes the data by looking at the reference fields (currently only the direct-reference). For ASN.1 structured data, you need only consult the `which` field to determine the type of data. You can the access the data directly through

the union. When constructing data for encoding, you set the union pointer to point to the data, and set the `which` field accordingly. Remember also to set the direct (or indirect) reference to the correct OID for the data type. For non-ASN.1 data such as MARC records, use the `octet_aligned` arm of the union.

Some servers return ASN.1 structured data values (e.g. database records) as BER-encoded records placed in the `octet-aligned` branch of the EXTERNAL CHOICE. The ASN-module will *not* automatically decode these records. To help you decode the records in the application, the function

```
   Z_ext_typeent *z_ext_gettypebyref(const oid *oid);
```

can be used to retrieve information about the known, external data types. The function returns a pointer to a static area, or NULL, if no match for the given direct reference is found. The `Z_ext_typeent` is defined as:

```
typedef struct Z_ext_typeent
{
    int oid[OID_SIZE]; /* the direct-reference OID. */
    int what;          /* discriminator value for the external CHOICE */
    Odr_fun fun;       /* decoder function */
} Z_ext_typeent;
```

The `what` member contains the `Z_External` union discriminator value for the given type: For the SUTRS record syntax, the value would be `Z_External_sutrs`. The `fun` member contains a pointer to the function which encodes/decodes the given type. Again, for the SUTRS record syntax, the value of `fun` would be `z_SUTRS` (a function pointer).

If you receive an EXTERNAL which contains an octet-string value that you suspect of being an ASN.1-structured data value, you can use `z_ext_gettypebyref` to look for the provided direct-reference. If the return value is different from NULL, you can use the provided function to decode the BER string (see Section 8.2).

If you want to *send* EXTERNALs containing ASN.1-structured values in the octet-aligned branch of the CHOICE, this is possible too. However, on the encoding phase, it requires a somewhat involved juggling around of the various buffers involved.

If you need to add new, externally defined data types, you must update the struct above, in the source file `prt-ext.h`, as well as the encoder/decoder in the file `prt-ext.c`. When changing the latter, remember to update both the `arm` array and the list `type_table`, which drives the CHOICE biasing that is necessary to tell the different, structured types apart on decoding.

Note
Eventually, the EXTERNAL processing will most likely automatically insert the correct OIDs or indirect-refs. First, however, we need to determine how application-context management (specifically the presentation-context-list) should fit into the various modules.

PDU Contents Table

We include, for reference, a listing of the fields of each top-level PDU, as well as their default settings.

Field	Type	Default Value
referenceId	Z_ReferenceId	NULL
protocolVersion	Odr_bitmask	Empty bitmask
options	Odr_bitmask	Empty bitmask
preferredMessageSize	Odr_int	30*1024
maximumRecordSize	Odr_int	30*1024
idAuthentication	Z_IdAuthentication	NULL
implementationId	char*	"81"
implementationName	char*	"YAZ"
implementationVersion	char*	YAZ_VERSION
userInformationField	Z_UserInformation	NULL
otherInfo	Z_OtherInformation	NULL

Table 5.1: Default settings for PDU Initialize Request

Field	Type	Default Value
referenceId	Z_ReferenceId	NULL
protocolVersion	Odr_bitmask	Empty bitmask
options	Odr_bitmask	Empty bitmask
preferredMessageSize	Odr_int	30*1024
maximumRecordSize	Odr_int	30*1024
result	Odr_bool	TRUE
implementationId	char*	"id)"
implementationName	char*	"YAZ"
implementationVersion	char*	YAZ_VERSION
userInformationField	Z_UserInformation	NULL
otherInfo	Z_OtherInformation	NULL

Table 5.2: Default settings for PDU Initialize Response

Field	Type	Default Value
referenceId	Z_ReferenceId	NULL
smallSetUpperBound	Odr_int	0
largeSetLowerBound	Odr_int	1
mediumSetPresentNumber	Odr_int	0
replaceIndicator	Odr_bool	TRUE
resultSetName	char *	"default"
num_databaseNames	Odr_int	0
databaseNames	char **	NULL
smallSetElementSetNames	Z_ElementSetNames	NULL
mediumSetElementSetNames	Z_ElementSetNames	NULL
preferredRecordSyntax	Odr_oid	NULL
query	Z_Query	NULL
additionalSearchInfo	Z_OtherInformation	NULL
otherInfo	Z_OtherInformation	NULL

Table 5.3: Default settings for PDU Search Request

Field	Type	Default Value
referenceId	Z_ReferenceId	NULL
resultCount	Odr_int	0
numberOfRecordsReturned	Odr_int	0
nextResultSetPosition	Odr_int	0
searchStatus	Odr_bool	TRUE
resultSetStatus	Odr_int	NULL
presentStatus	Odr_int	NULL
records	Z_Records	NULL
additionalSearchInfo	Z_OtherInformation	NULL
otherInfo	Z_OtherInformation	NULL

Table 5.4: Default settings for PDU Search Response

Field	Type	Default Value
referenceId	Z_ReferenceId	NULL
resultSetId	char*	"default"
resultSetStartPoint	Odr_int	1
numberOfRecordsRequested	Odr_int	10
num_ranges	Odr_int	0
additionalRanges	Z_Range	NULL
recordComposition	Z_RecordComposition	NULL
preferredRecordSyntax	Odr_oid	NULL
maxSegmentCount	Odr_int	NULL
maxRecordSize	Odr_int	NULL
maxSegmentSize	Odr_int	NULL
otherInfo	Z_OtherInformation	NULL

Table 5.5: Default settings for PDU Present Request

Field	Type	Default Value
referenceId	Z_ReferenceId	NULL
numberOfRecordsReturned	Odr_int	0
nextResultSetPosition	Odr_int	0
presentStatus	Odr_int	Z_PresentStatus_success
records	Z_Records	NULL
otherInfo	Z_OtherInformation	NULL

Table 5.6: Default settings for PDU Present Response

Field	Type	Default Value
referenceId	Z_ReferenceId	NULL
deleteFunction	Odr_int	Z_DeleteResultSetRequest_list
num_ids	Odr_int	0
resultSetList	char**	NULL
otherInfo	Z_OtherInformation	NULL

Table 5.7: Default settings for Delete Result Set Request

Field	Type	Default Value
referenceId	Z_ReferenceId	NULL
deleteOperationStatus	Odr_int	Z_DeleteStatus_success
num_statuses	Odr_int	0
deleteListStatuses	Z_ListStatus**	NULL
numberNotDeleted	Odr_int	NULL
num_bulkStatuses	Odr_int	0
bulkStatuses	Z_ListStatus	NULL
deleteMessage	char*	NULL
otherInfo	Z_OtherInformation	NULL

Table 5.8: Default settings for Delete Result Set Response

Field	Type	Default Value
referenceId	Z_ReferenceId	NULL
num_databaseNames	Odr_int	0
databaseNames	char**	NULL
attributeSet	Odr_oid	NULL
termListAndStartPoint	Z_AttributesPlus...	NULL
stepSize	Odr_int	NULL
numberOfTermsRequested	Odr_int	20
preferredPositionInResponse	Odr_int	NULL
otherInfo	Z_OtherInformation	NULL

Table 5.9: Default settings for Scan Request

Field	Type	Default Value
referenceId	Z_ReferenceId	NULL
stepSize	Odr_int	NULL
scanStatus	Odr_int	Z_Scan_success
numberOfEntriesReturned	Odr_int	0
positionOfTerm	Odr_int	NULL
entries	Z_ListEntries	NULL
attributeSet	Odr_oid	NULL
otherInfo	Z_OtherInformation	NULL

Table 5.10: Default settings for Scan Response

Field	Type	Default Value
referenceId	Z_ReferenceId	NULL
requestedAction	Odr_int	Z_TriggerResourceCtrl_resou..
prefResourceReportFormat	Odr_oid	NULL
resultSetWanted	Odr_bool	NULL
otherInfo	Z_OtherInformation	NULL

Table 5.11: Default settings for Trigger Resource Control Request

Field	Type	Default Value
referenceId	Z_ReferenceId	NULL
suspendedFlag	Odr_bool	NULL
resourceReport	Z_External	NULL
partialResultsAvailable	Odr_int	NULL
responseRequired	Odr_bool	FALSE
triggeredRequestFlag	Odr_bool	NULL
otherInfo	Z_OtherInformation	NULL

Table 5.12: Default settings for Resource Control Request

Field	Type	Default Value
referenceId	Z_ReferenceId	NULL
continueFlag	bool_t	TRUE
resultSetWanted	bool_t	NULL
otherInfo	Z_OtherInformation	NULL

Table 5.13: Default settings for Resource Control Response

Field	Type	Default Value
referenceId	Z_ReferenceId	NULL
which	enum	Z_AccessRequest_simpleForm;
u	union	NULL
otherInfo	Z_OtherInformation	NULL

Table 5.14: Default settings for Access Control Request

Field	Type	Default Value
referenceId	Z_ReferenceId	NULL
which	enum	Z_AccessResponse_simpleForm
u	union	NULL
diagnostic	Z_DiagRec	NULL
otherInfo	Z_OtherInformation	NULL

Table 5.15: Default settings for Access Control Response

Field	Type	Default Value
referenceId	Z_ReferenceId	NULL
numberOfRecordsReturned	Odr_int	value=0
num_segmentRecords	Odr_int	0
segmentRecords	Z_NamePlusRecord	NULL
otherInfo	Z_OtherInformation	NULL

Table 5.16: Default settings for Segment

Field	Type	Default Value
referenceId	Z_ReferenceId	NULL
closeReason	Odr_int	Z_Close_finished
diagnosticInformation	char*	NULL
resourceReportFormat	Odr_oid	NULL
resourceFormat	Z_External	NULL
otherInfo	Z_OtherInformation	NULL

Table 5.17: Default settings for Close

Chapter 6

SOAP and SRU

Introduction

YAZ uses a very simple implementation of SOAP that only (currently) supports what is sufficient to offer SRU SOAP functionality. The implementation uses the tree API of libxml2 to encode and decode SOAP packages.

Like the Z39.50 ASN.1 module, the YAZ SRU implementation uses simple C structs to represent SOAP packages as well as HTTP packages.

HTTP

YAZ only offers HTTP as transport carrier for SOAP, but it is relatively easy to change that.

The following definition of `Z_GDU` (Generic Data Unit) allows for both HTTP and Z39.50 in one packet.

```
#include <yaz/zgdu.h>

#define Z_GDU_Z3950         1
#define Z_GDU_HTTP_Request  2
#define Z_GDU_HTTP_Response 3
typedef struct {
  int which;
  union {
    Z_APDU *z3950;
    Z_HTTP_Request *HTTP_Request;
    Z_HTTP_Response *HTTP_Response;
  } u;
} Z_GDU ;
```

The corresponding Z_GDU encoder/decoder is `z_GDU`. The `z3950` is any of the known BER encoded Z39.50 APDUs. `HTTP_Request` and `HTTP_Response` is the HTTP Request and Response respectively.

SOAP Packages

Every SOAP package in YAZ is represented as follows:

```
#include <yaz/soap.h>

typedef struct {
    char *fault_code;
    char *fault_string;
    char *details;
} Z_SOAP_Fault;

typedef struct {
    int no;
    char *ns;
    void *p;
} Z_SOAP_Generic;

#define Z_SOAP_fault 1
#define Z_SOAP_generic 2
#define Z_SOAP_error 3
typedef struct {
    int which;
    union {
        Z_SOAP_Fault   *fault;
        Z_SOAP_Generic *generic;
        Z_SOAP_Fault   *soap_error;
    } u;
    const char *ns;
} Z_SOAP;
```

The `fault` and `soap_error` arms both represent a SOAP fault - struct `Z_SOAP_Fault`. Any other generic (valid) package is represented by `Z_SOAP_Generic`.

The `ns` as part of `Z_SOAP` is the namespace for SOAP itself and reflects the SOAP version. For version 1.1 it is `http://schemas.xmlsoap.org/soap/envelope/`, for version 1.2 it is `http://www.w3.org/2001/06/soap-envelope`.

```
int z_soap_codec(ODR o, Z_SOAP **pp,
                 char **content_buf, int *content_len,
                 Z_SOAP_Handler *handlers);
```

The `content_buf` and `content_len` is XML buffer and length of buffer respectively.

The handlers is a list of SOAP codec handlers - one handler for each service namespace. For SRU SOAP, the namespace would be http://www.loc.gov/zing/srw/v1.0/.

When decoding, the z_soap_codec inspects the XML content and tries to match one of the services namespaces of the supplied handlers. If there is a match. a handler function is invoked which decodes that particular SOAP package. If successful, the returned Z_SOAP package will be of type Z_SOAP_Generic. Member no is set the offset of the handler that matched; ns is set to namespace of the matching handler; the void pointer p is set to the C data structure associated with the handler.

When a NULL namespace is met (member ns below), that specifies end-of-list.

Each handler is defined as follows:

```
typedef struct {
    char *ns;
    void *client_data;
    Z_SOAP_fun f;
} Z_SOAP_Handler;
```

The ns is the namespace of the service associated with handler f. The client_data is user-defined data which is passed to the handler.

The prototype for a SOAP service handler is:

```
int handler(ODR o, void * ptr, void **handler_data,
            void *client_data, const char *ns);
```

The *o* specifies the mode (decode/encode) as usual. The second argument, *ptr*, is a libxml2 tree node pointer (xmlNodePtr) and is a pointer to the Body element of the SOAP package. The *handler_data* is an opaque pointer to C definitions associated with the SOAP service. The *client_data* is the pointer which was set as part of the Z_SOAP_handler. Finally, *ns* is the service namespace.

SRU

SRU SOAP is just one implementation of a SOAP handler as described in the previous section. The encoder/decoder handler for SRU is defined as follows:

```
#include <yaz/srw.h>

int yaz_srw_codec(ODR o, void * pptr,
                  Z_SRW_GDU **handler_data,
                  void *client_data, const char *ns);
```

Here, Z_SRW_GDU is either searchRetrieveRequest or a searchRetrieveResponse.

Note
The xQuery and xSortKeys are not handled yet by the SRW implementation of YAZ. Explain is also missing. Future versions of YAZ will include these features.

The definition of searchRetrieveRequest is:

```
typedef struct {

#define Z_SRW_query_type_cql  1
#define Z_SRW_query_type_xcql 2
#define Z_SRW_query_type_pqf  3
    int query_type;
    union {
        char *cql;
        char *xcql;
        char *pqf;
    } query;

#define Z_SRW_sort_type_none 1
#define Z_SRW_sort_type_sort 2
#define Z_SRW_sort_type_xSort 3
    int sort_type;
    union {
        char *none;
        char *sortKeys;
        char *xSortKeys;
    } sort;
    int  *startRecord;
    int  *maximumRecords;
    char *recordSchema;
    char *recordPacking;
    char *database;
} Z_SRW_searchRetrieveRequest;
```

Please observe that data of type xsd:string is represented as a char pointer (`char *`). A null pointer means that the element is absent. Data of type xsd:integer is represented as a pointer to an int (`int *`). Again, a null pointer is used for absent elements.

The SearchRetrieveResponse has the following definition.

```
typedef struct {
    int * numberOfRecords;
    char * resultSetId;
    int * resultSetIdleTime;
```

```
    Z_SRW_record *records;
    int num_records;

    Z_SRW_diagnostic *diagnostics;
    int num_diagnostics;
    int *nextRecordPosition;
} Z_SRW_searchRetrieveResponse;
```

The `num_records` and `num_diagnostics` is number of returned records and diagnostics respectively, and also correspond to the "size of" arrays `records` and `diagnostics`.

A retrieval record is defined as follows:

```
typedef struct {
    char *recordSchema;
    char *recordData_buf;
    int recordData_len;
    int *recordPosition;
} Z_SRW_record;
```

The record data is defined as a buffer of some length so that data can be of any type. SRW 1.0 currently doesn't allow for this (only XML), but future versions might do.

And, a diagnostic as:

```
typedef struct {
    int  *code;
    char *details;
} Z_SRW_diagnostic;
```

Chapter 7

Supporting Tools

In support of the service API - primarily the ASN module, which provides the programmatic interface to the Z39.50 APDUs, YAZ contains a collection of tools that support the development of applications.

Query Syntax Parsers

Since the type-1 (RPN) query structure has no direct, useful string representation, every origin application needs to provide some form of mapping from a local query notation or representation to a Z_RPNQuery structure. Some programmers will prefer to construct the query manually, perhaps using `odr_malloc()` to simplify memory management. The YAZ distribution includes three separate, query-generating tools that may be of use to you.

Prefix Query Format

Since RPN or reverse polish notation is really just a fancy way of describing a suffix notation format (operator follows operands), it would seem that the confusion is total when we now introduce a prefix notation for RPN. The reason is one of simple laziness - it's somewhat simpler to interpret a prefix format, and this utility was designed for maximum simplicity, to provide a baseline representation for use in simple test applications and scripting environments (like Tcl). The demonstration client included with YAZ uses the PQF.

Note
The PQF has been adopted by other parties developing Z39.50 software. It is often referred to as Prefix Query Notation - PQN.

The PQF is defined by the pquery module in the YAZ library. There are two sets of functions that have similar behavior. First set operates on a PQF parser handle, second set doesn't. First set of functions are more flexible than the second set. Second set is obsolete and is only provided to ensure backwards compatibility.

First set of functions all operate on a PQF parser handle:

```
#include <yaz/pquery.h>

YAZ_PQF_Parser yaz_pqf_create(void);

void yaz_pqf_destroy(YAZ_PQF_Parser p);

Z_RPNQuery *yaz_pqf_parse(YAZ_PQF_Parser p, ODR o, const char *qbuf);

Z_AttributesPlusTerm *yaz_pqf_scan(YAZ_PQF_Parser p, ODR o,
                      Odr_oid **attributeSetId, const char *qbuf);

int yaz_pqf_error(YAZ_PQF_Parser p, const char **msg, size_t *off);
```

A PQF parser is created and destructed by functions `yaz_pqf_create` and `yaz_pqf_destroy` respectively. Function `yaz_pqf_parse` parses the query given by string `qbuf`. If parsing was successful, a Z39.50 RPN Query is returned which is created using ODR stream `o`. If parsing failed, a NULL pointer is returned. Function `yaz_pqf_scan` takes a scan query in `qbuf`. If parsing was successful, the function returns attributes plus term pointer and modifies `attributeSetId` to hold attribute set for the scan request - both allocated using ODR stream `o`. If parsing failed, yaz_pqf_scan returns a NULL pointer. Error information for bad queries can be obtained by a call to `yaz_pqf_error` which returns an error code and modifies `*msg` to point to an error description, and modifies `*off` to the offset within the last query where parsing failed.

The second set of functions are declared as follows:

```
#include <yaz/pquery.h>

Z_RPNQuery *p_query_rpn(ODR o, oid_proto proto, const char *qbuf);

Z_AttributesPlusTerm *p_query_scan(ODR o, oid_proto proto,
                         Odr_oid **attributeSetP, const char *qbuf);

int p_query_attset(const char *arg);
```

The function `p_query_rpn()` takes as arguments an ODR stream (see section The ODR Module) to provide a memory source (the structure created is released on the next call to `odr_reset()` on the stream), a protocol identifier (one of the constants PROTO_Z3950 and PROTO_SR), an attribute set reference, and finally a null-terminated string holding the query string.

If the parse went well, `p_query_rpn()` returns a pointer to a `Z_RPNQuery` structure which can be placed directly into a `Z_SearchRequest`. If parsing failed, due to syntax error, a NULL pointer is returned.

The `p_query_attset` specifies which attribute set to use if the query doesn't specify one by the `@attrset` operator. The `p_query_attset` returns 0 if the argument is a valid attribute set specifier; otherwise the function returns -1.

The grammar of the PQF is as follows:

```
query ::= top-set query-struct.

top-set ::= [ '@attrset' string ]

query-struct ::= attr-spec | simple | complex | '@term' term- ↩
   type query

attr-spec ::= '@attr' [ string ] string query-struct

complex ::= operator query-struct query-struct.

operator ::= '@and' | '@or' | '@not' | '@prox' proximity.

simple ::= result-set | term.

result-set ::= '@set' string.

term ::= string.

proximity ::= exclusion distance ordered relation which-code  ↩
   unit-code.

exclusion ::= '1' | '0' | 'void'.

distance ::= integer.

ordered ::= '1' | '0'.

relation ::= integer.

which-code ::= 'known' | 'private' | integer.

unit-code ::= integer.

term-type ::= 'general' | 'numeric' | 'string' | 'oid' | ' ↩
   datetime' | 'null'.
```

You will note that the syntax above is a fairly faithful representation of RPN, except for the Attribute, which has been moved a step away from the term, allowing you to associate one or more attributes with an entire query structure. The parser will automatically apply the given attributes to each term as required.

The @attr operator is followed by an attribute specification (`attr-spec` above). The specification consists of an optional attribute set, an attribute type-value pair and a sub-query. The attribute type-value pair is packed in one string: an attribute type, an equals sign, and an attribute value, like this: `@attr 1=1003`. The type is always an integer, but the value may be either an integer or a string (if it doesn't start with a digit character). A string attribute-value is encoded as a Type-1 "complex" attribute with the list of values containing the single string specified, and including no semantic indicators.

Version 3 of the Z39.50 specification defines various encoding of terms. Use `@term type string`, where type is one of: `general`, `numeric` or `string` (for InternationalString). If no term type has been given, the `general` form is used. This is the only encoding allowed in both versions 2 and 3 of the Z39.50 standard.

Using Proximity Operators with PQF

Note

This is an advanced topic, describing how to construct queries that make very specific requirements on the relative location of their operands. You may wish to skip this section and go straight to the example PQF queries.

Warning

Most Z39.50 servers do not support proximity searching, or support only a small subset of the full functionality that can be expressed using the PQF proximity operator. Be aware that the ability to *express* a query in PQF is no guarantee that any given server will be able to *execute* it.

The proximity operator `@prox` is a special and more restrictive version of the conjunction operator `@and`. Its semantics are described in section 3.7.2 (Proximity) of Z39.50 the standard itself, which can be read on-line at http://www.loc.gov/z3950/agency/markup/09.html#3.7.2

In PQF, the proximity operation is represented by a sequence of the form

```
@prox exclusion distance ordered relation which-code unit-code
```

in which the meanings of the parameters are as described in the standard, and they can take the following values:

- **exclusion** 0 = false (i.e. the proximity condition specified by the remaining parameters must be satisfied) or 1 = true (the proximity condition specified by the remaining parameters must *not* be satisfied).
- **distance** An integer specifying the difference between the locations of the operands: e.g. two adjacent words would have distance=1 since their locations differ by one unit.
- **ordered** 1 = ordered (the operands must occur in the order the query specifies them) or 0 = unordered (they may appear in either order).
- **relation** Recognised values are 1 (lessThan), 2 (lessThanOrEqual), 3 (equal), 4 (greaterThanOrEqual), 5 (greaterThan) and 6 (notEqual).
- **which-code** `known` or `k` (the unit-code parameter is taken from the well-known list of alternatives described below) or `private` or `p` (the unit-code parameter has semantics specific to an out-of-band agreement such as a profile).
- **unit-code** If the which-code parameter is `known` then the recognised values are 1 (character), 2 (word), 3 (sentence), 4 (paragraph), 5 (section), 6 (chapter), 7 (document), 8 (element), 9 (subelement), 10 (elementType) and 11 (byte). If which-code is `private` then the acceptable values are determined by the profile.

(The numeric values of the relation and well-known unit-code parameters are taken straight from the ASN.1 of the proximity structure in the standard.)

PQF queries

Example 7.1 PQF queries using simple terms

```
dylan

"bob dylan"
```

Example 7.2 PQF boolean operators

```
@or "dylan" "zimmerman"

@and @or dylan zimmerman when

@and when @or dylan zimmerman
```

Example 7.3 PQF references to result sets

```
@set Result-1

@and @set seta @set setb
```

Example 7.4 Attributes for terms

```
@attr 1=4 computer

@attr 1=4 @attr 4=1 "self portrait"

@attrset exp1 @attr 1=1 CategoryList

@attr gils 1=2008 Copenhagen

@attr 1=/book/title computer
```

Example 7.5 PQF Proximity queries

```
@prox 0 3 1 2 k 2 dylan zimmerman
```

Here the parameters 0, 3, 1, 2, k and 2 represent exclusion, distance, ordered, relation, which-code and unit-code, in that order. So:

- exclusion = 0: the proximity condition must hold
- distance = 3: the terms must be three units apart
- ordered = 1: they must occur in the order they are specified

- relation = 2: lessThanOrEqual (to the distance of 3 units)
- which-code is "known", so the standard unit-codes are used
- unit-code = 2: word.

So the whole proximity query means that the words `dylan` and `zimmerman` must both occur in the record, in that order, differing in position by three or fewer words (i.e. with two or fewer words between them.) The query would find "Bob Dylan, aka. Robert Zimmerman", but not "Bob Dylan, born as Robert Zimmerman" since the distance in this case is four.

Example 7.6 PQF specification of search term type

```
@term string "a UTF-8 string, maybe?"
```

Example 7.7 PQF mixed queries

```
@or @and bob dylan @set Result-1

@attr 4=1 @and @attr 1=1 "bob dylan" @attr 1=4 "slow train coming"

@and @attr 2=4 @attr gils 1=2038 -114 @attr 2=2 @attr gils 1=2039 -109
```

The last of these examples is a spatial search: in the GILS attribute set, access point 2038 indicates West Bounding Coordinate and 2030 indicates East Bounding Coordinate, so the query is for areas extending from -114 degrees longitude to no more than -109 degrees longitude.

CCL

Not all users enjoy typing in prefix query structures and numerical attribute values, even in a minimalistic test client. In the library world, the more intuitive Common Command Language - CCL (ISO 8777) has enjoyed some popularity - especially before the widespread availability of graphical interfaces. It is still useful in applications where you for some reason or other need to provide a symbolic language for expressing boolean query structures.

CCL Syntax

The CCL parser obeys the following grammar for the FIND argument. The syntax is annotated using lines prefixed by `--`.

```
CCL-Find ::= CCL-Find Op Elements
           | Elements.

Op ::= "and" | "or" | "not"
-- The above means that Elements are separated by boolean operators.

Elements ::= '(' CCL-Find ')'
           | Set
           | Terms
           | Qualifiers Relation Terms
```

```
            | Qualifiers Relation '(' CCL-Find ')'
            | Qualifiers '=' string '-' string
-- Elements is either a recursive definition, a result set reference, ↩
    a
-- list of terms, qualifiers followed by terms, qualifiers followed
-- by a recursive definition or qualifiers in a range (lower - upper) ↩
   .

Set ::= 'set' = string
-- Reference to a result set

Terms ::= Terms Prox Term
       | Term
-- Proximity of terms.

Term ::= Term string
      | string
-- This basically means that a term may include a blank

Qualifiers ::= Qualifiers ',' string
            | string
-- Qualifiers is a list of strings separated by comma

Relation ::= '=' | '>=' | '<=' | '<>' | '>' | '<'
-- Relational operators. This really doesn't follow the ISO8777
-- standard.

Prox ::= '%' | '!'
-- Proximity operator
```

Example 7.8 CCL queries

The following queries are all valid:

```
dylan

"bob dylan"

dylan or zimmerman

set=1

(dylan and bob) or set=1

righttrunc?

"notrunc?"

singlechar#mask
```

Assuming that the qualifiers `ti` and `au` and `date` are defined, we may use:

```
ti=self portrait

au=(bob dylan and slow train coming)

date>1980 and (ti=((self portrait)))
```

CCL Qualifiers

Qualifiers are used to direct the search to a particular searchable index, such as title (ti) and author indexes (au). The CCL standard itself doesn't specify a particular set of qualifiers, but it does suggest a few shorthand notations. You can customize the CCL parser to support a particular set of qualifiers to reflect the current target profile. Traditionally, a qualifier would map to a particular use-attribute within the BIB-1 attribute set. It is also possible to set other attributes, such as the structure attribute.

A CCL profile is a set of predefined CCL qualifiers that may be read from a file or set in the CCL API. The YAZ client reads its CCL qualifiers from a file named `default.bib`. There are four types of lines in a CCL profile: qualifier specification, qualifier alias, comments and directives.

Qualifier specification

A qualifier specification is of the form:

qualifier-name [*attributeset*,]*type*=*val* [*attributeset*,]*type*=*val* ...

where *qualifier-name* is the name of the qualifier to be used (e.g. `ti`), *type* is attribute type in the attribute set (Bib-1 is used if no attribute set is given) and *val* is attribute value. The *type* can be specified as an integer, or as a single-letter: `u` for use, `r` for relation, `p` for position, `s` for structure,`t` for truncation, or `c` for completeness. The attributes for the special qualifier name `term` are used when no CCL qualifier is given in a query.

Refer to Bib-1 Attribute Set(7) or the complete list of Bib-1 attributes

It is also possible to specify non-numeric attribute values, which are used in combination with certain types. The special combinations are:

Example 7.9 CCL profile

Consider the following definition:

```
ti        u=4 s=1
au        u=1 s=1
term      s=105
ranked    r=102
date      u=30 r=o
```

`ti` and `au` both set structure attribute to phrase (s=1). `ti` sets the use-attribute to 4. `au` sets the use-attribute to 1. When no qualifiers are used in the query, the structure-attribute is set to free-form-text (105) (rule for `term`). The `date` sets the relation attribute to the relation used in the CCL query and sets the use attribute to 30 (Bib-1 Date).
You can combine attributes. To Search for "ranked title" you can do

Type	Description
u=*value*	Use attribute (1). Common use attributes are 1 Personal-name, 4 Title, 7 ISBN, 8 ISSN, 30 Date, 62 Subject, 1003 Author, 1016 Any. Specify value as an integer.
r=*value*	Relation attribute (2). Common values are 1 <, 2 <=, 3 =, 4 >=, 5 >, 6 <>, 100 phonetic, 101 stem, 102 relevance, 103 always matches.
p=*value*	Position attribute (3). Values: 1 first in field, 2 first in any subfield, 3 any position in field.
s=*value*	Structure attribute (4). Values: 1 phrase, 2 word, 3 key, 4 year, 5 date, 6 word list, 100 date (un), 101 name (norm), 102 name (un), 103 structure, 104 urx, 105 free-form-text, 106 document-text, 107 local-number, 108 string, 109 numeric string.
t=*value*	Truncation attribute (5). Values: 1 right, 2 left, 3 left and right, 100 none, 101 process #, 102 regular-1, 103 regular-2, 104 CCL.
c=*value*	Completeness attribute (6). Values: 1 incomplete subfield, 2 complete subfield, 3 complete field.

Table 7.1: Common Bib-1 attributes

```
ti,ranked=knuth computer
```

which will set relation=ranked, use=title, structure=phrase.
Query

```
date > 1980
```

is a valid query. But

```
ti > 1980
```

is invalid.

Qualifier alias

A qualifier alias is of the form:

q q1 q2 ..

which declares *q* to be an alias for *q1*, *q2*... such that the CCL query *q=x* is equivalent to *q1=x or q2=x or ...*.

Comments

Lines with white space or lines that begin with character # are treated as comments.

Directives

Directive specifications takes the form

@directive value

Name	Description
s=pw	The structure is set to either word or phrase depending on the number of tokens in a term (phrase-word).
s=al	Each token in the term is ANDed (and-list). This does not set the structure at all.
s=ol	Each token in the term is ORed (or-list). This does not set the structure at all.
s=ag	Tokens that appears as phrases (with blank in them) gets structure phrase attached (4=1). Tokens that appear to be words gets structure word attached (4=2). Phrases and words are ANDed. This is a variant of s=al and s=pw, with the main difference that words are not split (with operator AND) but instead kept in one RPN token. This facility appeared in YAZ 4.2.38.
s=sl	Tokens are split into sub-phrases of all combinations - in order. This facility appeared in YAZ 5.14.0.
r=o	Allows ranges and the operators greater-than, less-than, ... equals. This sets Bib-1 relation attribute accordingly (relation ordered). A query construct is only treated as a range if dash is used and that is surrounded by white-space. So -1980 is treated as term "-1980" not <=1980. If -1980 is used, however, that is treated as a range.
r=r	Similar to r=o but assumes that terms are non-negative (not prefixed with -). Thus, a dash will always be treated as a range. The construct 1980-1990 is treated as a range with r=r but as a single term "1980-1990" with r=o. The special attribute r=r is available in YAZ 2.0.24 or later.
r=omiteq	This will omit relation=equals (@attr 2=3) when r=o / r=r is used. This is useful for servers that somehow break when an explicit relation=equals is used. Omitting the relation is usually safe because "equals" is the default behavior. This tweak was added in YAZ version 5.1.2.
t=l	Allows term to be left-truncated. If term is of the form ?x, the resulting Type-1 term is x and truncation is left.
t=r	Allows term to be right-truncated. If term is of the form x?, the resulting Type-1 term is x and truncation is right.
t=n	If term is does not include ?, the truncation attribute is set to none (100).
t=b	Allows term to be both left-and-right truncated. If term is of the form ?x?, the resulting term is x and truncation is set to both left and right.
t=x	Allows masking anywhere in a term, thus fully supporting # (mask one character) and ? (zero or more of any). If masking is used, truncation is set to 102 (regexp-1 in term) and the term is converted accordingly to a regular expression.
t=z	Allows masking anywhere in a term, thus fully supporting # (mask one character) and ? (zero or more of any). If masking is used, truncation is set to 104 (Z39.58 in term) and the term is converted accordingly to Z39.58 masking term - actually the same truncation as CCL itself.

Table 7.2: Special attribute combos

Name	Description	Default
truncation	Truncation character	?
mask	Masking character. Requires YAZ 4.2.58 or later	#
field	Specifies how multiple fields are to be combined. There are two modes: or: multiple qualifier fields are ORed, merge: attributes for the qualifier fields are merged and assigned to one term.	merge
case	Specifies if CCL operators and qualifiers should be compared with case sensitivity or not. Specify 1 for case sensitive; 0 for case insensitive.	1
and	Specifies token for CCL operator AND.	and
or	Specifies token for CCL operator OR.	or
not	Specifies token for CCL operator NOT.	not
set	Specifies token for CCL operator SET.	set

Table 7.3: CCL directives

CCL API

All public definitions can be found in the header file ccl.h. A profile identifier is of type CCL_bibset. A profile must be created with the call to the function ccl_qual_mk which returns a profile handle of type CCL_bibset.

To read a file containing qualifier definitions the function ccl_qual_file may be convenient. This function takes an already opened FILE handle pointer as argument along with a CCL_bibset handle.

To parse a simple string with a FIND query use the function

```
struct ccl_rpn_node *ccl_find_str(CCL_bibset bibset, const char *str,
                                  int *error, int *pos);
```

which takes the CCL profile (bibset) and query (str) as input. Upon successful completion the RPN tree is returned. If an error occurs, such as a syntax error, the integer pointed to by error holds the error code and pos holds the offset inside query string in which the parsing failed.

An English representation of the error may be obtained by calling the ccl_err_msg function. The error codes are listed in ccl.h.

To convert the CCL RPN tree (type struct ccl_rpn_node *) to the Z_RPNQuery of YAZ the function ccl_rpn_query must be used. This function which is part of YAZ is implemented in yaz-ccl.c. After calling this function the CCL RPN tree is probably no longer needed. The ccl_rpn_delete destroys the CCL RPN tree.

A CCL profile may be destroyed by calling the ccl_qual_rm function.

The token names for the CCL operators may be changed by setting the globals (all type char *) ccl_token_and, ccl_token_or, ccl_token_not and ccl_token_set. An operator may have aliases, i.e. there may be more than one name for the operator. To do this, separate each alias with a space character.

CQL

CQL - Common Query Language - was defined for the SRU protocol. In many ways CQL has a similar

syntax to CCL. The objective of CQL is different. Where CCL aims to be an end-user language, CQL is *the* protocol query language for SRU.

Tip
If you are new to CQL, read the Gentle Introduction.

The CQL parser in YAZ provides the following:

- It parses and validates a CQL query.
- It generates a C structure that allows you to convert a CQL query to some other query language, such as SQL.
- The parser converts a valid CQL query to PQF, thus providing a way to use CQL for both SRU servers and Z39.50 targets at the same time.
- The parser converts CQL to XCQL. XCQL is an XML representation of CQL. XCQL is part of the SRU specification. However, since SRU supports CQL only, we don't expect XCQL to be widely used. Furthermore, CQL has the advantage over XCQL that it is easy to read.

CQL parsing

A CQL parser is represented by the `CQL_parser` handle. Its contents should be considered YAZ internal (private).

```
#include <yaz/cql.h>

typedef struct cql_parser *CQL_parser;

CQL_parser cql_parser_create(void);
void cql_parser_destroy(CQL_parser cp);
```

A parser is created by `cql_parser_create` and is destroyed by `cql_parser_destroy`.

To parse a CQL query string, the following function is provided:

```
int cql_parser_string(CQL_parser cp, const char *str);
```

A CQL query is parsed by the `cql_parser_string` which takes a query *`str`*. If the query was valid (no syntax errors), then zero is returned; otherwise -1 is returned to indicate a syntax error.

```
int cql_parser_stream(CQL_parser cp,
                      int (*getbyte)(void *client_data),
                      void (*ungetbyte)(int b, void *client_data),
                      void *client_data);

int cql_parser_stdio(CQL_parser cp, FILE *f);
```

The functions `cql_parser_stream` and `cql_parser_stdio` parse a CQL query - just like `cql_parser_string`. The only difference is that the CQL query can be fed to the parser in different ways. The `cql_parser_stream` uses a generic byte stream as input. The `cql_parser_stdio` uses a `FILE` handle which is opened for reading.

CQL tree

If the query string is valid, the CQL parser generates a tree representing the structure of the CQL query.

```
struct cql_node *cql_parser_result(CQL_parser cp);
```

`cql_parser_result` returns a pointer to the root node of the resulting tree.

Each node in a CQL tree is represented by a `struct cql_node`. It is defined as follows:

```
#define CQL_NODE_ST 1
#define CQL_NODE_BOOL 2
#define CQL_NODE_SORT 3
struct cql_node {
    int which;
    union {
        struct {
            char *index;
    char *index_uri;
            char *term;
            char *relation;
    char *relation_uri;
            struct cql_node *modifiers;
        } st;
        struct {
            char *value;
            struct cql_node *left;
            struct cql_node *right;
            struct cql_node *modifiers;
        } boolean;
        struct {
            char *index;
```

```
            struct cql_node *next;
            struct cql_node *modifiers;
            struct cql_node *search;
        } sort;
    } u;
};
```

There are three node types: search term (ST), boolean (BOOL) and sortby (SORT). A modifier is treated as a search term too.

The search term node has five members:

- `index`: index for search term. If an index is unspecified for a search term, `index` will be NULL.
- `index_uri`: index URI for search term or NULL if none could be resolved for the index.
- `term`: the search term itself.
- `relation`: relation for search term.
- `relation_uri`: relation URI for search term.
- `modifiers`: relation modifiers for search term. The `modifiers` list itself of cql_nodes each of type `ST`.

The boolean node represents `and`, `or`, `not` + proximity.

- `left` and `right`: left - and right operand respectively.
- `modifiers`: proximity arguments.

The sort node represents both the SORTBY clause.

CQL to PQF conversion

Conversion to PQF (and Z39.50 RPN) is tricky by the fact that the resulting RPN depends on the Z39.50 target capabilities (combinations of supported attributes). In addition, the CQL and SRU operates on index prefixes (URI or strings), whereas the RPN uses Object Identifiers for attribute sets.

The CQL library of YAZ defines a `cql_transform_t` type. It represents a particular mapping between CQL and RPN. This handle is created and destroyed by the functions:

```
cql_transform_t cql_transform_open_FILE (FILE *f);
cql_transform_t cql_transform_open_fname(const char *fname);
void cql_transform_close(cql_transform_t ct);
```

The first two functions create a transformation handle from either an already open FILE or from a filename respectively.

The handle is destroyed by `cql_transform_close` in which case no further reference of the handle is allowed.

When a `cql_transform_t` handle has been created you can convert to RPN.

```
int cql_transform_buf(cql_transform_t ct,
                      struct cql_node *cn, char *out, int max);
```

This function converts the CQL tree `cn` using handle `ct`. For the resulting PQF, you supply a buffer `out` which must be able to hold at at least `max` characters.

If conversion failed, `cql_transform_buf` returns a non-zero SRU error code; otherwise zero is returned (conversion successful). The meanings of the numeric error codes are listed in the SRU specification somewhere (no direct link anymore).

If conversion fails, more information can be obtained by calling

```
int cql_transform_error(cql_transform_t ct, char **addinfop);
```

This function returns the most recently returned numeric error-code and sets the string-pointer at `*addinfop` to point to a string containing additional information about the error that occurred: for example, if the error code is 15 ("Illegal or unsupported context set"), the additional information is the name of the requested context set that was not recognised.

The SRU error-codes may be translated into brief human-readable error messages using

```
const char *cql_strerror(int code);
```

If you wish to be able to produce a PQF result in a different way, there are two alternatives.

```
void cql_transform_pr(cql_transform_t ct,
                      struct cql_node *cn,
                      void (*pr)(const char *buf, void *client_data),
                      void *client_data);

int cql_transform_FILE(cql_transform_t ct,
                       struct cql_node *cn, FILE *f);
```

The former function produces output to a user-defined output stream. The latter writes the result to an already open `FILE`.

Specification of CQL to RPN mappings

The file supplied to functions `cql_transform_open_FILE`, `cql_transform_open_fname` follows a structure found in many Unix utilities. It consists of mapping specifications - one per line. Lines starting with # are ignored (comments).

Each line is of the form

CQL pattern = RPN equivalent

An RPN pattern is a simple attribute list. Each attribute pair takes the form:

[set] type=value

The attribute *set* is optional. The *type* is the attribute type, *value* the attribute value.

The character * (asterisk) has special meaning when used in the RPN pattern. Each occurrence of * is substituted with the CQL matching name (index, relation, qualifier etc). This facility can be used to copy a CQL name verbatim to the RPN result.

The following CQL patterns are recognized:

index.*set*.*name* This pattern is invoked when a CQL index, such as dc.title is converted. *set* and *name* are the context set and index name respectively. Typically, the RPN specifies an equivalent use attribute.

For terms not bound by an index, the pattern `index.cql.serverChoice` is used. Here, the prefix `cql` is defined as `http://www.loc.gov/zing/cql/cql-indexes/v1.0/`. If this pattern is not defined, the mapping will fail.

The pattern, `index.`*set*`.*` is used when no other index pattern is matched.

qualifier.*set*.*name* (DEPRECATED) For backwards compatibility, this is recognised as a synonym of `index.`*set*`.`*name*

relation.*relation* This pattern specifies how a CQL relation is mapped to RPN. The *pattern* is name of relation operator. Since = is used as separator between CQL pattern and RPN, CQL relations including = cannot be used directly. To avoid a conflict, the names `ge`, `eq`, `le`, must be used for CQL operators, greater-than-or-equal, equal, less-than-or-equal respectively. The RPN pattern is supposed to include a relation attribute.

For terms not bound by a relation, the pattern `relation.scr` is used. If the pattern is not defined, the mapping will fail.

The special pattern, `relation.*` is used when no other relation pattern is matched.

relationModifier.*mod* This pattern specifies how a CQL relation modifier is mapped to RPN. The RPN pattern is usually a relation attribute.

structure.*type* This pattern specifies how a CQL structure is mapped to RPN. Note that this CQL pattern is somewhat similar to CQL pattern `relation`. The *type* is a CQL relation.

The pattern, `structure.*` is used when no other structure pattern is matched. Usually, the RPN equivalent specifies a structure attribute.

position.*type* This pattern specifies how the anchor (position) of CQL is mapped to RPN. The *type* is one of `first`, `any`, `last`, `firstAndLast`.

The pattern, `position.*` is used when no other position pattern is matched.

set.*prefix* This specification defines a CQL context set for a given prefix. The value on the right hand side is the URI for the set - *not* RPN. All prefixes used in index patterns must be defined this way.

set This specification defines a default CQL context set for index names. The value on the right hand side is the URI for the set.

Example 7.10 CQL to RPN mapping file

This simple file defines two context sets, three indexes and three relations, a position pattern and a default structure.

```
set.cql  = http://www.loc.gov/zing/cql/context-sets/cql/v1.1/
set.dc   = http://www.loc.gov/zing/cql/dc-indexes/v1.0/

index.cql.serverChoice = 1=1016
index.dc.title         = 1=4
index.dc.subject       = 1=21

relation.<             = 2=1
relation.eq            = 2=3
relation.scr           = 2=3

position.any           = 3=3 6=1

structure.*            = 4=1
```

With the mappings above, the CQL query

```
computer
```

is converted to the PQF:

```
@attr 1=1016 @attr 2=3 @attr 4=1 @attr 3=3 @attr 6=1 "computer"
```

by rules `index.cql.serverChoice`, `relation.scr`, `structure.*`, `position.any`.
CQL query

```
computer^
```

is rejected, since `position.right` is undefined.
CQL query

```
>my = "http://www.loc.gov/zing/cql/dc-indexes/v1.0/" my.title = x
```

is converted to

```
@attr 1=4 @attr 2=3 @attr 4=1 @attr 3=3 @attr 6=1 "x"
```

Example 7.11 CQL to RPN string attributes

In this example we allow any index to be passed to RPN as a use attribute.

```
# Identifiers for prefixes used in this file. (index.*)
set.cql  = info:srw/cql-context-set/1/cql-v1.1
set.rpn  = http://bogus/rpn
set      = http://bogus/rpn

# The default index when none is specified by the query
index.cql.serverChoice     = 1=any

index.rpn.*                = 1=*
relation.eq                = 2=3
structure.*                = 4=1
position.any               = 3=3
```

The `http://bogus/rpn` context set is also the default so we can make queries such as

```
title = a
```

which is converted to

```
@attr 2=3 @attr 4=1 @attr 3=3 @attr 1=title "a"
```

Example 7.12 CQL to RPN using Bath Profile

The file `etc/pqf.properties` has mappings from the Bath Profile and Dublin Core to RPN. If YAZ is installed as a package it's usually located in `/usr/share/yaz/etc` and part of the development package, such as `libyaz-dev`.

CQL to XCQL conversion

Conversion from CQL to XCQL is trivial and does not require a mapping to be defined. There are three functions to choose from depending on the way you wish to store the resulting output (XML buffer containing XCQL).

```
int cql_to_xml_buf(struct cql_node *cn, char *out, int max);
void cql_to_xml(struct cql_node *cn,
                void (*pr)(const char *buf, void *client_data),
                void *client_data);
void cql_to_xml_stdio(struct cql_node *cn, FILE *f);
```

Function `cql_to_xml_buf` converts to XCQL and stores the result in a user-supplied buffer of a given max size.

`cql_to_xml` writes the result in a user-defined output stream. `cql_to_xml_stdio` writes to a a file.

PQF to CQL conversion

Conversion from PQF to CQL is offered by the two functions shown below. The former uses a generic stream for result. The latter puts result in a WRBUF (string container).

```
#include <yaz/rpn2cql.h>

int cql_transform_rpn2cql_stream(cql_transform_t ct,
                                 void (*pr)(const char *buf, void *client_d
                                 void *client_data,
                                 Z_RPNQuery *q);

int cql_transform_rpn2cql_wrbuf(cql_transform_t ct,
                                WRBUF w,
                                Z_RPNQuery *q);
```

The configuration is the same as used in CQL to PQF conversions.

Object Identifiers

The basic YAZ representation of an OID is an array of integers, terminated with the value -1. This integer is of type `Odr_oid`.

Fundamental OID operations and the type `Odr_oid` are defined in `yaz/oid_util.h`.

An OID can either be declared as a automatic variable or it can be allocated using the memory utilities or ODR/NMEM. It's guaranteed that an OID can fit in `OID_SIZE` integers.

Example 7.13 Create OID on stack

We can create an OID for the Bib-1 attribute set with:

```
Odr_oid bib1[OID_SIZE];
bib1[0] = 1;
bib1[1] = 2;
bib1[2] = 840;
bib1[3] = 10003;
bib1[4] = 3;
bib1[5] = 1;
bib1[6] = -1;
```

And OID may also be filled from a string-based representation using dots (.). This is achieved by the function

```
int oid_dotstring_to_oid(const char *name, Odr_oid *oid);
```

This functions returns 0 if name could be converted; -1 otherwise.

Example 7.14 Using oid_oiddotstring_to_oid

We can fill the Bib-1 attribute set OID more easily with:

```
Odr_oid bib1[OID_SIZE];
oid_oiddotstring_to_oid("1.2.840.10003.3.1", bib1);
```

We can also allocate an OID dynamically on an ODR stream with:

```
Odr_oid *odr_getoidbystr(ODR o, const char *str);
```

This creates an OID from a string-based representation using dots. This function take an ODR stream as parameter. This stream is used to allocate memory for the data elements, which is released on a subsequent call to `odr_reset()` on that stream.

Example 7.15 Using odr_getoidbystr

We can create an OID for the Bib-1 attribute set with:

```
Odr_oid *bib1 = odr_getoidbystr(odr, "1.2.840.10003.3.1");
```

The function

```
char *oid_oid_to_dotstring(const Odr_oid *oid, char *oidbuf)
```

does the reverse of `oid_oiddotstring_to_oid`. It converts an OID to the string-based representation using dots. The supplied char buffer `oidbuf` holds the resulting string and must be at least `OID_STR_MAX` in size.

OIDs can be copied with `oid_oidcpy` which takes two OID lists as arguments. Alternatively, an OID copy can be allocated on an ODR stream with:

```
Odr_oid *odr_oiddup(ODR odr, const Odr_oid *o);
```

OIDs can be compared with `oid_oidcmp` which returns zero if the two OIDs provided are identical; non-zero otherwise.

OID database

From YAZ version 3 and later, the oident system has been replaced by an OID database. OID database is a misnomer .. the old odient system was also a database.

The OID database is really just a map between named Object Identifiers (string) and their OID raw equivalents. Most operations either convert from string to OID or other way around.

Unfortunately, whenever we supply a string we must also specify the *OID class*. The class is necessary because some strings correspond to multiple OIDs. An example of such a string is `Bib-1` which may either be an attribute-set or a diagnostic-set.

Applications using the YAZ database should include `yaz/oid_db.h`.

A YAZ database handle is of type `yaz_oid_db_t`. Actually that's a pointer. You need not deal with that. YAZ has a built-in database which can be considered "constant" for most purposes. We can get hold of that by using function `yaz_oid_std`.

All functions with prefix `yaz_string_to_oid` converts from class + string to OID. We have variants of this operation due to different memory allocation strategies.

All functions with prefix `yaz_oid_to_string` converts from OID to string + class.

Example 7.16 Create OID with YAZ DB

We can create an OID for the Bib-1 attribute set on the ODR stream odr with:

```
       Odr_oid *bib1 =
yaz_string_to_oid_odr(yaz_oid_std(), CLASS_ATTSET, "Bib-1", odr);
```

This is more complex than using `odr_getoidbystr`. You would only use `yaz_string_to_oid_odr` when the string (here Bib-1) is supplied by a user or configuration.

Standard OIDs

All the object identifiers in the standard OID database as returned by `yaz_oid_std` can be referenced directly in a program as a constant OID. Each constant OID is prefixed with `yaz_oid_` - followed by OID class (lowercase) - then by OID name (normalized and lowercase).

See Appendix A for list of all object identifiers built into YAZ. These are declared in `yaz/oid_std.h` but are included by `yaz/oid_db.h` as well.

Example 7.17 Use a built-in OID

We can allocate our own OID filled with the constant OID for Bib-1 with:

```
       Odr_oid *bib1 = odr_oiddup(o, yaz_oid_attset_bib1);
```

Nibble Memory

Sometimes when you need to allocate and construct a large, interconnected complex of structures, it can be a bit of a pain to release the associated memory again. For the structures describing the Z39.50 PDUs and related structures, it is convenient to use the memory-management system of the ODR subsystem (see Section 8.2). However, in some circumstances where you might otherwise benefit from using a simple nibble-memory management system, it may be impractical to use `odr_malloc()` and `odr_reset()`. For this purpose, the memory manager which also supports the ODR streams is made available in the NMEM module. The external interface to this module is given in the `nmem.h` file.

The following prototypes are given:

```
NMEM nmem_create(void);
void nmem_destroy(NMEM n);
void *nmem_malloc(NMEM n, size_t size);
void nmem_reset(NMEM n);
size_t nmem_total(NMEM n);
void nmem_init(void);
void nmem_exit(void);
```

The nmem_create() function returns a pointer to a memory control handle, which can be released again by nmem_destroy() when no longer needed. The function nmem_malloc() allocates a block of memory of the requested size. A call to nmem_reset() or nmem_destroy() will release all memory allocated on the handle since it was created (or since the last call to nmem_reset(). The function nmem_total() returns the number of bytes currently allocated on the handle.

The nibble-memory pool is shared amongst threads. POSIX mutexes and WIN32 Critical sections are introduced to keep the module thread safe. Function nmem_init() initializes the nibble-memory library and it is called automatically the first time the YAZ.DLL is loaded. YAZ uses function DllMain to achieve this. You should *not* call nmem_init or nmem_exit unless you're absolute sure what you're doing. Note that in previous YAZ versions you'd have to call nmem_init yourself.

Log

YAZ has evolved a fairly complex log system which should be useful both for debugging YAZ itself, debugging applications that use YAZ, and for production use of those applications.

The log functions are declared in header yaz/log.h and implemented in src/log.c. Due to name clash with syslog and some math utilities the logging interface has been modified as of YAZ 2.0.29. The obsolete interface is still available in header file yaz/log.h. The key points of the interface are:

```
void yaz_log(int level, const char *fmt, ...)
void yaz_log_init(int level, const char *prefix, const char *name);
void yaz_log_init_file(const char *fname);
void yaz_log_init_level(int level);
void yaz_log_init_prefix(const char *prefix);
void yaz_log_time_format(const char *fmt);
void yaz_log_init_max_size(int mx);

int yaz_log_mask_str(const char *str);
int yaz_log_module_level(const char *name);
```

The reason for the whole log module is the yaz_log function. It takes a bitmask indicating the log levels, a printf-like format string, and a variable number of arguments to log.

The log level is a bit mask, that says on which level(s) the log entry should be made, and optionally set some behaviour of the logging. In the most simple cases, it can be one of YLOG_FATAL, YLOG_DEBUG, YLOG_WARN, YLOG_LOG. Those can be combined with bits that modify the way the log entry is written:YLOG_ERRNO, YLOG_NOTIME, YLOG_FLUSH. Most of the rest of the bits are deprecated, and should not be used. Use the dynamic log levels instead.

Applications that use YAZ, should not use the LOG_LOG for ordinary messages, but should make use of the dynamic loglevel system. This consists of two parts, defining the loglevel and checking it.

To define the log levels, the (main) program should pass a string to yaz_log_mask_str to define which log levels are to be logged. This string should be a comma-separated list of log level names, and can contain both hard-coded names and dynamic ones. The log level calculation starts with YLOG_DEFAULT_LEVEL and adds a bit for each word it meets, unless the word starts with a '-', in which case it clears the bit. If the string 'none' is found, all bits are cleared. Typically this string comes from the command-line, often

identified by -v. The yaz_log_mask_str returns a log level that should be passed to yaz_log_init_level for it to take effect.

Each module should check what log bits should be used, by calling yaz_log_module_level with a suitable name for the module. The name is cleared of a preceding path and an extension, if any, so it is quite possible to use __FILE__ for it. If the name has been passed to yaz_log_mask_str, the routine returns a non-zero bitmask, which should then be used in consequent calls to yaz_log. (It can also be tested, so as to avoid unnecessary calls to yaz_log, in time-critical places, or when the log entry would take time to construct.)

Yaz uses the following dynamic log levels: server, session, request, requestdetail for the server functionality. zoom for the zoom client API. ztest for the simple test server. malloc, nmem, odr, eventl for internal debugging of yaz itself. Of course, any program using yaz is welcome to define as many new ones as it needs.

By default the log is written to stderr, but this can be changed by a call to yaz_log_init_file or yaz_log_init. If the log is directed to a file, the file size is checked at every write, and if it exceeds the limit given in yaz_log_init_max_size, the log is rotated. The rotation keeps one old version (with a .1 appended to the name). The size defaults to 1GB. Setting it to zero will disable the rotation feature.

```
   A typical yaz-log looks like this
13:23:14-23/11 yaz-ztest(1) [session] Starting session from tcp:127.0.0.1 ↩
    (pid=30968)
13:23:14-23/11 yaz-ztest(1) [request] Init from 'YAZ' (81) (ver 2.0.28)  ↩
   OK
13:23:17-23/11 yaz-ztest(1) [request] Search Z: @attrset Bib-1 foo  OK:7  ↩
   hits
13:23:22-23/11 yaz-ztest(1) [request] Present: [1] 2+2  OK 2 records  ↩
   returned
13:24:13-23/11 yaz-ztest(1) [request] Close OK
```

The log entries start with a time stamp. This can be omitted by setting the YLOG_NOTIME bit in the loglevel. This way automatic tests can be hoped to produce identical log files, that are easy to diff. The format of the time stamp can be set with yaz_log_time_format, which takes a format string just like strftime.

Next in a log line comes the prefix, often the name of the program. For yaz-based servers, it can also contain the session number. Then comes one or more logbits in square brackets, depending on the logging level set by yaz_log_init_level and the loglevel passed to yaz_log_init_level. Finally comes the format string and additional values passed to yaz_log

The log level YLOG_LOGLVL, enabled by the string loglevel, will log all the log-level affecting operations. This can come in handy if you need to know what other log levels would be useful. Grep the logfile for [loglevel].

The log system is almost independent of the rest of YAZ, the only important dependence is of nmem, and that only for using the semaphore definition there.

The dynamic log levels and log rotation were introduced in YAZ 2.0.28. At the same time, the log bit names were changed from LOG_something to YLOG_something, to avoid collision with syslog.h.

MARC

YAZ provides a fast utility for working with MARC records. Early versions of the MARC utility only allowed decoding of ISO2709. Today the utility may both encode - and decode to a variety of formats.

```
#include <yaz/marcdisp.h>

/* create handler */
yaz_marc_t yaz_marc_create(void);
/* destroy */
void yaz_marc_destroy(yaz_marc_t mt);

/* set XML mode YAZ_MARC_LINE, YAZ_MARC_SIMPLEXML, ... */
void yaz_marc_xml(yaz_marc_t mt, int xmlmode);
#define YAZ_MARC_LINE      0
#define YAZ_MARC_SIMPLEXML 1
#define YAZ_MARC_OAIMARC   2
#define YAZ_MARC_MARCXML   3
#define YAZ_MARC_ISO2709   4
#define YAZ_MARC_XCHANGE   5
#define YAZ_MARC_CHECK     6
#define YAZ_MARC_TURBOMARC 7
#define YAZ_MARC_JSON      8

/* supply iconv handle for character set conversion .. */
void yaz_marc_iconv(yaz_marc_t mt, yaz_iconv_t cd);

/* set debug level, 0=none, 1=more, 2=even more, .. */
void yaz_marc_debug(yaz_marc_t mt, int level);

/* decode MARC in buf of size bsize. Returns >0 on success; <=0 on failu
On success, result in *result with size *rsize. */
int yaz_marc_decode_buf(yaz_marc_t mt, const char *buf, int bsize,
                        const char **result, size_t *rsize);

/* decode MARC in buf of size bsize. Returns >0 on success; <=0 on failu
   On success, result in WRBUF */
int yaz_marc_decode_wrbuf(yaz_marc_t mt, const char *buf,
                          int bsize, WRBUF wrbuf);
```

Note
The synopsis is just a basic subset of all functionality. Refer to the actual header file `marcdisp.h` for details.

A MARC conversion handle must be created by using `yaz_marc_create` and destroyed by calling `yaz_marc_destroy`.

All other functions operate on a `yaz_marc_t` handle. The output is specified by a call to `yaz_marc_xml`. The `xmlmode` must be one of

YAZ_MARC_LINE A simple line-by-line format suitable for display but not recommended for further (machine) processing.

YAZ_MARC_MARCXML MARCXML.

YAZ_MARC_ISO2709 ISO2709 (sometimes just referred to as "MARC").

YAZ_MARC_XCHANGE MarcXchange.

YAZ_MARC_CHECK Pseudo format for validation only. Does not generate any real output except diagnostics.

YAZ_MARC_TURBOMARC XML format with same semantics as MARCXML but more compact and geared towards fast processing with XSLT. Refer to Section 7.5.1 for more information.

YAZ_MARC_JSON MARC-in-JSON format.

The actual conversion functions are `yaz_marc_decode_buf` and `yaz_marc_decode_wrbuf` which decodes and encodes a MARC record. The former function operates on simple buffers, and stores the resulting record in a WRBUF handle (WRBUF is a simple string type).

Example 7.18 Display of MARC record

The following program snippet illustrates how the MARC API may be used to convert a MARC record to the line-by-line format:

```
void print_marc(const char *marc_buf, int marc_buf_size)
{
   char *result;      /* for result buf */
   size_t result_len;    /* for size of result */
   yaz_marc_t mt = yaz_marc_create();
   yaz_marc_xml(mt, YAZ_MARC_LINE);
   yaz_marc_decode_buf(mt, marc_buf, marc_buf_size,
                       &result, &result_len);
   fwrite(result, result_len, 1, stdout);
   yaz_marc_destroy(mt);  /* note that result is now freed... */
}
```

TurboMARC

TurboMARC is yet another XML encoding of a MARC record. The format was designed for fast processing with XSLT.

Applications like Pazpar2 uses XSLT to convert an XML encoded MARC record to an internal representation. This conversion mostly checks the tag of a MARC field to determine the basic rules in the conversion. This check is costly when that tag is encoded as an attribute in MARCXML. By having the tag value as the element instead, makes processing many times faster (at least for Libxslt).

TurboMARC is encoded as follows:

- Record elements is part of namespace "`http://www.indexdata.com/turbomarc`".
- A record is enclosed in element `r`.
- A collection of records is enclosed in element `collection`.
- The leader is encoded as element `l` with the leader content as its (text) value.
- A control field is encoded as element `c` concatenated with the tag value of the control field if the tag value matches the regular expression `[a-zA-Z0-9]*`. If the tag value does not match the regular expression `[a-zA-Z0-9]*` the control field is encoded as element `c` and attribute `code` will hold the tag value. This rule ensures that in the rare cases where a tag value might result in a non-well-formed XML, then YAZ will encode it as a coded attribute (as in MARCXML).

  The control field content is the text value of this element. Indicators are encoded as attribute names `i1`, `i2`, etc. and corresponding values for each indicator.
- A data field is encoded as element `d` concatenated with the tag value of the data field or using the attribute `code` as described in the rules for control fields. The children of the data field element are subfield elements. Each subfield element is encoded as `s` concatenated with the sub field code. The text of the subfield element is the contents of the subfield. Indicators are encoded as attributes for the data field element, similar to the encoding for control fields.

Retrieval Facility

YAZ version 2.1.20 or later includes a Retrieval facility tool which allows a SRU/Z39.50 to describe itself and perform record conversions. The idea is the following:

- An SRU/Z39.50 client sends a retrieval request which includes a combination of the following parameters: syntax (format), schema (or element set name).
- The retrieval facility is invoked with parameters in a server/proxy. The retrieval facility matches the parameters a set of "supported" retrieval types. If there is no match, the retrieval signals an error (syntax and / or schema not supported).
- For a successful match, the backend is invoked with the same or altered retrieval parameters (syntax, schema). If a record is received from the backend, it is converted to the frontend name / syntax.
- The resulting record is sent back the client and tagged with the frontend syntax / schema.

The Retrieval facility is driven by an XML configuration. The configuration is neither Z39.50 ZeeRex or SRU ZeeRex. But it should be easy to generate both of them from the XML configuration. (Unfortunately the two versions of ZeeRex differ substantially in this regard.)

Retrieval XML format

All elements should be covered by namespace `http://indexdata.com/yaz` . The root element node must be `retrievalinfo`.

The `retrievalinfo` must include one or more `retrieval` elements. Each `retrieval` defines specific combination of syntax, name and identifier supported by this retrieval service.

The `retrieval` element may include any of the following attributes:

`syntax` (REQUIRED) Defines the record syntax. Possible values is any of the names defined in YAZ' OID database or a raw OID in (n.n ... n).

`name` (OPTIONAL) Defines the name of the retrieval format. This can be any string. For SRU, the value is equivalent to schema (short-hand); for Z39.50 it's equivalent to simple element set name. For YAZ 3.0.24 and later this name may be specified as a glob expression with operators `*` and `?`.

`identifier` (OPTIONAL) Defines the URI schema name of the retrieval format. This can be any string. For SRU, the value is equivalent to URI schema. For Z39.50, there is no equivalent.

The `retrieval` may include one `backend` element. If a `backend` element is given, it specifies how the records are retrieved by some backend and how the records are converted from the backend to the "frontend".

The attributes, `name` and `syntax` may be specified for the `backend` element. The semantics of these attributes is equivalent to those for the `retrieval`. However, these values are passed to the "backend".

The `backend` element may include one or more conversion instructions (as children elements). The supported conversions are:

`marc` The `marc` element specifies a conversion to - and from ISO2709 encoded MARC and MARCXML/MarcXchange. The following attributes may be specified:

`inputformat` (REQUIRED) Format of input. Supported values are `marc` (for ISO2709), `xml` (MARCXML/MarcXchange) and `json` (MARC-in-JSON).

`outputformat` (REQUIRED) Format of output. Supported values are `line` (MARC line format); `marcxml` (for MARCXML), `marc` (ISO2709), `turbomarc`, `marcxchange` (for MarcXchange), or `json` (MARC-in-JSON).

`inputcharset` (OPTIONAL) Encoding of input. For XML input formats, this need not be given, but for ISO2709 based input formats, this should be set to the encoding used. For MARC21 records, a common inputcharset value would be `marc-8`.

`outputcharset` (OPTIONAL) Encoding of output. If outputformat is XML based, it is strongly recommended to use `utf-8`.

`select` The `select` selects one or more text nodes and decodes them as XML. The following attributes may be specified:

`path` (REQUIRED) X-Path expression for selecting text nodes.

This conversion is available in YAZ 5.8.0 and later.

solrmarc The `solrmarc` decodes solrmarc records. It assumes that the input is pure solrmarc text (no escaping) and will convert all sequences of the form #XX; to a single character of the hexadecimal value as given by XX. The output, presumably, is a valid ISO2709 buffer.

This conversion is available in YAZ 5.0.21 and later.

xslt The `xslt` element specifies a conversion via XSLT. The following attributes may be specified:

stylesheet (REQUIRED) Stylesheet file.

In addition, the element can be configured as follows:

param (OPTIONAL) A `param` tag configures a parameter to be passed to the XSLT stylesheet. Multiple `param` tags may be defined.

rdf-lookup The `rdf-lookup` element looks up BIBFRAME elements in some suitable service, for example http://id.loc.gov/authorities/names and replaces the URIs for specified elements with URIs it finds at that service. Its configuration consists of

debug (OPTIONAL) Attribute to the `rdf-lookup` tag to enable debug output. A value of "1" makes the filter to add a XML comment next to each key it tried to look up, showing the URL, the result, and timing. This is useful for debugging the configuration. The default is not to add any comments.

timeout (OPTIONAL) Attribute of the `rdf-lookup` tag which defines timeout in seconds for the HTTP based rdf-lookup.

namespace (OPTIONAL) A `namespace` tag declares a namespace to be used in the `xpath` below. The tag requires two attributes: `prefix` and `href`.

lookup (REQUIRED) A section that defines one tag to be looked up, for example an author.The `xpath` attribute (REQUIRED) specifies the path to the element(s).

key (REQUIRED) A tag withing the `lookup` tag specifies the value to be used in the lookup, for example a name or an ID. It is a relative Xpath starting from the tag specified in the `lookup`.

server (OPTIONAL) Specifies the URL for server to use for the lookup. A `%s` is replaced by the key value to be looked up. If not specified, defaults to the same as the previous `lookup` section, or lacking one, to `http://id.loc.gov/authorities/names/label/%s` . The `method` attribute can be used to specify the HTTP method to be used in this lookup. The default is GET, and the useful alternative is HEAD.

See the example below.

This conversion is available in YAZ 5.19.0 and later.

Retrieval Facility Examples

Example 7.19 MARC21 backend

A typical way to use the retrieval facility is to enable XML for servers that only supports ISO2709 encoded MARC21 records.

```
    <retrievalinfo>
      <retrieval syntax="usmarc" name="F"/>
      <retrieval syntax="usmarc" name="B"/>
      <retrieval syntax="xml" name="marcxml"
     identifier="info:srw/schema/1/marcxml-v1.1">
        <backend syntax="usmarc" name="F">
    <marc inputformat="marc" outputformat="marcxml"
    inputcharset="marc-8"/>
  </backend>
      </retrieval>
      <retrieval syntax="xml" name="dc">
        <backend syntax="usmarc" name="F">
    <marc inputformat="marc" outputformat="marcxml"
    inputcharset="marc-8"/>
          <xslt stylesheet="MARC21slim2DC.xsl"/>
  </backend>
      </retrieval>
    </retrievalinfo>
```

This means that our frontend supports:

- MARC21 F(ull) records.
- MARC21 B(rief) records.
- MARCXML records.
- Dublin core records.

Example 7.20 MARCXML backend

SRW/SRU and Solr backends return records in XML. If they return MARCXML or MarcXchange, the retrieval module can convert those into ISO2709 formats, most commonly USMARC (AKA MARC21). In this example, the backend returns MARCXML for schema="marcxml".

```
    <retrievalinfo>
      <retrieval syntax="usmarc">
        <backend syntax="xml" name="marcxml">
    <marc inputformat="xml" outputformat="marc"
    outputcharset="marc-8"/>
  </backend>
      </retrieval>
      <retrieval syntax="xml" name="marcxml"
     identifier="info:srw/schema/1/marcxml-v1.1"/>
      <retrieval syntax="xml" name="dc">
        <backend syntax="xml" name="marcxml">
```

```
          <xslt stylesheet="MARC21slim2DC.xsl"/>
  </backend>
      </retrieval>
    </retrievalinfo>
```

This means that our frontend supports:

- MARC21 records (any element set name) in MARC-8 encoding.
- MARCXML records for element-set=marcxml
- Dublin core records for element-set=dc.

Example 7.21 RDF-lookup backend

This is a minimal example of the `backend` configuration for the rdf-lookup. It could well be used with some heavy xslt transforms that make BIBFRAME records out of MarxXml.

```
        <backend syntax="xml" name="rdf-lookup">
          <rdf-lookup debug="1" timeout="10">
            <namespace prefix="bf" href="http://id.loc.gov/ontologies/ ↩
               bibframe/" />
            <namespace prefix="bflc" href="http://id.loc.gov/ontologies/ ↩
               bibframe/lc-extensions/"/>
            <lookup xpath="//bf:contribution/bf:Contribution/bf:agent/bf: ↩
               Agent">
              <key field="bflc:name00MatchKey"/>
              <key field="bflc:name01MatchKey"/>
              <key field="bflc:name11MatchKey"/>
              <server url="http://id.loc.gov/authorities/names/label/%s"  ↩
                 method="HEAD"/>
            </lookup>
          </rdf-lookup>
        </backend>
```

The `debug=1` attribute tells the filter to add XML comments to the key nodes that indicate what lookup it tried to do, how it went, and how long it took.
The namespace prefix `bf:` is defined in the `namespace` tags. These namespaces are used in the xpath expressions in the lookup sections.
The `lookup` tag specifies one tag to be looked up. The `xpath` attribute defines which node to modify. It may make use of the `namespace` definitions above.
The `server` tag gives the URL to be used for the lookup. A `%s` in the string will get replaced by the key value. If there is no `server` tag, the one from the preceding `lookup` section is used, and if there is no previous section, the id.loc.gov address is used as a default. The default is to make a GET request, this example uses HEAD

API

It should be easy to use the retrieval systems from applications. Refer to the headers `yaz/retrieval.h` and `yaz/record_conv.h`.

Sorting

This chapter describes sorting and how it is supported in YAZ. Sorting applies to a result-set. The Z39.50 sorting facility takes one or more input result-sets and one result-set as output. The most simple case is that the input-set is the same as the output-set.

Z39.50 sorting has a separate APDU (service) that is, thus, performed following a search (two phases).

In SRU/Solr, however, the model is different. Here, sorting is specified during the search operation. Note, however, that SRU might perform sort as separate search, by referring to an existing result-set in the query (result-set reference).

Using the Z39.50 sort service

yaz-client and the ZOOM API support the Z39.50 sort facility. In any case the sort sequence or sort criteria is using a string notation. This notation is a one-line notation suitable for being manually entered or generated, and allows for easy logging (one liner). For the ZOOM API, the sort is specified in the call to ZOOM_query_sortby function. For yaz-client the sort is performed and specified using the sort and sort+ commands. For description of the sort criteria notation refer to the sort command in the yaz-client manual.

The ZOOM API might choose one of several sort strategies for sorting. Refer to Table 3.2.

Type-7 sort

Type-7 sort is an extension to the Bib-1 based RPN query where the sort specification is embedded as an Attribute-Plus-Term.

The objectives for introducing Type-7 sorting is that it allows a client to perform sorting even if it does not implement/support Z39.50 sort. Virtually all Z39.50 client software supports RPN queries. It also may improve performance because the sort criteria is specified along with the search query.

The sort is triggered by the presence of type 7, and the value of type 7 specifies the sortRelation . The value for type 7 is 1 for ascending and 2 for descending. For the sortElement only the generic part is handled. If generic sortKey is of type sortField, then attribute type 1 is present and the value is sortField (InternationalString). If generic sortKey is of type sortAttributes, then the attributes in the list are used. Generic sortKey of type elementSpec is not supported.

The term in the sorting Attribute-Plus-Term combo should hold an integer. The value is 0 for primary sorting criteria, 1 for second criteria, etc.

Facets

YAZ supports facets in the Solr, SRU 2.0 and Z39.50 protocols.

Like Type-1/RPN, YAZ supports a string notation for specifying facets. This notataion maps straight to facets.asn. The notation is parsed by function `yaz_pqf_parse_facet_list` defined in header `yaz/pquery.h`.

For ZOOM C the facets are specified by option "facets". For yaz-client, the 'facets' command is used.

The grammar of this specification is as follows:

```
facet-spec ::= facet-list

facet-list ::= facet-list ',' attr-spec | attr-spec

attr-spec ::= attr-spec '@attr' string | '@attr' string
```

The notation is inspired by PQF. The string following '@attr' must not include blanks and is of the form *type=value*, where *type* is an integer and *value* is a string or an integer.

There is no formal facets attribute set (it is not given in the protocol by the facets, although it could). The following types apply:

Type	**Description**
1	Field-name. This is often a string, e.g. "Author", "Year", etc.
2	Sort order. Value should be an integer. Value 0: count descending (frequency). Value 1: alpha ascending.
3	Number of terms requested.
4	Start offset (starting from 1)

Table 7.4: Facet attributes

Chapter 8

The ODR Module

Introduction

ODR is the BER-encoding/decoding subsystem of YAZ. Care has been taken to isolate ODR from the rest of the package - specifically from the transport interface. ODR may be used in any context where basic ASN.1/BER representations are used.

If you are only interested in writing a Z39.50 implementation based on the PDUs that are already provided with YAZ, you only need to concern yourself with the section on managing ODR streams (Section 8.2). Only if you need to implement ASN.1 beyond that which has been provided, should you worry about the second half of the documentation (Section 8.3). If you use one of the higher-level interfaces, you can skip this section entirely.

This is important, so we'll repeat it for emphasis: *You do not need to read Section 8.3 to implement Z39.50 with YAZ.*

If you need a part of the protocol that isn't already in YAZ, you should contact the authors before going to work on it yourself: We might already be working on it. Conversely, if you implement a useful part of the protocol before us, we'd be happy to include it in a future release.

Using ODR

ODR Streams

Conceptually, the ODR stream is the source of encoded data in the decoding mode; when encoding, it is the receptacle for the encoded data. Before you can use an ODR stream it must be allocated. This is done with the function

```
ODR odr_createmem(int direction);
```

The `odr_createmem()` function takes as argument one of three manifest constants: `ODR_ENCODE`, `ODR_DECODE`, or `ODR_PRINT`. An ODR stream can be in only one mode - it is not possible to change its

mode once it's selected. Typically, your program will allocate at least two ODR streams - one for decoding, and one for encoding.

When you're done with the stream, you can use

```
void odr_destroy(ODR o);
```

to release the resources allocated for the stream.

Memory Management

Two forms of memory management take place in the ODR system. The first one, which has to do with allocating little bits of memory (sometimes quite large bits of memory, actually) when a protocol package is decoded, and turned into a complex of interlinked structures. This section deals with this system, and how you can use it for your own purposes. The next section deals with the memory management which is required when encoding data - to make sure that a large enough buffer is available to hold the fully encoded PDU.

The ODR module has its own memory management system, which is used whenever memory is required. Specifically, it is used to allocate space for data when decoding incoming PDUs. You can use the memory system for your own purposes, by using the function

```
void *odr_malloc(ODR o, size_t size);
```

You can't use the normal `free(2)` routine to free memory allocated by this function, and ODR doesn't provide a parallel function. Instead, you can call

```
void odr_reset(ODR o);
```

when you are done with the memory: Everything allocated since the last call to `odr_reset()` is released. The `odr_reset()` call is also required to clear up an error condition on a stream.

The function

```
size_t odr_total(ODR o);
```

returns the number of bytes allocated on the stream since the last call to `odr_reset()`.

The memory subsystem of ODR is fairly efficient at allocating and releasing little bits of memory. Rather than managing the individual, small bits of space, the system maintains a free-list of larger chunks of memory, which are handed out in small bits. This scheme is generally known as a *nibble-memory* system. It is very useful for maintaining short-lived constructions such as protocol PDUs.

If you want to retain a bit of memory beyond the next call to `odr_reset()`, you can use the function

```
ODR_MEM odr_extract_mem(ODR o);
```

This function will give you control of the memory recently allocated on the ODR stream. The memory will live (past calls to `odr_reset()`), until you call the function

```
void odr_release_mem(ODR_MEM p);
```

The opaque `ODR_MEM` handle has no other purpose than referencing the memory block for you until you want to release it.

You can use `odr_extract_mem()` repeatedly between allocating data, to retain individual control of separate chunks of data.

Encoding and Decoding Data

When encoding data, the ODR stream will write the encoded octet string in an internal buffer. To retrieve the data, use the function

```
char *odr_getbuf(ODR o, int *len, int *size);
```

The integer pointed to by len is set to the length of the encoded data, and a pointer to that data is returned. `*size` is set to the size of the buffer (unless `size` is null, signaling that you are not interested in the size). The next call to a primitive function using the same ODR stream will overwrite the data, unless a different buffer has been supplied using the call

```
void odr_setbuf(ODR o, char *buf, int len, int can_grow);
```

which sets the encoding (or decoding) buffer used by `o` to `buf`, using the length `len`. Before a call to an encoding function, you can use `odr_setbuf()` to provide the stream with an encoding buffer of sufficient size (length). The `can_grow` parameter tells the encoding ODR stream whether it is allowed to use `realloc(2)` to increase the size of the buffer when necessary. The default condition of a new encoding stream is equivalent to the results of calling

```
odr_setbuf(stream, 0, 0, 1);
```

In this case, the stream will allocate and reallocate memory as necessary. The stream reallocates memory by repeatedly doubling the size of the buffer - the result is that the buffer will typically reach its maximum, working size with only a small number of reallocation operations. The memory is freed by the stream when the latter is destroyed, unless it was assigned by the user with the `can_grow` parameter set to zero (in this case, you are expected to retain control of the memory yourself).

To assume full control of an encoded buffer, you must first call `odr_getbuf()` to fetch the buffer and its length. Next, you should call `odr_setbuf()` to provide a different buffer (or a null pointer) to the stream. In the simplest case, you will reuse the same buffer over and over again, and you will just need to call `odr_getbuf()` after each encoding operation to get the length and address of the buffer. Note that the stream may reallocate the buffer during an encoding operation, so it is necessary to retrieve the correct address after each encoding operation.

It is important to realize that the ODR stream will not release this memory when you call `odr_reset()`: It will merely update its internal pointers to prepare for the encoding of a new data value. When the stream is released by the `odr_destroy()` function, the memory given to it by `odr_setbuf` will be released *only* if the `can_grow` parameter to `odr_setbuf()` was nonzero. The `can_grow` parameter, in other words, is a way of signaling who is to own the buffer, you or the ODR stream. If you never call `odr_setbuf()` on your encoding stream, which is typically the case, the buffer allocated by the stream will belong to the stream by default.

When you wish to decode data, you should first call `odr_setbuf()`, to tell the decoding stream where to find the encoded data, and how long the buffer is (the `can_grow` parameter is ignored by a decoding stream). After this, you can call the function corresponding to the data you wish to decode (e.g. `odr_integer()` odr `z_APDU()`).

Example 8.1 Encoding and decoding functions

```
int odr_integer(ODR o, Odr_int **p, int optional, const char *name);

int z_APDU(ODR o, Z_APDU **p, int optional, const char *name);
```

If the data is absent (or doesn't match the tag corresponding to the type), the return value will be either 0 or 1 depending on the `optional` flag. If `optional` is 0 and the data is absent, an error flag will be raised in the stream, and you'll need to call `odr_reset()` before you can use the stream again. If `optional` is nonzero, the pointer *pointed* to/ by `p` will be set to the null value, and the function will return 1. The `name` argument is used to pretty-print the tag in question. It may be set to `NULL` if pretty-printing is not desired.

If the data value is found where it's expected, the pointer *pointed to* by the `p` argument will be set to point to the decoded type. The space for the type will be allocated and owned by the ODR stream, and it will live until you call `odr_reset()` on the stream. You cannot use `free(2)` to release the memory. You can decode several data elements (by repeated calls to `odr_setbuf()` and your decoding function), and new memory will be allocated each time. When you do call `odr_reset()`, everything decoded since the last call to `odr_reset()` will be released.

Example 8.2 Encoding and decoding of an integer

The use of the double indirection can be a little confusing at first (its purpose will become clear later on, hopefully), so an example is in order. We'll encode an integer value, and immediately decode it again using a different stream. A useless, but informative operation.

```
void do_nothing_useful(Odr_int value)
{
    ODR encode, decode;
    Odr_int *valp, *resvalp;
    char *bufferp;
```

```
    int len;

    /* allocate streams */
    if (!(encode = odr_createmem(ODR_ENCODE)))
        return;
    if (!(decode = odr_createmem(ODR_DECODE)))
        return;

    valp = &value;
    if (odr_integer(encode, &valp, 0, 0) == 0)
    {
        printf("encoding went bad\n");
        return;
    }
    bufferp = odr_getbuf(encode, &len, 0);
    printf("length of encoded data is %d\n", len);

    /* now let's decode the thing again */
    odr_setbuf(decode, bufferp, len, 0);
    if (odr_integer(decode, &resvalp, 0, 0) == 0)
    {
        printf("decoding went bad\n");
        return;
    }
    /* ODR_INT_PRINTF format for printf (such as %d) */
    printf("the value is " ODR_INT_PRINTF "\n", *resvalp);

    /* clean up */
    odr_destroy(encode);
    odr_destroy(decode);
}
```

This looks like a lot of work, offhand. In practice, the ODR streams will typically be allocated once, in the beginning of your program (or at the beginning of a new network session), and the encoding and decoding will only take place in a few, isolated places in your program, so the overhead is quite manageable.

Printing

When an ODR stream is created of type `ODR_PRINT` the ODR module will print the contents of a PDU in a readable format. By default output is written to the `stderr` stream. This behavior can be changed, however, by calling the function

```
odr_setprint(ODR o, FILE *file);
```

before encoders or decoders are being invoked. It is also possible to direct the output to a buffer (or indeed another file), by using the more generic mechanism:

```
void odr_set_stream(ODR o, void *handle,
                    void (*stream_write)(ODR o, void *handle, int type,
                                         const char *buf, int len),
                    void (*stream_close)(void *handle));
```

Here the user provides an opaque handle and two handlers, *stream_write* for writing, and *stream_close* which is supposed to close/free resources associated with handle. The *stream_close* handler is optional and if NULL for the function is provided, it will not be invoked. The *stream_write* takes the ODR handle as parameter, the user-defined handle, a type ODR_OCTETSTRING, ODR_VISIBLESTRING which indicates the type of contents being written.

Another utility useful for diagnostics (error handling) or as part of the printing facilities is:

```
const char **odr_get_element_path(ODR o);
```

which returns a list of current elements that ODR deals with at the moment. For the returned array, say ar, then ar[0] is the top level element, ar[n] is the last. The last element has the property that ar[n+1] ==NULL.

Example 8.3 Element Path for record

For a database record part of a PresentResponse the array returned by odr_get_element is presentResponse, databaseOrSurDiagnostics, ?, record, ?, databaseRecord . The question mark appears due to unnamed constructions.

Diagnostics

The encoding/decoding functions all return 0 when an error occurs. Until you call odr_reset(), you cannot use the stream again, and any function called will immediately return 0.

To provide information to the programmer or administrator, the function

```
void odr_perror(ODR o, char *message);
```

is provided, which prints the message argument to stderr along with an error message from the stream.

You can also use the function

```
int odr_geterror(ODR o);
```

to get the current error number from the screen. The number will be one of these constants:

The character string array

```
char *odr_errlist[]
```

can be indexed by the error code to obtain a human-readable representation of the problem.

code	Description
OMEMORY	Memory allocation failed.
OSYSERR	A system- or library call has failed. The standard diagnostic variable `errno` should be examined to determine the actual error.
OSPACE	No more space for encoding. This will only occur when the user has explicitly provided a buffer for an encoding stream without allowing the system to allocate more space.
OREQUIRED	This is a common protocol error; A required data element was missing during encoding or decoding.
OUNEXPECTED	An unexpected data element was found during decoding.
OOTHER	Other error. This is typically an indication of misuse of the ODR system by the programmer, and also that the diagnostic system isn't as good as it should be, yet.

Table 8.1: ODR Error codes

Summary and Synopsis

```
#include <yaz/odr.h>

ODR odr_createmem(int direction);

void odr_destroy(ODR o);

void odr_reset(ODR o);

char *odr_getbuf(ODR o, int *len, int *size);

void odr_setbuf(ODR o, char *buf, int len, int can_grow);

void *odr_malloc(ODR o, int size);

NMEM odr_extract_mem(ODR o);

int odr_geterror(ODR o);

void odr_perror(ODR o, const char *message);

extern char *odr_errlist[];
```

Programming with ODR

The API of ODR is designed to reflect the structure of ASN.1, rather than BER itself. Future releases may be able to represent data in other external forms.

Tip
There is an ASN.1 tutorial available at this site. This site also has standards for ASN.1 (X.680) and BER (X.690) online.

The ODR interface is based loosely on that of the Sun Microsystems XDR routines. Specifically, each function which corresponds to an ASN.1 primitive type has a dual function. Depending on the settings of the ODR stream which is supplied as a parameter, the function may be used either to encode or decode data. The functions that can be built using these primitive functions, to represent more complex data types, share this quality. The result is that you only have to enter the definition for a type once - and you have the functionality of encoding, decoding (and pretty-printing) all in one unit. The resulting C source code is quite compact, and is a pretty straightforward representation of the source ASN.1 specification.

In many cases, the model of the XDR functions works quite well in this role. In others, it is less elegant. Most of the hassle comes from the optional SEQUENCE members which don't exist in XDR.

The Primitive ASN.1 Types

ASN.1 defines a number of primitive types (many of which correspond roughly to primitive types in structured programming languages, such as C).

INTEGER

The ODR function for encoding or decoding (or printing) the ASN.1 INTEGER type looks like this:

```
int odr_integer(ODR o, Odr_int **p, int optional, const char *name);
```

The `Odr_int` is just a simple integer.

This form is typical of the primitive ODR functions. They are named after the type of data that they encode or decode. They take an ODR stream, an indirect reference to the type in question, and an `optional` flag (corresponding to the OPTIONAL keyword of ASN.1) as parameters. They all return an integer value of either one or zero. When you use the primitive functions to construct encoders for complex types of your own, you should follow this model as well. This ensures that your new types can be reused as elements in yet more complex types.

The `o` parameter should obviously refer to a properly initialized ODR stream of the right type (encoding/decoding/printing) for the operation that you wish to perform.

When encoding or printing, the function first looks at `* p`. If `* p` (the pointer pointed to by `p`) is a null pointer, this is taken to mean that the data element is absent. If the `optional` parameter is nonzero, the

function will return one (signifying success) without any further processing. If the `optional` is zero, an internal error flag is set in the ODR stream, and the function will return 0. No further operations can be carried out on the stream without a call to the function `odr_reset()`.

If `*p` is not a null pointer, it is expected to point to an instance of the data type. The data will be subjected to the encoding rules, and the result will be placed in the buffer held by the ODR stream.

The other ASN.1 primitives have similar functions that operate in similar manners:

BOOLEAN

```
int odr_bool(ODR o, Odr_bool **p, int optional, const char *name);
```

REAL

Not defined.

NULL

```
int odr_null(ODR o, Odr_null **p, int optional, const char *name);
```

In this case, the value of **p is not important. If `*p` is different from the null pointer, the null value is present, otherwise it's absent.

OCTET STRING

```
typedef struct odr_oct
{
    unsigned char *buf;
    int len;
} Odr_oct;

int odr_octetstring(ODR o, Odr_oct **p, int optional,
                    const char *name);
```

The `buf` field should point to the character array that holds the octetstring. The `len` field holds the actual length. The character array need not be null-terminated.

To make things a little easier, an alternative is given for string types that are not expected to contain embedded NULL characters (e.g. VisibleString):

```
int odr_cstring(ODR o, char **p, int optional, const char *name);
```

which encodes or decodes between OCTETSTRING representations and null-terminated C strings.

Functions are provided for the derived string types, e.g.:

```
int odr_visiblestring(ODR o, char **p, int optional,
                      const char *name);
```

BIT STRING

```
int odr_bitstring(ODR o, Odr_bitmask **p, int optional,
                  const char *name);
```

The opaque type `Odr_bitmask` is only suitable for holding relatively brief bit strings, e.g. for options fields, etc. The constant `ODR_BITMASK_SIZE` multiplied by 8 gives the maximum possible number of bits.

A set of macros are provided for manipulating the `Odr_bitmask` type:

```
void ODR_MASK_ZERO(Odr_bitmask *b);

void ODR_MASK_SET(Odr_bitmask *b, int bitno);

void ODR_MASK_CLEAR(Odr_bitmask *b, int bitno);

int ODR_MASK_GET(Odr_bitmask *b, int bitno);
```

The functions are modeled after the manipulation functions that accompany the `fd_set` type used by the `select(2)` call. `ODR_MASK_ZERO` should always be called first on a new bitmask, to initialize the bits to zero.

OBJECT IDENTIFIER

```
int odr_oid(ODR o, Odr_oid **p, int optional, const char *name);
```

The C OID representation is simply an array of integers, terminated by the value -1 (the `Odr_oid` type is synonymous with the `short` type). We suggest that you use the OID database module (see Section 7.2.1) to handle object identifiers in your application.

Tagging Primitive Types

The simplest way of tagging a type is to use the `odr_implicit_tag()` or `odr_explicit_tag()` macros:

```
int odr_implicit_tag(ODR o, Odr_fun fun, int class, int tag,
                     int optional, const char *name);

int odr_explicit_tag(ODR o, Odr_fun fun, int class, int tag,
                     int optional, const char *name);
```

To create a type derived from the integer type by implicit tagging, you might write:

```
     MyInt ::= [210] IMPLICIT INTEGER
```

In the ODR system, this would be written like:

```
int myInt(ODR o, Odr_int **p, int optional, const char *name)
{
    return odr_implicit_tag(o, odr_integer, p,
          ODR_CONTEXT, 210, optional, name);
}
```

The function `myInt()` can then be used like any of the primitive functions provided by ODR. Note that the behavior of `odr_explicit_tag()` and `odr_implicit_tag()` macros act exactly the same as the functions they are applied to - they respond to error conditions, etc, in the same manner - they simply have three extra parameters. The class parameter may take one of the values: `ODR_CONTEXT`, `ODR_PRIVATE`, `ODR_UNIVERSAL`, or `/ODR_APPLICATION`.

Constructed Types

Constructed types are created by combining primitive types. The ODR system only implements the SEQUENCE and SEQUENCE OF constructions (although adding the rest of the container types should be simple enough, if the need arises).

For implementing SEQUENCEs, the functions

```
int odr_sequence_begin(ODR o, void *p, int size, const char *name);
int odr_sequence_end(ODR o);
```

are provided.

The `odr_sequence_begin()` function should be called in the beginning of a function that implements a SEQUENCE type. Its parameters are the ODR stream, a pointer (to a pointer to the type you're implementing), and the `size` of the type (typically a C structure). On encoding, it returns 1 if `* p` is a null pointer. The `size` parameter is ignored. On decoding, it returns 1 if the type is found in the data stream. `size` bytes of memory are allocated, and `*p` is set to point to this space. The `odr_sequence_end()` is called at the end of the complex function. Assume that a type is defined like this:

```
MySequence ::= SEQUENCE {
     intval INTEGER,
     boolval BOOLEAN OPTIONAL
}
```

The corresponding ODR encoder/decoder function and the associated data structures could be written like this:

```
typedef struct MySequence
{
    Odr_int *intval;
    Odr_bool *boolval;
} MySequence;

int mySequence(ODR o, MySequence **p, int optional, const char *name)
{
    if (odr_sequence_begin(o, p, sizeof(**p), name) == 0)
        return optional && odr_ok(o);
    return
        odr_integer(o, &(*p)->intval, 0, "intval") &&
        odr_bool(o, &(*p)->boolval, 1, "boolval") &&
        odr_sequence_end(o);
}
```

Note the 1 in the call to `odr_bool()`, to mark that the sequence member is optional. If either of the member types had been tagged, the macros `odr_implicit_tag()` or `odr_explicit_tag()` could have been used. The new function can be used exactly like the standard functions provided with ODR. It will encode, decode or pretty-print a data value of the `MySequence` type. We like to name types with an initial capital, as done in ASN.1 definitions, and to name the corresponding function with the first character of the name in lower case. You could, of course, name your structures, types, and functions any way you please - as long as you're consistent, and your code is easily readable. `odr_ok` is just that - a predicate that returns the state of the stream. It is used to ensure that the behavior of the new type is compatible with the interface of the primitive types.

Tagging Constructed Types

Note
See Section 8.3.2 for information on how to tag the primitive types, as well as types that are already defined.

Implicit Tagging

Assume the type above had been defined as

```
MySequence ::= [10] IMPLICIT SEQUENCE {
      intval INTEGER,
```

```
      boolval BOOLEAN OPTIONAL
}
```

You would implement this in ODR by calling the function

```
int odr_implicit_settag(ODR o, int class, int tag);
```

which overrides the tag of the type immediately following it. The macro `odr_implicit_tag()` works by calling `odr_implicit_settag()` immediately before calling the function pointer argument. Your type function could look like this:

```
int mySequence(ODR o, MySequence **p, int optional, const char *name)
{
    if (odr_implicit_settag(o, ODR_CONTEXT, 10) == 0 ||
        odr_sequence_begin(o, p, sizeof(**p), name) == 0)
        return optional && odr_ok(o);
    return
        odr_integer(o, &(*p)->intval, 0, "intval") &&
        odr_bool(o, &(*p)->boolval, 1, "boolval") &&
        odr_sequence_end(o);
}
```

The definition of the structure `MySequence` would be the same.

Explicit Tagging

Explicit tagging of constructed types is a little more complicated, since you are in effect adding a level of construction to the data.

Assume the definition:

```
MySequence ::= [10] IMPLICIT SEQUENCE {
   intval INTEGER,
   boolval BOOLEAN OPTIONAL
}
```

Since the new type has an extra level of construction, two new functions are needed to encapsulate the base type:

```
int odr_constructed_begin(ODR o, void *p, int class, int tag,
                          const char *name);

int odr_constructed_end(ODR o);
```

Assume that the IMPLICIT in the type definition above were replaced with EXPLICIT (or that the IMPLICIT keyword was simply deleted, which would be equivalent). The structure definition would look the same, but the function would look like this:

```
int mySequence(ODR o, MySequence **p, int optional, const char *name)
{
    if (odr_constructed_begin(o, p, ODR_CONTEXT, 10, name) == 0)
        return optional && odr_ok(o);
    if (o->direction == ODR_DECODE)
        *p = odr_malloc(o, sizeof(**p));
    if (odr_sequence_begin(o, p, sizeof(**p), 0) == 0)
    {
        *p = 0; /* this is almost certainly a protocol error */
        return 0;
    }
    return
        odr_integer(o, &(*p)->intval, 0, "intval") &&
        odr_bool(o, &(*p)->boolval, 1, "boolval") &&
        odr_sequence_end(o) &&
        odr_constructed_end(o);
}
```

Notice that the interface here gets kind of nasty. The reason is simple: Explicitly tagged, constructed types are fairly rare in the protocols that we care about, so the aesthetic annoyance (not to mention the dangers of a cluttered interface) is less than the time that would be required to develop a better interface. Nevertheless, it is far from satisfying, and it's a point that will be worked on in the future. One option for you would be to simply apply the `odr_explicit_tag()` macro to the first function, and not have to worry about `odr_constructed_*` yourself. Incidentally, as you might have guessed, the `odr_sequence_` functions are themselves implemented using the `/odr_constructed_` functions.

SEQUENCE OF

To handle sequences (arrays) of a specific type, the function

```
int odr_sequence_of(ODR o, int (*fun)(ODR o, void *p, int optional),
                    void *p, int *num, const char *name);
```

The `fun` parameter is a pointer to the decoder/encoder function of the type. `p` is a pointer to an array of pointers to your type. `num` is the number of elements in the array.

Assume a type

```
MyArray ::= SEQUENCE OF INTEGER
```

The C representation might be

```
typedef struct MyArray
{
    int num_elements;
    Odr_int **elements;
} MyArray;
```

And the function might look like

```
int myArray(ODR o, MyArray **p, int optional, const char *name)
{
    if (o->direction == ODR_DECODE)
        *p = odr_malloc(o, sizeof(**p));
    if (odr_sequence_of(o, odr_integer, &(*p)->elements,
        &(*p)->num_elements, name))
        return 1;
    *p = 0;
        return optional && odr_ok(o);
}
```

CHOICE Types

The choice type is used fairly often in some ASN.1 definitions, so some work has gone into streamlining its interface.

CHOICE types are handled by the function:

```
int odr_choice(ODR o, Odr_arm arm[], void *p, void *whichp,
               const char *name);
```

The `arm` array is used to describe each of the possible types that the CHOICE type may assume. Internally in your application, the CHOICE type is represented as a discriminated union. That is, a C union accompanied by an integer (or enum) identifying the active 'arm' of the union. `whichp` is a pointer to the union discriminator. When encoding, it is examined to determine the current type. When decoding, it is set to reference the type that was found in the input stream.

The Odr_arm type is defined thus:

```
typedef struct odr_arm
{
    int tagmode;
    int class;
    int tag;
    int which;
    Odr_fun fun;
    char *name;
} Odr_arm;
```

The interpretation of the fields are:

tagmode Either `ODR_IMPLICIT`, `ODR_EXPLICIT`, or `ODR_NONE` (-1) to mark no tagging.

which The value of the discriminator that corresponds to this CHOICE element. Typically, it will be a #defined constant, or an enum member.

fun A pointer to a function that implements the type of the CHOICE member. It may be either a standard ODR type or a type defined by yourself.

name Name of tag.

A handy way to prepare the array for use by the `odr_choice()` function is to define it as a static, initialized array in the beginning of your decoding/encoding function. Assume the type definition:

```
MyChoice ::= CHOICE {
    untagged INTEGER,
    tagged   [99] IMPLICIT INTEGER,
    other    BOOLEAN
}
```

Your C type might look like

```
typedef struct MyChoice
{
    enum
    {
        MyChoice_untagged,
        MyChoice_tagged,
        MyChoice_other
    } which;
    union
    {
        Odr_int *untagged;
        Odr_int *tagged;
        Odr_bool *other;
    } u;
};
```

And your function could look like this:

```
int myChoice(ODR o, MyChoice **p, int optional, const char *name)
{
    static Odr_arm arm[] =
    {
      {-1, -1, -1, MyChoice_untagged, odr_integer, "untagged"},
      {ODR_IMPLICIT, ODR_CONTEXT, 99, MyChoice_tagged, odr_integer,
      "tagged"},
      {-1, -1, -1, MyChoice_other, odr_boolean, "other"},
      {-1, -1, -1, -1, 0}
    };

    if (o->direction == ODR_DECODE)
        *p = odr_malloc(o, sizeof(**p);
    else if (!*p)
        return optional && odr_ok(o);

    if (odr_choice(o, arm, &(*p)->u, &(*p)->which), name)
```

```
        return 1;
    *p = 0;
        return optional && odr_ok(o);
}
```

In some cases (say, a non-optional choice which is a member of a sequence), you can "embed" the union and its discriminator in the structure belonging to the enclosing type, and you won't need to fiddle with memory allocation to create a separate structure to wrap the discriminator and union.

The corresponding function is somewhat nicer in the Sun XDR interface. Most of the complexity of this interface comes from the possibility of declaring sequence elements (including CHOICEs) optional.

The ASN.1 specifications naturally require that each member of a CHOICE have a distinct tag, so they can be told apart on decoding. Sometimes it can be useful to define a CHOICE that has multiple types that share the same tag. You'll need some other mechanism, perhaps keyed to the context of the CHOICE type. In effect, we would like to introduce a level of context-sensitiveness to our ASN.1 specification. When encoding an internal representation, we have no problem, as long as each CHOICE member has a distinct discriminator value. For decoding, we need a way to tell the choice function to look for a specific arm of the table. The function

```
void odr_choice_bias(ODR o, int what);
```

provides this functionality. When called, it leaves a notice for the next call to `odr_choice()` to be called on the decoding stream `o`, that only the `arm` entry with a `which` field equal to `what` should be tried.

The most important application (perhaps the only one, really) is in the definition of application-specific EXTERNAL encoders/decoders which will automatically decode an ANY member given the direct or indirect reference.

Debugging

The protocol modules are suffering somewhat from a lack of diagnostic tools at the moment. Specifically ways to pretty-print PDUs that aren't recognized by the system. We'll include something to this end in a not-too-distant release. In the meantime, what we do when we get packages we don't understand is to compile the ODR module with `ODR_DEBUG` defined. This causes the module to dump tracing information as it processes data units. With this output and the protocol specification (Z39.50), it is generally fairly easy to see what goes wrong.

Chapter 9

The COMSTACK Module

Synopsis (blocking mode)

```
    COMSTACK stack;
    char *buf = 0;
    int size = 0, length_incoming;
    char server_address_str[] = "localhost:9999";
    void *server_address_ip;
    int status;

    char *protocol_package = "GET / HTTP/1.0\r\n\r\n";
    int protocol_package_length = strlen(protocol_package);

    stack = cs_create(tcpip_type, 1, PROTO_HTTP);
    if (!stack) {
        perror("cs_create");  /* use perror() here since we have no stack  ↩
           yet */
        return -1;
    }

    server_address_ip = cs_straddr(stack, server_address_str);
    if (!server_address_ip) {
        fprintf(stderr, "cs_straddr: address could not be resolved\n");
        return -1;
    }

    status = cs_connect(stack, server_address_ip);
    if (status) {
        fprintf(stderr, "cs_connect: %s\n", cs_strerror(stack));
        return -1;
    }

    status = cs_rcvconnect(stack);
    if (status) {
        fprintf(stderr, "cs_rcvconnect: %s\n", cs_strerror(stack));
```

```
        return -1;
    }

    status = cs_put(stack, protocol_package, protocol_package_length);
    if (status) {
        fprintf(stderr, "cs_put: %s\n", cs_strerror(stack));
        return -1;
    }

    /* Now get a response */
    length_incoming = cs_get(stack, &buf, &size);
    if (!length_incoming) {
        fprintf(stderr, "Connection closed\n");
        return -1;
    } else if (length_incoming < 0) {
        fprintf(stderr, "cs_get: %s\n", cs_strerror(stack));
        return -1;
    }

    /* Print result */
    fwrite(buf, length_incoming, 1, stdout);

    /* clean up */
    cs_close(stack);
    if (buf)
        xfree(buf);
    return 0;
```

Introduction

The COMSTACK subsystem provides a transparent interface to different types of transport stacks for the exchange of BER-encoded data and HTTP packets. At present, the RFC1729 method (BER over TCP/IP), local UNIX socket and an experimental SSL stack are supported, but others may be added in time. The philosophy of the module is to provide a simple interface by hiding unused options and facilities of the underlying libraries. This is always done at the risk of losing generality, and it may prove that the interface will need extension later on.

Note
There hasn't been interest in the XTImOSI stack for some years. Therefore, it is no longer supported.

The interface is implemented in such a fashion that only the sub-layers constructed to the transport methods that you wish to use in your application are linked in.

You will note that even though simplicity was a goal in the design, the interface is still orders of magnitudes more complex than the transport systems found in many other packages. One reason is that the interface

needs to support the somewhat different requirements of the different lower-layer communications stacks; another important reason is that the interface seeks to provide a more or less industrial-strength approach to asynchronous event-handling. When no function is allowed to block, things get more complex - particularly on the server side. We urge you to have a look at the demonstration client and server provided with the package. They are meant to be easily readable and instructive, while still being at least moderately useful.

Common Functions

Managing Endpoints

```
COMSTACK cs_create(CS_TYPE type, int blocking, int protocol);
```

Creates an instance of the protocol stack - a communications endpoint. The `type` parameter determines the mode of communication. At present the following values are supported:

`tcpip_type` TCP/IP (BER over TCP/IP or HTTP over TCP/IP)

`ssl_type` Secure Socket Layer (SSL). This COMSTACK is experimental and is not fully implemented. If HTTP is used, this effectively is HTTPS.

`unix_type` Unix socket (unix only). Local Transfer via file socket. See unix(7).

The `cs_create` function returns a null-pointer if a system error occurs. The `blocking` parameter should be '1' if you wish the association to operate in blocking mode, and '0' otherwise. The `protocol` field should be `PROTO_Z3950` or `PROTO_HTTP`. Protocol `PROTO_SR` is no longer supported.

```
void cs_close(COMSTACK handle);
```

Closes the connection (as elegantly as the lower layers will permit), and releases the resources pointed to by the `handle` parameter. The `handle` should not be referenced again after this call.

Note
We really need a soft disconnect, don't we?

Data Exchange

```
int cs_put(COMSTACK handle, char *buf, int len);
```

Sends `buf` down the wire. In blocking mode, this function will return only when a full buffer has been written, or an error has occurred. In nonblocking mode, it's possible that the function will be unable to send the full buffer at once, which will be indicated by a return value of 1. The function will keep track of the number of octets already written; you should call it repeatedly with the same values of `buf` and `len`, until the buffer has been transmitted. When a full buffer has been sent, the function will return 0 for success. The return value -1 indicates an error condition (see below).

```
int cs_get(COMSTACK handle, char **buf, int *size);
```

Receives a PDU or HTTP Response from the peer. Returns the number of bytes read. In nonblocking mode, it is possible that not all of the packet can be read at once. In this case, the function returns 1. To simplify the interface, the function is responsible for managing the size of the buffer. It will be reallocated if necessary to contain large packages, and will sometimes be moved around internally by the subsystem when partial packages are read. Before calling `cs_get` for the first time, the buffer can be initialized to the null pointer, and the length should also be set to 0 (cs_get will perform a `malloc(2)` on the buffer for you). When a full buffer has been read, the size of the package is returned (which will always be greater than 1). The return value -1 indicates an error condition.

See also the `cs_more()` function below.

```
int cs_more(COMSTACK handle);
```

The `cs_more()` function should be used in conjunction with `cs_get` and `select(2)`. The `cs_get()` function will sometimes (notably in the TCP/IP mode) read more than a single protocol package off the network. When this happens, the extra package is stored by the subsystem. After calling `cs_get()`, and before waiting for more input, You should always call `cs_more()` to check if there's a full protocol package already read. If `cs_more()` returns 1, `cs_get()` can be used to immediately fetch the new package. For the mOSI subsystem, the function should always return 0, but if you want your stuff to be protocol independent, you should use it.

Note

The `cs_more()` function is required because the RFC1729-method does not provide a way of separating individual PDUs, short of partially decoding the BER. Some other implementations will carefully nibble at the packet by calling `read(2)` several times. This was felt to be too inefficient (or at least clumsy) - hence the call for this extra function.

```
int cs_look(COMSTACK handle);
```

This function is useful when you're operating in nonblocking mode. Call it when `select(2)` tells you there's something happening on the line. It returns one of the following values:

CS_NONE No event is pending. The data found on the line was not a complete package.

CS_CONNECT A response to your connect request has been received. Call `cs_rcvconnect` to process the event and to finalize the connection establishment.

CS_DISCON The other side has closed the connection (or maybe sent a disconnect request - but do we care? Maybe later). Call `cs_close` to close your end of the association as well.

CS_LISTEN A connect request has been received. Call `cs_listen` to process the event.

CS_DATA There's data to be found on the line. Call `cs_get` to get it.

Note

You should be aware that even if `cs_look()` tells you that there's an event event pending, the corresponding function may still return and tell you there was nothing to be found. This means that only part of a package was available for reading. The same event will show up again, when more data has arrived.

```
int cs_fileno(COMSTACK h);
```

returns the file descriptor of the association. Use this when file-level operations on the endpoint are required (`select(2)` operations, specifically).

Client Side

```
int cs_connect(COMSTACK handle, void *address);
```

Initiate a connection with the target at `address` (more on addresses below). The function will return 0 on success, and 1 if the operation does not complete immediately (this will only happen on a nonblocking endpoint). In this case, use `cs_rcvconnect` to complete the operation, when `select(2)` or `poll(2)` reports input pending on the association.

```
int cs_rcvconnect(COMSTACK handle);
```

Complete a connect operation initiated by `cs_connect()`. It will return 0 on success; 1 if the operation has not yet completed (in this case, call the function again later); -1 if an error has occurred.

Server Side

To establish a server under the inetd server, you can use

```
COMSTACK cs_createbysocket(int socket, CS_TYPE type, int blocking,
                           int protocol);
```

The `socket` parameter is an established socket (when your application is invoked from inetd, the socket will typically be 0. The following parameters are identical to the ones for `cs_create`.

```
int cs_bind(COMSTACK handle, void *address, int mode)
```

Binds a local address to the endpoint. Read about addresses below. The `mode` parameter should be either `CS_CLIENT` or `CS_SERVER`.

```
int cs_listen(COMSTACK handle, char *addr, int *addrlen);
```

Call this to process incoming events on an endpoint that has been bound in listening mode. It will return 0 to indicate that the connect request has been received, 1 to signal a partial reception, and -1 to indicate an error condition.

```
COMSTACK cs_accept(COMSTACK handle);
```

This finalizes the server-side association establishment, after cs_listen has completed successfully. It returns a new connection endpoint, which represents the new association. The application will typically wish to fork off a process to handle the association at this point, and continue listen for new connections on the old `handle`.

You can use the call

```
const char *cs_addrstr(COMSTACK);
```

on an established connection to retrieve the host-name of the remote host.

Note
You may need to use this function with some care if your name server service is slow or unreliable.

Addresses

The low-level format of the addresses are different depending on the mode of communication you have chosen. A function is provided by each of the lower layers to map a user-friendly string-form address to the binary form required by the lower layers.

```
void *cs_straddr(COMSTACK handle, const char *str);
```

The format for TCP/IP and SSL addresses is:

```
<host> [ ':' <portnum> ]
```

The `hostname` can be either a domain name or an IP address. The port number, if omitted, defaults to 210.

For TCP/IP and SSL, the special hostnames `@`, maps to `IN6ADDR_ANY_INIT` with IPV4 binding as well (bindv6only=0), The special hostname `@4` binds to `INADDR_ANY` (IPV4 only listener). The special hostname `@6` binds to `IN6ADDR_ANY_INIT` with bindv6only=1 (IPV6 only listener).

For UNIX sockets, the format of an address is the socket filename.

When a connection has been established, you can use

```
const char *cs_addrstr(COMSTACK h);
```

to retrieve the host name of the peer system. The function returns a pointer to a static area, which is overwritten on the next call to the function.

A fairly recent addition to the COMSTACK module is the utility function

```
COMSTACK cs_create_host (const char *str, int blocking, void **vp);
```

which is just a wrapper for `cs_create` and `cs_straddr`. The *`str`* is similar to that described for `cs_straddr` but with a prefix denoting the COMSTACK type. Prefixes supported are `tcp:` and `unix:` and `ssl:` for TCP/IP and UNIX and SSL respectively. If no prefix is given, then TCP/IP is used. The *`blocking`* is passed to function `cs_create`. The third parameter *`vp`* is a pointer to COMSTACK stack type specific values. Parameter *`vp`* is reserved for future use. Set it to `NULL`.

SSL

```
void *cs_get_ssl(COMSTACK cs);
```

Returns the SSL handle, `SSL *` for comstack. If comstack is not of type SSL, then NULL is returned.

```
int cs_set_ssl_ctx(COMSTACK cs, void *ctx);
```

Sets SSL context for comstack. The parameter is expected to be of type `SSL_CTX *`. This function should be called just after comstack has been created (before connect, bind, etc). This function returns 1 for success; 0 for failure.

```
int cs_set_ssl_certificate_file(COMSTACK cs, const char *fname);
```

Sets SSL certificate for comstack as a PEM file. This function returns 1 for success; 0 for failure.

```
int cs_get_ssl_peer_certificate_x509(COMSTACK cs, char **buf, int *len)
```

This function returns the peer certificate. If successful, `*buf` and `*len` holds X509 buffer and length respectively. Buffer should be freed with `xfree`. This function returns 1 for success; 0 for failure.

Diagnostics

All functions return -1 if an error occurs. Typically, the functions will return 0 on success, but the data exchange functions (`cs_get`, `cs_put`, `cs_more`) follow special rules. Consult their descriptions.

The error code for the COMSTACK can be retrieved using C macro `cs_errno` which will return one of the error codes `CSYSERR`, `CSOUTSTATE`, `CSNODATA`, ...

```
int cs_errno(COMSTACK handle);
```

You can the textual representation of the error code by using `cs_errmsg`, which works like `strerror(3)`.

```
const char *cs_errmsg(int n);
```

It is also possible to get straight to the textual representation without the error code, by using `cs_strerror`.

```
const char *cs_strerror(COMSTACK h);
```

Summary and Synopsis

```
#include <yaz/comstack.h>

#include <yaz/tcpip.h>  /* this is for TCP/IP and SSL support */
#include <yaz/unix.h>   /* this is for UNIX socket support */

COMSTACK cs_create(CS_TYPE type, int blocking, int protocol);
```

```
COMSTACK cs_createbysocket(int s, CS_TYPE type, int blocking,
                           int protocol);
COMSTACK cs_create_host(const char *str, int blocking,
                        void **vp);

int cs_bind(COMSTACK handle, int mode);

int cs_connect(COMSTACK handle, void *address);

int cs_rcvconnect(COMSTACK handle);

int cs_listen(COMSTACK handle);

COMSTACK cs_accept(COMSTACK handle);

int cs_put(COMSTACK handle, char *buf, int len);

int cs_get(COMSTACK handle, char **buf, int *size);

int cs_more(COMSTACK handle);

void cs_close(COMSTACK handle);

int cs_look(COMSTACK handle);

void *cs_straddr(COMSTACK handle, const char *str);

const char *cs_addrstr(COMSTACK h);
```

Chapter 10

Future Directions

We have a new and better version of the front-end server on the drawing board. Resources and external commitments will govern when we'll be able to do something real with it. Features should include greater flexibility, greater support for access/resource control, and easy support for Explain (possibly with Zebra as an extra database engine).

YAZ is a BER toolkit and as such should support all protocols out there based on that. We'd like to see running ILL applications. It shouldn't be that hard. Another thing that would be interesting is LDAP. Maybe a generic framework for doing IR using both LDAP and Z39.50 transparently.

The SOAP implementation is incomplete. In the future we hope to add more features to it. Perhaps make a WSDL/XML Schema compiler. The authors of libxml2 are already working on XML Schema and RELAX NG compilers so this may not be too hard.

It would be neat to have a proper module mechanism for the Generic Frontend Server so that backend would be dynamically loaded (as shared objects / DLLs).

Other than that, YAZ generally moves in the directions which appear to make the most people happy (including ourselves, as prime users of the software). If there's something you'd like to see in here, then drop us a note and let's see what we can come up with.

Chapter 11

Reference

The material in this chapter is drawn directly from the individual manual entries.

yaz-client

yaz-client — Z39.50/SRU client for implementors

Synopsis

`yaz-client` [`-a` *apdulog*] [`-b` *berdump*] [`-c` *cclfile*] [`-d` *dump*] [`-f` *cmdfile*] [`-k` *size*] [`-m` *marclog*] [`-p` *proxy-addr*] [`-q` *cqlfile*] [`-t` *dispcharset*] [`-u` *auth*] [`-v` *loglevel*] [`-V`] [`-x`] [server-addr]

DESCRIPTION

yaz-client is a Z39.50/SRU client (origin) with a simple command line interface that allows you to test behavior and performance of Z39.50 targets and SRU servers.

From YAZ version 4.1.0 **yaz-client** may also operate as a Solr Web Service client.

If the *server-addr* is specified, the client creates a connection to the Z39.50/SRU target at the address given.

When **yaz-client** is started it tries to read commands from one of the following files:

- Command file if it is given by option -f.
- `.yazclientrc` in current working directory.
- `.yazclientrc` in the user's home directory. The value of the `HOME` is used to determine the home directory. Normally, `HOME` is only set on POSIX systems such as Linux, FreeBSD, Solaris.

OPTIONS

-a *filename* If specified, logging of protocol packages will be appended to the file given. If *filename* is specified as −, the output is written to `stdout`.

-b *filename* If specified, YAZ will dump BER data in readable notation to the file specified. If *filename* is specified as − the output is written to `stdout`.

-c *filename* If specified, CCL configuration will be read from the file given.

-d *dump* If specified, YAZ will dump BER data for all PDUs sent and received to individual files, named *dump*.DDD.`raw`, where DDD is 001, 002, 003, ..

-f *cmdfile* Reads commands from *cmdfile*. When this option is used, YAZ client does not read .yaz-clientrc from current directory or home directory.

-k *size* Sets preferred messages and maximum record size for Initialize Request in kilobytes. Default value is 65536 (64 MB).

-m *filename* If specified, retrieved records will be appended to the file given.

-p *proxy-addr* If specified, the client will use the proxy at the address given. YAZ client will connect to a proxy on the address and port given. The actual target will be specified as part of the InitRequest to inform the proxy about the actual target.

-q *filename* If specified, CQL configuration will be read from the file given.

-t *displaycharset* If displaycharset is given, it specifies name of the character set of the output (on the terminal on which YAZ client is running).

-u *auth* If specified, the *auth* string will be used for authentication.

−v *level* Sets the LOG level to *level*. Level is a sequence of tokens separated by comma. Each token is a integer or a named LOG item - one of `fatal`, `debug`, `warn`, `log`, `malloc`, `all`, `none`.

-V Prints YAZ version.

-x Makes the YAZ client print hex dumps of packages sent and received on standard output.

COMMANDS

The YAZ client accepts the following commands.

open *zurl* Opens a connection to a server. The syntax for *zurl* is the same as described above for connecting from the command line.

Syntax:

[`(tcp|ssl|unix|http)':'`]*host* [:*port*][/*base*]

quit Quits YAZ client

find ***query*** Sends a Search Request using the *query* given. By default the query is assumed to be PQF. See command querytype for more information.

delete ***setname*** Deletes result set with name *setname* on the server.

base ***base1 base2 ...*** Sets the name(s) of the database(s) to search. One or more databases may be specified, separated by blanks. This command overrides the database given in *zurl*.

show **[*start*[+*number*]]** Fetches records by sending a Present Request from the start position given by *start* and a number of records given by *number*. If *start* is not given, then the client will fetch from the position of the last retrieved record plus 1. If *number* is not given, then one record will be fetched at a time.

scan ***term*** Scans database index for a term. The syntax resembles the syntax for find. If you want to scan for the word `water` you could write

```
scan water
```

but if you want to scan only in, say the title field, you would write

```
scan @attr 1=4 water
```

setscan ***set term*** Scans database index for a term within a result set. This is similar to the scan command but has a result set as its first argument.

scanpos ***pos*** Sets preferred position for scan. This value is used in the next scan. By default, position is 1.

scansize ***size*** Sets number of entries to be returned by scan. Default number of entries is 20.

scanstep ***step*** Set step-size for scan. This value is used in the next scan sent to the target. By default step-size is 0.

sort ***sortspecs*** Sorts a result set. The sort command takes a sequence of space-separated sort specifications, with each sort specification consisting of two space-separated words (so that the whole specification list is made up of an even number of words). The first word of each specification holds a field (sort criterion) and the second holds flags. If the sort criterion includes = it is assumed that the `SortKey` is of type `sortAttributes` using Bib-1: in this case the integer before = is the attribute type and the integer following = is the attribute value. If no = character is in the criterion, it is treated as a sortfield of type InternationalString. The flags word of each sort specification must consist of `s` for case sensitive or `i` for case insensitive, and < for ascending order or > for descending order.

sort+ Same as `sort` but stores the sorted result set in a new result set.

authentication ***openauth*** Sets up an authentication string if a server requires authentication (v2 OpenStyle). The authentication string is first sent to the server when the open command is issued and the Z39.50 Initialize Request is sent, so this command must be used before `open` in order to be effective. A common convention for the *authopen* string is that the username - and password is separated by a slash, e.g. `myusername/mysecret`.

sru ***method version*** Selects Web Service method and version. Must be one of `post`, `get`, `soap` (default) or `solr`. Version should be either 1.1, 1.2 or 2.0 for SRU. Other versions are allowed - for testing purposes (version negotiation with SRU server). The version is currently not used for Solr Web Services

list_all This command displays status and values for many settings.

lslb ***n*** Sets the limit for when no records should be returned together with the search result. See the Z39.50 standard on set bounds for more details.

ssub ***n*** Sets the limit for when all records should be returned with the search result. See the Z39.50 standard on set bounds for more details.

mspn ***n*** Sets the number of records that should be returned if the number of records in the result set is between the values of `lslb` and `ssub`. See the Z39.50 standard on set bounds for more details.

status Displays the values of `lslb`, `ssub` and `mspn`.

setname Switches named result sets on and off. Default is on.

cancel Sends a Trigger Resource Control Request to the target.

facets ***spec*** Specifies requested facets to be used in search. The notation is specified in Section 7.8.

format ***oid*** Sets the preferred transfer syntax for retrieved records. yaz-client supports all the record syntaxes that currently are registered. See Z39.50 Record Syntax Identifiers for more details. Commonly used records syntaxes include usmarc, sutrs and xml.

elements ***e*** Sets the element set name for the records. Many targets support element sets B (for brief) and F (for full).

close Sends a Z39.50 Close APDU and closes connection with the peer

querytype ***type*** Sets the query type as used by command find. The following is supported: `prefix` for Prefix Query Notation (Type-1 Query); `ccl` for CCL search (Type-2 Query), `cql` for CQL (Type-104 search with CQL OID), `ccl2rpn` for CCL to RPN conversion (Type-1 Query), `cql2rpn` for CQL to RPN conversion (Type-1 Query).

attributeset ***set*** Sets attribute set OID for prefix queries (RPN, Type-1).

refid ***id*** Sets reference ID for Z39.50 Request(s).

itemorder ***type no*** Sends an Item Order Request using the ILL External. *`type`* is either 1 or 2 which corresponds to ILL-Profile 1 and 2 respectively. The *`no`* is the Result Set position of the record to be ordered.

update ***action recid doc*** Sends Item Update Request. The *`action`* argument must be the action type: one of `insert`, `replace`, `delete` and `update`. The second argument, *`recid`*, is the record identifier (any string). Third argument which is optional is the record document for the request. If doc is preceded with "<", then the following characters are treated as a filename with the records to be updated. Otherwise doc is treated as a document itself. The doc may also be quoted in double quotes. If doc is omitted, the last received record (as part of present response or piggybacked search response) is used for the update.

source *filename* Executes list of commands from file *filename*, just like 'source' on most UNIX shells. A single dot (`.`) can be used as an alternative.

! *args* Executes command *args* in subshell using the `system` call.

push_command *command* The push_command takes another command as its argument. That command is then added to the history information (so you can retrieve it later). The command itself is not executed. This command only works if you have GNU readline/history enabled.

set_apdufile *filename* Sets that APDU should be logged to file *filename*. Another way to achieve APDU log is by using command-line option `-a`.

set_auto_reconnect *flag* Specifies whether YAZ client automatically reconnects if the target closes connection (Z39.50 only).

flag must be either `on` or `off`.

set_auto_wait *flag* Specifies whether YAZ client should wait for response protocol packages after a request. By default YAZ client waits (`on`) for response packages immediately after a command (find, show) has been issued. If `off` is used, YAZ client does not attempt to receive packages automatically. These will have to be manually received when command wait_response is used.

flag must be either `on` or `off`.

set_marcdump *filename* Specifies that all retrieved records should be appended to file *filename*. This command does the thing as option `-m`.

schema *schemaid* Specifies schema for retrieval. Schema may be specified as an OID for Z39.50. For SRU, schema is a simple string URI.

charset *negotiationcharset* [*displaycharset*] [[*marccharset*]] Specifies character set (encoding) for Z39.50 negotiation / SRU encoding and/or character set for output (terminal).

negotiationcharset is the name of the character set to be negotiated by the server. The special name – for *negotiationcharset* specifies *no* character set to be negotiated.

If *displaycharset* is given, it specifies name of the character set of the output (on the terminal on which YAZ client is running). To disable conversion of characters to the output encoding, the special name – (dash) can be used. If the special name `auto` is given, YAZ client will convert strings to the encoding of the terminal as returned by `nl_langinfo` call.

If *marccharset* is given, it specifies name of the character set of retrieved MARC records from server. See also `marccharset` command.

Note

Since character set negotiation takes effect in the Z39.50 Initialize Request you should issue this command before command open is used.

Note

MARC records are not covered by Z39.50 character set negotiation, so that's why there is a separate character that must be known in order to do meaningful conversion(s).

---

negcharset ***charset*** Specifies character set for negotiation (Z39.50). The argument is the same as second argument for command charset.

displaycharset ***charset*** Specifies character set for output (display). The argument is the same as second argument for command charset.

marccharset ***charset*** Specifies character set for retrieved MARC records so that YAZ client can display them in a character suitable for your display. See `charset` command. If `auto` is given, YAZ will assume that MARC21/USMARC is using MARC8/UTF8 and ISO-8859-1 for all other MARC variants. The charset argument is the same as third argument for command charset.

querycharset ***charset*** Specifies character set for query terms for Z39.50 RPN queries and Z39.50 Scan Requests (termListAndStartPoint). This is a pure client-side conversion which converts from displayCharset to queryCharset.

set_cclfile ***filename*** Specifies that CCL fields should be read from file file *filename*. This command does the thing as option `-c`.

set_cqlfile ***filename*** Specifies that CQL fields should be read from file file *filename*. This command does the thing as option `-q`.

register_oid ***name class OID*** This command allows you to register your own object identifier - so that instead of entering a long dot-notation you can use a short name instead. The *name* is your name for the OID, *class* is the class, and *OID* is the raw OID in dot notation. Class is one of: `appctx`, `absyn`, `attet`, `transyn`, `diagset`, `recsyn`, `resform`, `accform`, `extserv`, `userinfo`, `elemspec`, `varset`, `schema`, `tagset`, `general`. If you're in doubt use the `general` class.

register_tab ***command string*** This command registers a TAB completion string for the command given.

sleep ***seconds*** This command makes YAZ client sleep (be idle) for the number of seconds given.

wait_response [***number***] This command makes YAZ client wait for a number of response packages from target. If *number* is omitted, 1 is assumed.

This command is rarely used and is only useful if command set_auto_wait is set to off.

xmles ***OID doc*** Sends XML Extended Services request using the OID and doc given.

zversion ***ver*** This command sets Z39.50 version for negotiation. Should be used before open. By default 3 (version 3) is used.

options ***op1 op2..*** This command sets Z39.50 options for negotiation. Should be used before open.

The following options are supported: `search`, `present`, `delSet`, `resourceReport`, `triggerResourceCtrl`, `resourceCtrl`, `accessCtrl`, `scan`, `sort`, `extendedServices`, `level_1Segmentation`, `level_2Segmentation`, `concurrentOperations`, `namedResultSets`, `encapsulation`, `resultCount`, `negotiationModel`, `duplicationDetection`, `queryType104`, `pQESCorrection`, `stringSchema`.

EXAMPLE

The simplest example of a Prefix Query would be something like

```
f knuth
```

or

```
f "donald knuth"
```

In those queries, no attributes were specified. This leaves it up to the server what fields to search but most servers will search in all fields. Some servers do not support this feature though, and require that some attributes are defined. To add one attribute you could do:

```
f @attr 1=4 computer
```

where we search in the title field, since the use(1) is title(4). If we want to search in the author field *and* in the title field, and in the title field using right truncation it could look something like this:

```
f @and @attr 1=1003 knuth @attr 1=4 @attr 5=1 computer
```

Finally using a mix of Bib-1 and GILS attributes could look something like this:

```
f @attrset Bib-1 @and @attr GILS 1=2008 Washington @attr 1=21 weather
```

FILES

`yaz-<version>/client/client.c`

`$HOME/.yazclientrc`

`$HOME/.yazclient.history`

SEE ALSO

yaz(7) bib1-attr(7)

yaz-ztest

yaz-ztest — Z39.50/SRU Test Server

Synopsis

`application` [`-install`] [`-installa`] [`-remove`] [`-a` *file*] [`-v` *level*] [`-l` *file*] [`-u` *uid*] [`-c` *config*] [`-f` *vconfig*] [`-C` *fname*] [`-t` *minutes*] [`-k` *kilobytes*] [`-K`] [`-d` *daemon*] [`-w` *dir*] [`-p` *pidfile*] [`-r` *kilobytes*] [`-ziDSTV1`] [listener-spec...]

DESCRIPTION

yaz-ztest is a Z39.50/SRU test server that uses the YAZ generic frontend server (GFS) API. The server acts as a real Z39.50/SRU server but does not use a database. It returns a random hit count and returns a subset of a few built-in records.

The *listener-spec* consists of a transport mode followed by a colon, followed by a listener address. The transport mode is either `tcp`, `unix`, or `ssl`.

For TCP and SSL, an address has the form:

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

For UNIX local socket, the address is the filename of the local socket.

OPTIONS

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

-S Don't fork or make threads on connection requests. This is good for debugging, but not recommended for real operation: Although the server is asynchronous and non-blocking, it can be nice to keep a software malfunction (okay then, a crash) from affecting all current users.

-1 Like `-S` but after one session the server exits. This mode is for debugging *only*.

-T Operate the server in threaded mode. The server creates a thread for each connection rather than fork a process. Only available on UNIX systems that offer POSIX threads.

-s Use the SR protocol (obsolete).

-z Use the Z39.50 protocol (default). This option and `-s` complement each other. You can use both multiple times on the same command line, between listener-specifications (see below). This way, you can set up the server to listen for connections in both protocols concurrently, on different local ports.

-l ***file*** The logfile.

-c ***config*** A user option that serves as a specifier for some sort of configuration, usually a filename. The argument to this option is transferred to member `configname` of the `statserv_options_block`.

-f ***vconfig*** This specifies an XML file that describes one or more YAZ frontend virtual servers.

-C ***fname*** Sets SSL certificate file name for server (PEM).

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

-u ***uid*** Set user ID. Sets the real UID of the server process to that of the given user. 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 *dir* The server changes to this directory before listening to incoming connections. This option is useful when the server is operating from the inetd daemon (see `-i`).

-p *pidfile* Specifies that the server should write its Process ID to the file given by *pidfile*. A typical location would be `/var/run/yaz-ztest.pid`.

-i Use this to make the the server run from the inetd server (UNIX only).

-D Use this to make the server put itself in the background and run as a daemon. If neither `-i` nor `-D` is given, the server starts in the foreground.

-install Use this to install the server as an NT service (Windows NT/2000/XP only). Control the server by going to the Services in the Control Panel.

-installa Use this to install the server as an NT service and mark it as "auto-start. Control the server by going to the Services in the Control Panel.

-remove Use this to remove the server from the NT services (Windows NT/2000/XP only).

-t *minutes* Idle session timeout, in minutes.

-k *size* Maximum record size/message size, in kilobytes.

-K Forces no-keepalive for HTTP sessions. By default GFS will keep sessions alive for HTTP 1.1 sessions (as defined by the standard). Using this option will force GFS to close the connection for each operation.

-r *size* Maximum size of log file before rotation occurs, in kilobytes. Default size is 1048576 k (=1 GB).

-d *daemon* Set name of daemon to be used in hosts access file. See hosts_access(5) and tcpd(8).

-m *time-format* Sets the format of time-stamps in the log-file. Specify a string in the input format to `strftime()`.

-V Display YAZ version and exit.

TESTING

yaz-ztest normally returns a random hit count between 0 and 24. However, if a query term includes leading digits, then the integer value of that term is used as hit count. This allows testers to return any number of hits. **yaz-ztest** includes 24 MARC records for testing. Hit counts exceeding 24 will make **yaz-ztest** return the same record batch over and over. So record at position 1, 25, 49, etc. are equivalent.

For XML, if no element set is given or element has value "marcxml", MARCXML is returned (each of the 24 dummy records converted from ISO2709 to XML). For element set OP, then OPAC XML is returned.

yaz-ztest may also return predefined XML records (for testing). This is enabled if `YAZ_ZTEST_XML_FETCH` environment variable is defined. A record is fetched from a file (one record per file). The path for the filename is *F*`E.`*d*`.xml` where *F* is the YAZ_ZTEST_XML_FETCH value (possibly empty), *E* is element-set, *d* is record position (starting from 1).

The following databases are honored by **yaz-ztest**: `Default`, `slow` and `db.*` (all databases with prefix "db"). Any other database will make **yaz-ztest** return diagnostic 109: "Database unavailable".

Options for search may be included in the form or URL get arguments included as part of the Z39.50 database name. The following database options are present: `search-delay`, `present-delay`, `fetch-delay` and `seed`.

The former, delay type options, specify a fake delay (sleep) that **yaz-ztest** will perform when searching, presenting, fetching records respectively. The value of the delay may either be a fixed floating point value which specifies the delay in seconds. Alternatively the value may be given as two floating point numbers separated by colon, which will make **yaz-ztest** perform a random sleep between the first and second number.

The database parameter `seed` takes an integer as value. This will call `srand` with this integer to ensure that the random behavior can be re-played.

Suppose we want searches to take between 0.1 and 0.5 seconds and a fetch to take 0.2 second. To access test database Default we'd use: `Default?search-delay=0.1:0.5&fetch-delay=0.2`.

GFS CONFIGURATION AND VIRTUAL HOSTS

The Virtual hosts mechanism allows a YAZ frontend server to support multiple backends. A backend is selected on the basis of the TCP/IP binding (port+listening adddress) and/or the virtual host.

A backend can be configured to execute in a particular working directory. Or the YAZ frontend may perform CQL to RPN conversion, thus allowing traditional Z39.50 backends to be offered as a SRW/SRU service. SRW/SRU Explain information for a particular backend may also be specified.

For the HTTP protocol, the virtual host is specified in the Host header. For the Z39.50 protocol, the virtual host is specified as in the Initialize Request in the OtherInfo, OID 1.2.840.10003.10.1000.81.1.

Note
Not all Z39.50 clients allow the VHOST information to be set. For those, the selection of the backend must rely on the TCP/IP information alone (port and address).

The YAZ frontend server uses XML to describe the backend configurations. Command-line option `-f` specifies filename of the XML configuration.

The configuration uses the root element `yazgfs`. This element includes a list of `listen` elements, followed by one or more `server` elements.

The `listen` describes listener (transport end point), such as TCP/IP, Unix file socket or SSL server. Content for a listener:

CDATA (required) The CDATA for the `listen` element holds the listener string, such as `tcp:@:210`, `tcp:server1:2100`, etc.

attribute `id` (optional) Identifier for this listener. This may be referred to from server sections.

Note
We expect more information to be added for the listen section in a future version, such as CERT file for SSL servers.

The `server` describes a server and the parameters for this server type. Content for a server:

attribute `id` (optional) Identifier for this server. Currently not used for anything, but it might be for logging purposes.

attribute `listenref` (optional) Specifies one or more listeners for this server. Each server ID is separated by a comma. If this attribute is not given, the server is accessible from all listeners. In order for the server to be used for real, however, the virtual host must match if specified in the configuration.

element `config` (optional) Specifies the server configuration. This is equivalent to the config specified using command line option `-c`.

element `directory` (optional) Specifies a working directory for this backend server. If specified, the YAZ frontend changes current working directory to this directory whenever a backend of this type is started (backend handler bend_start), stopped (backend handler hand_stop) and initialized (bend_init).

element `host` (optional) Specifies the virtual host for this server. If this is specified a client *must* specify this host string in order to use this backend.

element `cql2rpn` (optional) Specifies a filename that includes CQL to RPN conversion for this backend server. See Section 7.1.3.4. If given, the backend server will only "see" a Type-1/RPN query.

element `ccl2rpn` (optional) Specifies a filename that includes CCL to RPN conversion for this backend server. See Section 7.1.2.2. If given, the backend server will only "see" a Type-1/RPN query.

element `stylesheet` (optional) Specifies the stylesheet reference to be part of SRU HTTP responses when the client does not specify one. If none is given, then if the client does not specify one, then no stylesheet reference is part of the SRU HTTP response.

element `client_query_charset` (optional) If specified, a conversion from the character set given to UTF-8 is performed by the generic frontend server. It is only executed for Z39.50 search requests (SRU/Solr are assumed to be UTF-8 encoded already).

element `docpath` (optional) Specifies a path for local file access using HTTP. All URLs with a leading prefix (/ excluded) that matches the value of `docpath` are used for file access. For example, if the server is to offer access in directory `xsl`, the docpath would be `xsl` and all URLs of the form `http://host/xsl` will result in a local file access.

element `explain` (optional) Specifies SRW/SRU ZeeRex content for this server. Copied verbatim to the client. As things are now, some of the Explain content seem redundant because host information, etc. is also stored elsewhere.

element `maximumrecordsize` (optional) Specifies maximum record size/message size, in bytes. This value also serves as the maximum size of *incoming* packages (for Record Updates etc). It's the same value as that given by the `-k` option.

element `retrievalinfo` (optional) Enables the retrieval facility to support conversions and specifications of record formats/types. See Section 7.6 for more information.

The XML below configures a server that accepts connections from two ports, TCP/IP port 9900 and a local UNIX file socket. We name the TCP/IP server `public` and the other server `internal`.

```
<yazgfs>
 <listen id="public">tcp:@:9900</listen>
 <listen id="internal">unix:/var/tmp/socket</listen>
 <server id="server1">
   <host>server1.mydomain</host>
   <directory>/var/www/s1</directory>
   <config>config.cfg</config>
 </server>
 <server id="server2" listenref="public,internal">
   <host>server2.mydomain</host>
   <directory>/var/www/s2</directory>
   <config>config.cfg</config>
   <cql2rpn>../etc/pqf.properties</cql2rpn>
   <explain xmlns="http://explain.z3950.org/dtd/2.0/">
     <serverInfo>
       <host>server2.mydomain</host>
       <port>9900</port>
       <database>a</database>
     </serverInfo>
   </explain>
 </server>
 <server id="server3" listenref="internal">
   <directory>/var/www/s3</directory>
   <config>config.cfg</config>
 </server>
</yazgfs>
```

There are three configured backend servers. The first two servers, "server1" and "server2", can be reached by both listener addresses. "server1" is reached by all (two) since no listenref attribute is specified. "server2" is reached by the two listeners specified. In order to distinguish between the two, a virtual host has been specified for each server in the host elements.

For "server2" elements for CQL to RPN conversion is supported and explain information has been added (a short one here to keep the example small).

The third server, "server3" can only be reached via listener "internal".

FILES

yaz-<version>/ztest/yaz-ztest.c

yaz-<version>/include/yaz/backend.h

SEE ALSO

yaz(7) yaz-log(7)

yaz-config

yaz-config — Script to get information about YAZ.

Synopsis

`yaz-config` [`--prefix[=`*`DIR`*`]`] [`--version`] [`--libs`] [`--lalibs`] [`--cflags`] [`--include`] [`--comp`] [`-V`] [libraries...]

DESCRIPTION

yaz-config is a script that returns information that your own software should use to build software that uses YAZ.

The following libraries are supported:

threads Use the threaded version of YAZ.

OPTIONS

--prefix[=*DIR*] Returns prefix of YAZ or assume a different one if DIR is specified.

--version Returns version of YAZ.

--libs Library specification be used when using YAZ.

--lalibs Return library specification.

--cflags Return C Compiler flags.

--include Return C compiler includes for YAZ header files (-Ipath).

--comp Returns full path to YAZ' ASN.1 compiler: yaz-asncomp.

-V Returns YAZ SHA1 ID (from Git) and version.

FILES

`/usr/bin/yaz-config`

`/usr/lib/libyaz*.a`

`/usr/include/yaz/*.h`

SEE ALSO

yaz(7)

Section "How to make apps using YAZ on UNIX" in the YAZ manual.

yaz

yaz — Z39.50 toolkit.

DESCRIPTION

YAZ is a C/C++ programmer's toolkit supporting the development of Z39.50v3 clients and servers. The YAZ toolkit offers several different levels of access to the ISO23950/Z39.50, SRU Solr (client only) and ILL protocols. The level that you need to use depends on your requirements, and the role (server or client) that you want to implement.

COPYRIGHT

SEE ALSO

yaz-client(1), yaz-ztest(8), yaz-config(8), zoomsh(1) bib1-attr(7)

YAZ manual (`/usr/share/doc/yaz`)

YAZ home page.

Z39.50 Maintenance Agency Page.

zoomsh

zoomsh — ZOOM shell

Synopsis

`zoomsh` [`-a` *`apdufile`*] [`-e`] [`-v` *`loglevel`*] [commands...]

DESCRIPTION

zoomsh is a ZOOM client with a simple command line interface. The client demonstrates the ZOOM API and is useful for testing targets.

You may pass one or more commands to **zoomsh**. These commands are invoked first.

OPTIONS

-a *`apdufile`* Logs protocol packages into apdufile (APDU log).

-e Makes zoomsh stop processing commands as soon as an error occur. The exit code of zoomsh is 1 if error occurs; 0 otherwise.

-v *`loglevel`* Sets YAZ log level to *`loglevel`*.

EXAMPLES

If you start the **yaz-ztest** in one console you can use the ZOOM shell as follows:

```
$ zoomsh
ZOOM>connect localhost:9999
ZOOM>search computer
localhost:9999: 7 hits
ZOOM>show 0 1
1 Default USmarc
001    11224466
003 DLC
005 00000000000000.0
008 910710c19910701nju           00010 eng
010    $a    11224466
040    $a DLC $c DLC
050 00 $a 123-xyz
100 10 $a Jack Collins
245 10 $a How to program a computer
260 1  $a Penguin
263    $a 8710
300    $a p. cm.
```

```
ZOOM>quit
```

You can also achieve the same result by passing the commands as arguments on a single command line:

```
$ zoomsh "connect localhost:9999" "search computer" "show 0 1" quit
```

COMMANDS

connect ***zurl*** Connects to the target given by *zurl*.

close [***zurl***] Closes connection to target given by *zurl* or all targets if *zurl* was omitted.

show [***start*** [***count***]] Displays count records starting at offset given by *start*. First records has offset 0 (unlike the Z39.50 protocol).

quit Quits **zoomsh**.

set ***name*** [***value***] Sets option *name* to *value*.

get ***name*** Prints value of option *name*.

help Prints list of available commands.

SEE ALSO

yaz(7), yaz-ztest(8),

Section "Building clients with ZOOM" in the YAZ manual.

ZOOM home page.

yaz-asncomp

yaz-asncomp — YAZ ASN.1 compiler

Synopsis

yaz-asncomp [-v] [-c *cfile*] [-h *hfile*] [-p *pfile*] [-d *config*] [-I *includeout*] [-i *includedir*] [-m *module*] [filename]

DESCRIPTION

yaz-asncomp is an ASN.1 compiler that reads an ASN.1 specification in *filename* and produces C/C++ definitions and BER encoders/decoders for it.

The produced C/C++ code and header files uses the ODR module of YAZ which is a library that encodes/decodes/prints BER packages. **yaz-asncomp** allows you to specify name of resulting source via options. Alternatively, you can specify a DEFINITIONS file, which provides customized output to many output files - if the ASN.1 specification file consists of many modules.

This utility is written in Tcl. Any version of Tcl should work.

OPTIONS

-v Makes the ASN.1 compiler print more verbose about the various stages of operations.

-c ***cfile*** Specifies the name of the C/C++ file with encoders/decoders.

-h ***hfile*** Specifies the name of header file with definitions.

-p ***pfile*** Specifies the name of the a private header file with definitions. By default all definitions are put in header file (option -h).

-d ***dfile*** Specifies the name of a definitions file.

-I ***iout*** Specifies first part of directory in which header files are written.

-i ***idir*** Specifies second part of directory in which header files are written.

-m ***module*** Specifies that ASN.1 compiler should only process the module given. If this option is not specified, all modules in the ASN.1 file are processed.

DEFINITIONS FILE

The definitions file is really a Tcl script but follows traditional rules for Shell like configuration files. That is # denotes the beginning of a comment. Definitions are line oriented. The definitions files usually consist of a series of variable assignments of the form:

`set` *name value*

Available variables are:

default-prefix Sets prefix for names in the produced output. The value consists of three tokens: C function prefix, C typedef prefix and preprocessor prefix respectively.

prefix(*module*) This value sets prefix values for module *module*. The value has same form as `default-prefix`.

filename(*module*) Specifies filename for C/header file for module *module*.

init(*module*,h) Code fragment to be put in first part of public header for module *module*.

body(*module*,h) Code fragment to be put in last part of public header for module *module* (trailer).

init(*module*,c) Code fragment to be put in first part of C based encoder/decoder for module *module*.

body(*module*,c) Code fragment to be put in last part of C based encoder/decoder for module *module* (trailer).

map(*module*,*name*) Maps ASN.1 type in module *module* of *name* to value.

membermap(*module*,*name*,*member*) Maps member *member* in SEQUENCE/CHOICE of *name* in module *module* to value. The value consists of one or two tokens. First token is name of C preprocessor part. Second token is resulting C member name. If second token is omitted the value (one token) is both preprocessor part and C `struct`,`union`.

unionmap(*module*,*name*,*member*) Maps member *member* in CHOICE of *name* in module *module* to value. Value consists of two or three tokens. The first token is name of the integer in the `union` that is used as selector for the `union` itself. The second token is name of the `union`. The third token overrides the name of the CHOICE member; if omitted the member name is used.

FILES

`/usr/share/yaz/z39.50/z.tcl`

`/usr/share/yaz/z39.50/*.asn`

SEE ALSO

yaz(7)

Section "The ODR Module" in the YAZ manual.

yaz-marcdump

yaz-marcdump — MARC record dump utility

Synopsis

`yaz-marcdump` [`-i` *format*] [`-o` *format*] [`-f` *from*] [`-t` *to*] [`-l` *spec*] [`-c` *cfile*] [`-s` *prefix*] [`-C` *size*] [`-n`] [`-p`] [`-v`] [`-V`] [file...]

DESCRIPTION

yaz-marcdump reads MARC records from one or more files. It parses each record and supports output in line-format, ISO2709, MARCXML, MARC-in-JSON, MarcXchange as well as Hex output.

This utility parses records ISO2709(raw MARC), line format, MARC-in-JSON format as well as XML if that is structured as MARCXML/MarcXchange.

MARC-in-JSON encoding/decoding is supported in YAZ 5.0.5 and later.

Note
As of YAZ 2.1.18, OAI-MARC is no longer supported. OAI-MARC is deprecated. Use MARCXML instead.

By default, each record is written to standard output in a line format with newline for each field, $x for each subfield x. The output format may be changed with option `-o`,

yaz-marcdump can also be requested to perform character set conversion of each record.

OPTIONS

-i *format* Specifies input format. Must be one of `marcxml`, `marc` (ISO2709), `marcxchange` (ISO25577), `line` (line mode MARC), `turbomarc` (Turbo MARC), or `json` (MARC-in-JSON).

-o *format* Specifies output format. Must be one of `marcxml`, `marc` (ISO2709), `marcxchange` (ISO25577), `line` (line mode MARC), `turbomarc` (Turbo MARC), or `json` (MARC-in-JSON).

-f *from* Specify the character set of the input MARC record. Should be used in conjunction with option `-t`. Refer to the yaz-iconv man page for supported character sets.

-t *to* Specify the character set of the output. Should be used in conjunction with option `-f`. Refer to the yaz-iconv man page for supported character sets.

-l *leaderspec* Specify a simple modification string for MARC leader. The *leaderspec* is a list of pos=value pairs, where pos is an integer offset (0 - 23) for leader. Value is either a quoted string or an integer (character value in decimal). Pairs are comma separated. For example, to set leader at offset 9 to a, use `9='a'`.

-s *prefix* Writes a chunk of records to a separate file with prefix given, i.e. splits a record batch into files with only at most "chunk" ISO2709 records per file. By default chunk is 1 (one record per file). See option `-C`.

-C *chunksize* Specifies chunk size; to be used conjunction with option `-s`.

-p Makes yaz-marcdump print record number and input file offset of each record read.

-n MARC output is omitted so that MARC input is only checked.

-v Writes more information about the parsing process. Useful if you have ill-formatted ISO2709 records as input.

-V Prints YAZ version.

EXAMPLES

The following command converts MARC21/USMARC in MARC-8 encoding to MARC21/USMARC in UTF-8 encoding. Leader offset 9 is set to 'a'. Both input and output records are ISO2709 encoded.

```
yaz-marcdump -f MARC-8 -t UTF-8 -o marc -l 9=97 marc21.raw >marc21.utf8 ↩
   .raw
```

The same records may be converted to MARCXML instead in UTF-8:

```
yaz-marcdump -f MARC-8 -t UTF-8 -o marcxml marc21.raw >marcxml.xml
```

Turbo MARC is a compact XML notation with same semantics as MARCXML, but which allows for faster processing via XSLT. In order to generate Turbo MARC records encoded in UTF-8 from MARC21 (ISO), one could use:

```
yaz-marcdump -f MARC8 -t UTF8 -o turbomarc -i marc marc21.raw >out.xml
```

FILES

prefix/bin/yaz-marcdump

prefix/include/yaz/marcdisp.h

SEE ALSO

yaz(7)

yaz-iconv(1)

yaz-iconv

yaz-iconv — YAZ Character set conversion utility

Synopsis

yaz-iconv [-f *from*] [-t *to*] [-v] [file...]

DESCRIPTION

yaz-iconv converts data in the character set specified by *from* to output in the character set as specified by *to*.

This **yaz-iconv** utility is similar to the **iconv** found on many POSIX systems (Glibc, Solaris, etc).

If no *file* is specified, **yaz-iconv** reads from standard input.

OPTIONS

-f*from*] Specify the character set *from* of the input file. Should be used in conjunction with option `-t`.

-t*to*] Specify the character set *of* of the output. Should be used in conjunction with option `-f`.

-v Print more information about the conversion process.

ENCODINGS

The yaz-iconv command and the API as defined in `yaz/yaz-iconv.h` is a wrapper for the library system call iconv. But YAZ' iconv utility also implements conversions on its own. The table below lists characters sets (or encodings) that are supported by YAZ. Each character set is marked with either *encode* or *decode*. If an encoding is encode-enabled, YAZ may convert *to* the designated encoding. If an encoding is decode-enabled, YAZ may convert *from* the designated encoding.

marc8 (encode, decode) The MARC8 encoding as defined by the Library of Congress. Most MARC21/USMAR records use this encoding.

marc8s (encode, decode) Like MARC8 but conversion prefers non-combined characters in the Latin-1 plane over combined characters.

marc8lossy (encode) Lossy encoding of MARC-8.

marc8lossless (encode) Lossless encoding of MARC8.

utf8 (encode, decode) The most commonly used UNICODE encoding on the Internet.

iso8859-1 (encode, decode) ISO-8859-1, AKA Latin-1.

iso5426 (decode) ISO 5426. Some MARC records (UNIMARC) use this encoding.

iso5428:1984 (encode, decode) ISO 5428:1984.

advancegreek (encode, decode) An encoding for Greek in use by some vendors (Advance).

danmarc (decode) Danmarc (in danish) is an encoding based on UNICODE which is used for DanMARC2 records.

EXAMPLES

The following command converts from ISO-8859-1 (Latin-1) to UTF-8.

```
yaz-iconv -f ISO-8859-1 -t UTF-8 <input.lst >output.lst
```

FILES

prefix/bin/yaz-iconv

prefix/include/yaz/yaz-iconv.h

SEE ALSO

yaz(7) iconv(1)

yaz-log

yaz-log — Log handling in all yaz-based programs

Synopsis

`yaz-XXXX` [`-v` *`loglevel,...`*] [`-l` *`logfile`*]

DESCRIPTION

All YAZ-based programs use a common log subsystem, and should support common command line options for controlling it. This man page documents those.

OPTIONS

-l ***`logfile`*** Specify the file where the log is to be written. If none is specified, `stderr` is used. The log is appended to this file. If the file grows overly large, it is silently rotated: It is renamed to *`logfile`*.1, *`logfile`*.2, .., 9 (old such file is deleted), and a new file is opened. The limit defaults to 1GB, but can be set by the program. The rotating limit can be specified with option `-r` for the YAZ frontend server (yaz-ztest).

Rotation can also be implicitly enabled by using a filename which gets changed for a given date, due to substitutions as given by the strftime(3) function.

-v ***`loglevel`*** Specify the logging level. The argument is a set of log level names, separated by commas (no whitespace!), optionally preceded by a '-' to negate that level. Most programs have their own default, often containing `fatal,warn,log`, and some application-specific values. The default list can be cleared with the word `none`, or individual bits can be removed by prefixing them with a dash '-'.

LOG LEVELS TO CONTROL LOGGING

Some of the log levels control the way the log is written.

`flush` causes the log to be flushed after every write. This can have serious implications to performance, and should not be used in production. On the other hand, when debugging a program crash, this can be extremely useful. The option `debug` implies `flush` as well.

`notime` prevents the writing of time stamps. This is intended for automatic test scripts, which should produce predictable log files that are easy to compare.

GENERAL LOG LEVELS IN YAZ ITSELF

YAZ itself uses the following log levels:

`fatal` for fatal errors, that prevent further execution of the program.

`warn` for warnings about things that should be corrected.

`debug` for debugging. This flag may be used temporarily when developing or debugging yaz, or a program that uses yaz. It is practically deprecated, you should be defining and using your own log levels (see below).

`all` turns on almost all hard-coded log levels.

`loglevel` logs information about the log levels used by the program. Every time the log level is changed, lists all bits that are on. Every time a module asks for its log bits, this is logged. This can be used for getting an idea of what log levels are available in any program that uses yaz-log. Start the program with `-v none,loglevel`, and do some common operations with it. Another way is to grep for `yaz_log_module_level` in the source code, as in

```
find . -name '*.[ch]' -print |
   xargs grep yaz_log_module_level |
   grep '"' |
   cut -d'"' -f2 |
   sort -u
```

`eventl`, `malloc`, `nmem`, `odr` are used internally for debugging yaz.

LOG LEVELS FOR CLIENTS

`zoom` logs the calls to the zoom API, which may be useful in debugging client applications.

LOG LEVELS FOR SERVERS

`server` logs the server functions on a high level, starting up, listening on a port, etc.

`session` logs individual sessions (connections).

`request` logs a one-liner for each request (init, search, etc.).

`requestdetail` logs the details of every request, before it is passed to the back-end, and the results received from it.

Each server program (zebra, etc.) is supposed to define its own log levels in addition to these. As they depend on the server in question, they can not be described here. See above how to find out about them.

LOGGING EXAMPLES

See what log levels yaz-ztest is using:

```
yaz-ztest -l -v none,loglevel
14:43:29-23/11 [loglevel] Setting log level to 4096 = 0x00001000
14:43:29-23/11 [loglevel] Static  log bit 00000001 'fatal' is off
14:43:29-23/11 [loglevel] Static  log bit 00000002 'debug' is off
14:43:29-23/11 [loglevel] Static  log bit 00000004 'warn' is off
14:43:29-23/11 [loglevel] Static  log bit 00000008 'log' is off
14:43:29-23/11 [loglevel] Static  log bit 00000080 'malloc' is off
14:43:29-23/11 [loglevel] Static  log bit 00000800 'flush' is off
14:43:29-23/11 [loglevel] Static  log bit 00001000 'loglevel' is ON
14:43:29-23/11 [loglevel] Static  log bit 00002000 'server' is off
14:43:29-23/11 [loglevel] Dynamic log bit 00004000 'session' is off
14:43:29-23/11 [loglevel] Dynamic log bit 00008000 'request' is off
14:44:13-23/11 yaz-ztest [loglevel] returning log bit 0x4000 for ' ↩
   session'
14:44:13-23/11 yaz-ztest [loglevel] returning log bit 0x2000 for ' ↩
   server'
14:44:13-23/11 yaz-ztest [loglevel] returning NO log bit for 'eventl'
14:44:20-23/11 yaz-ztest [loglevel] returning log bit 0x4000 for ' ↩
   session'
14:44:20-23/11 yaz-ztest [loglevel] returning log bit 0x8000 for ' ↩
   request'
14:44:20-23/11 yaz-ztest [loglevel] returning NO log bit for ' ↩
   requestdetail'
14:44:20-23/11 yaz-ztest [loglevel] returning NO log bit for 'odr'
14:44:20-23/11 yaz-ztest [loglevel] returning NO log bit for 'ztest'
```

See the details of the requests for yaz-ztest

```
./yaz-ztest -l -v requestdetail
14:45:35-23/11 yaz-ztest [server] Adding static Z3950 listener on tcp:@ ↩
   :9999
14:45:35-23/11 yaz-ztest [server] Starting server ./yaz-ztest pid=32200
14:45:38-23/11 yaz-ztest [session] Starting session from tcp:127.0.0.1 ( ↩
   pid=32200)
14:45:38-23/11 yaz-ztest [requestdetail] Got initRequest
14:45:38-23/11 yaz-ztest [requestdetail] Id:        81
14:45:38-23/11 yaz-ztest [requestdetail] Name:      YAZ
14:45:38-23/11 yaz-ztest [requestdetail] Version:   2.0.28
14:45:38-23/11 yaz-ztest [requestdetail] Negotiated to v3: srch prst del ↩
    extendedServices namedresults scan sort
14:45:38-23/11 yaz-ztest [request] Init from 'YAZ' (81) (ver 2.0.28) OK
14:45:39-23/11 yaz-ztest [requestdetail] Got SearchRequest.
14:45:39-23/11 yaz-ztest [requestdetail] ResultSet '1'
14:45:39-23/11 yaz-ztest [requestdetail] Database 'Default'
14:45:39-23/11 yaz-ztest [requestdetail] RPN query. Type: Bib-1
14:45:39-23/11 yaz-ztest [requestdetail]  term 'foo' (general)
14:45:39-23/11 yaz-ztest [requestdetail] resultCount: 7
14:45:39-23/11 yaz-ztest [request] Search Z: @attrset Bib-1 foo  OK:7  ↩
   hits
14:45:41-23/11 yaz-ztest [requestdetail] Got PresentRequest.
```

```
14:45:41-23/11 yaz-ztest [requestdetail] Request to pack 1+1 1
14:45:41-23/11 yaz-ztest [requestdetail] pms=1048576, mrs=1048576
14:45:41-23/11 yaz-ztest [request] Present: [1] 1+1  OK 1 records  ↩
   returned
```

LOG FILENAME EXAMPLES

A file with format my_YYYYMMDD.log (where Y, M, D is year, month, and day digits) is given as follows: `-l my_%Y%m%d.log` . And since the filename is depending on day, rotation will occur on midnight.

A weekly log could be specified as `-l my_%Y%U.log`.

FILES

prefix`/include/yaz/log.h` *prefix*`/src/log.c`

SEE ALSO

yaz(7) yaz-ztest(8) yaz-client(1) strftime(3)

yaz-illclient

yaz-illclient — ILL client

Synopsis

`yaz-illclient` [`-f` *filename*] [`-v` *loglevel*] [`-D`*name=value*...] [`-o`] [`-u` *user*] [`-p` *password*] [`-V`] [server-addr]

DESCRIPTION

yaz-illclient is a client which sends an ISO ILL request to a remote server and decodes the response from it. Exactly one server address (*server-addr*) must be specified.

OPTIONS

-f *filename*] Specify filename.

-v *loglevel*] Specify the log level.

-D *name=value*] Defines name & value pair.

-o Enable OCLC authentication.

-u *user*] Specify user.

-p *password*] Specify password.

-V Show yaz-illclient version.

EXAMPLES

None yet.

FILES

None yet.

SEE ALSO

yaz(7)

yaz-icu

yaz-icu — YAZ ICU utility

Synopsis

`yaz-icu` [-c *config*] [-p *opt*] [-s] [-x] [infile]

DESCRIPTION

yaz-icu is a utility which demonstrates the ICU chain module of yaz. (`yaz/icu.h`).

The utility can be used in two ways. It may read some text using an XML configuration for configuring ICU and show text analysis. This mode is triggered by option `-c` which specifies the configuration to be used. The input file is read from standard input or from a file if `infile` is specified.

The utility may also show ICU information. This is triggered by option `-p`.

OPTIONS

-c *config* Specifies the file containing ICU chain configuration which is XML based.

-p *type* Specifies extra information to be printed about the ICU system. If *type* is `c` then ICU converters are printed. If *type* is `l`, then available locales are printed. If *type* is `t`, then available transliterators are printed.

-s Specifies that output should include sort key as well. Note that sort key differs between ICU versions.

-x Specifies that output should be XML based rather than "text" based.

ICU chain configuration

The ICU chain configuration specifies one or more rules to convert text data into tokens. The configuration format is XML based.

The toplevel element must be named `icu_chain`. The `icu_chain` element has one required attribute `locale` which specifies the ICU locale to be used in the conversion steps.

The `icu_chain` element must include elements where each element specifies a conversion step. The conversion is performed in the order in which the conversion steps are specified. Each conversion element takes one attribute: `rule` which serves as argument to the conversion step.

The following conversion elements are available:

casemap Converts case (and rule specifies how):

- **l** Lower case using ICU function u_strToLower.
- **u** Upper case using ICU function u_strToUpper.
- **t** To title using ICU function u_strToTitle.
- **f** Fold case using ICU function u_strFoldCase.

display This is a meta step which specifies that a term/token is to be displayed. This term is retrieved in an application using function icu_chain_token_display (`yaz/icu.h`).

transform Specifies an ICU transform rule using a transliterator Identifier. The rule attribute is the transliterator Identifier. See ICU Transforms for more information.

transliterate Specifies a rule-based transliterator. The rule attribute is the custom transformation rule to be used. See ICU Transforms for more information.

tokenize Breaks / tokenizes a string into components using ICU functions ubrk_open, ubrk_setText, .. . The rule is one of:

- **l** Line. ICU: UBRK_LINE.
- **s** Sentence. ICU: UBRK_SENTENCE.
- **w** Word. ICU: UBRK_WORD.

c Character. ICU: UBRK_CHARACTER.

t Title. ICU: UBRK_TITLE.

join Joins tokens into one string. The rule attribute is the joining string, which may be empty. The join conversion element was added in YAZ 4.2.49.

EXAMPLES

The following command analyzes text in file `text` using ICU chain configuration `chain.xml`:

```
cat text | yaz-icu -c chain.xml
```

The chain.xml might look as follows:

```
<icu_chain locale="en">
  <transform rule="[:Control:] Any-Remove"/>
  <tokenize rule="w"/>
  <transform rule="[[:WhiteSpace:][:Punctuation:]] Remove"/>
  <transliterate rule="xy > z;"/>
  <display/>
  <casemap rule="l"/>
</icu_chain>
```

SEE ALSO

yaz(7)

ICU Home

ICU Transforms

yaz-url

yaz-url — YAZ URL fetch utility

Synopsis

`yaz-url` [-H *name:value*] [-m *method*] [-O *fname*] [-p *fname*] [-R *num*] [-u *user/password*] [-v] [-x *proxy*] [url...]

DESCRIPTION

yaz-url is utility to get web content. It is very limited in functionality compared to programs such as curl, wget.

The options must precede the URL given on the command line, to take effect.

Fetched HTTP content is written to stdout, unless option -O is given.

OPTIONS

-H *name:value* Specifies HTTP header content with name and value. This option can be given multiple times (for different names, of course).

-m *method* Specifies the HTTP method to be used for the next URL. Default is method "GET". However, option -p sets it to "POST".

-O *fname* Sets output filename for HTTP content.

-p *fname* Sets a file to be POSTed in the folloing URL.

-R *num* Sets maximum number of HTTP redirects to be followed. A value of zero disables follow of HTTP redirects.

-u *user/password* Specifies a user and a password to be uesd in HTTP basic authentication in the following URL fetch. The user and password must be separated by a slash (thus it is not possible to specify a user with a slash in it).

-v Makes yaz-url dump each HTTP request/response to stdout.

-x *proxy* Specifies a proxy to be used for URL fetch.

SEE ALSO

yaz(7)

Bib-1 Attribute Set

bib1-attr — Bib-1 Attribute Set

DESCRIPTION

This reference entry lists the Bib-1 attribute set types and values.

TYPES

The Bib-1 attribute set defines six attribute types: Use (1), Relation (2), Position (3), Structure (4), Truncation (5) and Completeness (6).

USE (1)

```
1     Personal-name
2     Corporate-name
3     Conference-name
4     Title
5     Title-series
6     Title-uniform
7     ISBN
8     ISSN
9     LC-card-number
10    BNB-card-number
11    BGF-number
12    Local-number
13    Dewey-classification
14    UDC-classification
15    Bliss-classification
16    LC-call-number
17    NLM-call-number
18    NAL-call-number
19    MOS-call-number
20    Local-classification
21    Subject-heading
22    Subject-Rameau
23    BDI-index-subject
24    INSPEC-subject
25    MESH-subject
26    PA-subject
27    LC-subject-heading
28    RVM-subject-heading
29    Local-subject-index
30    Date
31    Date-of-publication
32    Date-of-acquisition
33    Title-key
34    Title-collective
35    Title-parallel
36    Title-cover
37    Title-added-title-page
38    Title-caption
39    Title-running
40    Title-spine
41    Title-other-variant
42    Title-former
43    Title-abbreviated
44    Title-expanded
45    Subject-precis
46    Subject-rswk
47    Subject-subdivision
48    Number-natl-biblio
```

```
    49    Number-legal-deposit
    50    Number-govt-pub
    51    Number-music-publisher
    52    Number-db
    53    Number-local-call
    54    Code-language
    55    Code-geographic
    56    Code-institution
    57    Name-and-title
    58    Name-geographic
    59    Place-publication
    60    CODEN
    61    Microform-generation
    62    Abstract
    63    Note
    1000  Author-title
    1001  Record-type
    1002  Name
    1003  Author
    1004  Author-name-personal
    1005  Author-name-corporate
    1006  Author-name-conference
    1007  Identifier-standard
    1008  Subject-LC-childrens
    1009  Subject-name-personal
    1010  Body-of-text
    1011  Date/time-added-to-db
    1012  Date/time-last-modified
    1013  Authority/format-id
    1014  Concept-text
    1015  Concept-reference
    1016  Any
    1017  Server-choice
    1018  Publisher
    1019  Record-source
    1020  Editor
    1021  Bib-level
    1022  Geographic-class
    1023  Indexed-by
    1024  Map-scale
    1025  Music-key
    1026  Related-periodical
    1027  Report-number
    1028  Stock-number
    1030  Thematic-number
    1031  Material-type
    1032  Doc-id
    1033  Host-item
    1034  Content-type
    1035  Anywhere
```

```
1036  Author-Title-Subject
```

RELATION (2)

```
1 Less than
2 Less than or equal
3 Equal
4 Greater or equal
5 Greater than
6 Not equal
100 Phonetic
101 Stem
102 Relevance
103 AlwaysMatches
```

POSITION (3)

```
1 First in field
2 First in subfield
3 Any position in field
```

STRUCTURE (4)

```
1 Phrase
2 Word
3 Key
4 Year
5 Date (normalized)
6 Word list
100 Date (un-normalized)
101 Name (normalized)
102 Name (un-normalized)
103 Structure
104 Urx
105 Free-form-text
106 Document-text
107 Local-number
108 String
109 Numeric-string
```

TRUNCATION (5)

```
1 Right truncation
2 Left truncation
3 Left and right truncation
100 Do not truncate
101 Process # in search term  . regular #=.*
102 RegExpr-1
103 RegExpr-2
104 Process # ?n . regular: #=., ?n=.{0,n} or ?=.* Z39.58
```

Thw 105-106 truncation attributes below are only supported by Index Data's Zebra server.

```
105 Process * ! regular: *=.*, !=. and right truncate
106 Process * ! regular: *=.*, !=.
```

COMPLETENESS (6)

```
1 Incomplete subfield
2 Complete subfield
3 Complete field
```

SORTING (7)

```
1 ascending
2 descending
```

Type 7 is an Index Data extension to RPN queries that allows embedding a sort critieria into a query.

SEE ALSO

Bib-1 Attribute Set

Attibute Set Bib-1 Semantics.

yaz-json-parse

yaz-json-parse — YAZ JSON parser

Synopsis

`yaz-json-parse` [-p]

DESCRIPTION

yaz-json-parse is a utility which demonstrates the JSON API of YAZ. (`yaz/json.h`).

The program attempts to parse a JSON from standard input (stdin). It will return exit code 1 if parsing fails and the parsing error message will be printed to standard error (stderr). The program returns exit code 0 if parsing succeeds, and returns no output unless -p is given (see below).

OPTIONS

-p Makes the JSON parser echo the JSON result string to standard output, if parsing from stdin was successful. If -p is given twice, then the output is a multi-line output with indentation (pretty print).

SEE ALSO

yaz(7)

yaz-record-iconv

yaz-record-conv — YAZ Record Conversion Utility

Synopsis

`yaz-record-conv` [`-v` *`loglevel`*] [*`config`*] [*`fname`*...]

DESCRIPTION

yaz-record-conv is a program that exercises the record conversion sub system. Refer to record_conv.h header.

OPTIONS

`-v` *`level`* Sets the LOG level to *`level`*. Level is a sequence of tokens separated by comma. Each token is a integer or a named LOG item - one of `fatal`, `debug`, `warn`, `log`, `malloc`, `all`, `none`.

EXAMPLES

The following backend configuration converts MARC records (ISO2709) to Dublin-Core XML.

```
    <backend name="F" syntax="usmarc">
      <marc inputcharset="marc-8" inputformat="marc" outputformat="marcxml ↩
         "/>
      <xslt stylesheet="../etc/MARC21slim2DC.xsl"/>
    </backend>
```

We can convert one of the sample records from YAZ' test directory with:

```
$ ../util/yaz-record-conv record-conv-conf.xml marc6.marc
<?xml version="1.0"?>
<dc:dc xmlns:dc="http://purl.org/dc/elements/1.1/">
  <dc:title>How to program a computer</dc:title>
  <dc:creator>
    Jack Collins
  </dc:creator>
  <dc:type>text</dc:type>
  <dc:publisher>Penguin</dc:publisher>
  <dc:language>eng</dc:language>
</dc:dc>
```

FILES

`record_conv.h`

SEE ALSO

yaz(7)

Appendix A

List of Object Identifiers

These is a list of object identifiers that are built into YAZ.

Name	Class	Constant / OID
BER	TRANSYN	yaz_oid_transyn_ber
		2.1.1
ISO2709	TRANSYN	yaz_oid_transyn_iso2709
		1.0.2709.1.1
ISOILL-1	GENERAL	yaz_oid_general_isoill_1
		1.0.10161.2.1
Z-APDU	ABSYN	yaz_oid_absyn_z_apdu
		2.1
Z-BASIC	APPCTX	yaz_oid_appctx_z_basic
		1.1
Bib-1	ATTSET	yaz_oid_attset_bib_1
		Z3950_PREFIX.3.1
Exp-1	ATTSET	yaz_oid_attset_exp_1
		Z3950_PREFIX.3.2
Ext-1	ATTSET	yaz_oid_attset_ext_1
		Z3950_PREFIX.3.3
CCL-1	ATTSET	yaz_oid_attset_ccl_1
		Z3950_PREFIX.3.4
GILS	ATTSET	yaz_oid_attset_gils
		Z3950_PREFIX.3.5
GILS-attset	ATTSET	yaz_oid_attset_gils_attset
		Z3950_PREFIX.3.5
STAS-attset	ATTSET	yaz_oid_attset_stas_attset
		Z3950_PREFIX.3.6
Collections-attset	ATTSET	yaz_oid_attset_collections_attset
		Z3950_PREFIX.3.7
CIMI-attset	ATTSET	yaz_oid_attset_cimi_attset
		Z3950_PREFIX.3.8
Geo-attset	ATTSET	yaz_oid_attset_geo_attset

Name	Class	Constant / OID
		Z3950_PREFIX.3.9
ZBIG	ATTSET	yaz_oid_attset_zbig
		Z3950_PREFIX.3.10
Util	ATTSET	yaz_oid_attset_util
		Z3950_PREFIX.3.11
XD-1	ATTSET	yaz_oid_attset_xd_1
		Z3950_PREFIX.3.12
Zthes	ATTSET	yaz_oid_attset_zthes
		Z3950_PREFIX.3.13
Fin-1	ATTSET	yaz_oid_attset_fin_1
		Z3950_PREFIX.3.14
Dan-1	ATTSET	yaz_oid_attset_dan_1
		Z3950_PREFIX.3.15
Holdings	ATTSET	yaz_oid_attset_holdings
		Z3950_PREFIX.3.16
MARC	ATTSET	yaz_oid_attset_marc
		Z3950_PREFIX.3.17
Bib-2	ATTSET	yaz_oid_attset_bib_2
		Z3950_PREFIX.3.18
ZeeRex	ATTSET	yaz_oid_attset_zeerex
		Z3950_PREFIX.3.19
Thesaurus-attset	ATTSET	yaz_oid_attset_thesaurus_attset
		Z3950_PREFIX.3.1000.81.1
IDXPATH	ATTSET	yaz_oid_attset_idxpath
		Z3950_PREFIX.3.1000.81.2
EXTLITE	ATTSET	yaz_oid_attset_extlite
		Z3950_PREFIX.3.1000.81.3
Bib-1	DIAGSET	yaz_oid_diagset_bib_1
		Z3950_PREFIX.4.1
Diag-1	DIAGSET	yaz_oid_diagset_diag_1
		Z3950_PREFIX.4.2
Diag-ES	DIAGSET	yaz_oid_diagset_diag_es
		Z3950_PREFIX.4.3
Diag-General	DIAGSET	yaz_oid_diagset_diag_general
		Z3950_PREFIX.4.3
Unimarc	RECSYN	yaz_oid_recsyn_unimarc
		Z3950_PREFIX.5.1
Intermarc	RECSYN	yaz_oid_recsyn_intermarc
		Z3950_PREFIX.5.2
CCF	RECSYN	yaz_oid_recsyn_ccf
		Z3950_PREFIX.5.3
USmarc	RECSYN	yaz_oid_recsyn_usmarc
		Z3950_PREFIX.5.10
MARC21	RECSYN	yaz_oid_recsyn_marc21

Name	Class	Constant / OID
		Z3950_PREFIX.5.10
UKmarc	RECSYN	yaz_oid_recsyn_ukmarc
		Z3950_PREFIX.5.11
Normarc	RECSYN	yaz_oid_recsyn_normarc
		Z3950_PREFIX.5.12
Librismarc	RECSYN	yaz_oid_recsyn_librismarc
		Z3950_PREFIX.5.13
Danmarc	RECSYN	yaz_oid_recsyn_danmarc
		Z3950_PREFIX.5.14
Finmarc	RECSYN	yaz_oid_recsyn_finmarc
		Z3950_PREFIX.5.15
MAB	RECSYN	yaz_oid_recsyn_mab
		Z3950_PREFIX.5.16
Canmarc	RECSYN	yaz_oid_recsyn_canmarc
		Z3950_PREFIX.5.17
SBN	RECSYN	yaz_oid_recsyn_sbn
		Z3950_PREFIX.5.18
Picamarc	RECSYN	yaz_oid_recsyn_picamarc
		Z3950_PREFIX.5.19
Ausmarc	RECSYN	yaz_oid_recsyn_ausmarc
		Z3950_PREFIX.5.20
Ibermarc	RECSYN	yaz_oid_recsyn_ibermarc
		Z3950_PREFIX.5.21
Carmarc	RECSYN	yaz_oid_recsyn_carmarc
		Z3950_PREFIX.5.22
Malmarc	RECSYN	yaz_oid_recsyn_malmarc
		Z3950_PREFIX.5.23
JPmarc	RECSYN	yaz_oid_recsyn_jpmarc
		Z3950_PREFIX.5.24
SWEmarc	RECSYN	yaz_oid_recsyn_swemarc
		Z3950_PREFIX.5.25
SIGLEmarc	RECSYN	yaz_oid_recsyn_siglemarc
		Z3950_PREFIX.5.26
ISDSmarc	RECSYN	yaz_oid_recsyn_isdsmarc
		Z3950_PREFIX.5.27
RUSmarc	RECSYN	yaz_oid_recsyn_rusmarc
		Z3950_PREFIX.5.28
Hunmarc	RECSYN	yaz_oid_recsyn_hunmarc
		Z3950_PREFIX.5.29
NACSIS-CATP	RECSYN	yaz_oid_recsyn_nacsis_catp
		Z3950_PREFIX.5.30
FINMARC2000	RECSYN	yaz_oid_recsyn_finmarc2000
		Z3950_PREFIX.5.31
MARC21-fin	RECSYN	yaz_oid_recsyn_marc21_fin
		Z3950_PREFIX.5.32
Explain	RECSYN	yaz_oid_recsyn_explain

Name	Class	Constant / OID
		Z3950_PREFIX.5.100
SUTRS	RECSYN	yaz_oid_recsyn_sutrs
		Z3950_PREFIX.5.101
OPAC	RECSYN	yaz_oid_recsyn_opac
		Z3950_PREFIX.5.102
Summary	RECSYN	yaz_oid_recsyn_summary
		Z3950_PREFIX.5.103
GRS-0	RECSYN	yaz_oid_recsyn_grs_0
		Z3950_PREFIX.5.104
GRS-1	RECSYN	yaz_oid_recsyn_grs_1
		Z3950_PREFIX.5.105
Extended	RECSYN	yaz_oid_recsyn_extended
		Z3950_PREFIX.5.106
Fragment	RECSYN	yaz_oid_recsyn_fragment
		Z3950_PREFIX.5.107
pdf	RECSYN	yaz_oid_recsyn_pdf
		Z3950_PREFIX.5.109.1
postscript	RECSYN	yaz_oid_recsyn_postscript
		Z3950_PREFIX.5.109.2
html	RECSYN	yaz_oid_recsyn_html
		Z3950_PREFIX.5.109.3
tiff	RECSYN	yaz_oid_recsyn_tiff
		Z3950_PREFIX.5.109.4
gif	RECSYN	yaz_oid_recsyn_gif
		Z3950_PREFIX.5.109.5
jpeg	RECSYN	yaz_oid_recsyn_jpeg
		Z3950_PREFIX.5.109.6
png	RECSYN	yaz_oid_recsyn_png
		Z3950_PREFIX.5.109.7
mpeg	RECSYN	yaz_oid_recsyn_mpeg
		Z3950_PREFIX.5.109.8
sgml	RECSYN	yaz_oid_recsyn_sgml
		Z3950_PREFIX.5.109.9
tiff-b	RECSYN	yaz_oid_recsyn_tiff_b
		Z3950_PREFIX.5.110.1
wav	RECSYN	yaz_oid_recsyn_wav
		Z3950_PREFIX.5.110.2
SQL-RS	RECSYN	yaz_oid_recsyn_sql_rs
		Z3950_PREFIX.5.111
SOIF	RECSYN	yaz_oid_recsyn_soif
		Z3950_PREFIX.5.1000.81.2
JSON	RECSYN	yaz_oid_recsyn_json
		Z3950_PREFIX.5.1000.81.3
XML	RECSYN	yaz_oid_recsyn_xml
		Z3950_PREFIX.5.109.10
text-XML	RECSYN	yaz_oid_recsyn_text_xml

Name	Class	Constant / OID
		Z3950_PREFIX.5.109.10
application-XML	RECSYN	yaz_oid_recsyn_application_xml
		Z3950_PREFIX.5.109.11
Resource-1	RESFORM	yaz_oid_resform_resource_1
		Z3950_PREFIX.7.1
Resource-2	RESFORM	yaz_oid_resform_resource_2
		Z3950_PREFIX.7.2
UNIverse-Resource-Report	RESFORM	yaz_oid_resform_universe_resource_report
		Z3950_PREFIX.7.1000.81.1
Prompt-1	ACCFORM	yaz_oid_accform_prompt_1
		Z3950_PREFIX.8.1
Des-1	ACCFORM	yaz_oid_accform_des_1
		Z3950_PREFIX.8.2
Krb-1	ACCFORM	yaz_oid_accform_krb_1
		Z3950_PREFIX.8.3
Persistent set	EXTSERV	yaz_oid_extserv_persistent_set
		Z3950_PREFIX.9.1
Persistent query	EXTSERV	yaz_oid_extserv_persistent_query
		Z3950_PREFIX.9.2
Periodic query	EXTSERV	yaz_oid_extserv_periodic_query
		Z3950_PREFIX.9.3
Item order	EXTSERV	yaz_oid_extserv_item_order
		Z3950_PREFIX.9.4
Database Update (first version)	EXTSERV	yaz_oid_extserv_database_update_first_version
		Z3950_PREFIX.9.5
Database Update (second version)	EXTSERV	yaz_oid_extserv_database_update_second_version
		Z3950_PREFIX.9.5.1
Database Update	EXTSERV	yaz_oid_extserv_database_update
		Z3950_PREFIX.9.5.1.1
exp. spec.	EXTSERV	yaz_oid_extserv_exp__spec_
		Z3950_PREFIX.9.6
exp. inv.	EXTSERV	yaz_oid_extserv_exp__inv_
		Z3950_PREFIX.9.7
Admin	EXTSERV	yaz_oid_extserv_admin
		Z3950_PREFIX.9.1000.81.1
searchResult-1	USERINFO	yaz_oid_userinfo_searchresult_1
		Z3950_PREFIX.10.1

Name	Class	Constant / OID
CharSetandLanguageNegotiation	USERINFO	yaz_oid_userinfo_charsetandlanguagenegotiation
		Z3950_PREFIX.10.2
UserInfo-1	USERINFO	yaz_oid_userinfo_userinfo_1
		Z3950_PREFIX.10.3
MultipleSearchTerms-1	USERINFO	yaz_oid_userinfo_multiplesearchterms_1
		Z3950_PREFIX.10.4
MultipleSearchTerms-2	USERINFO	yaz_oid_userinfo_multiplesearchterms_2
		Z3950_PREFIX.10.5
DateTime	USERINFO	yaz_oid_userinfo_datetime
		Z3950_PREFIX.10.6
Proxy	USERINFO	yaz_oid_userinfo_proxy
		Z3950_PREFIX.10.1000.81.1
Cookie	USERINFO	yaz_oid_userinfo_cookie
		Z3950_PREFIX.10.1000.81.2
Client-IP	USERINFO	yaz_oid_userinfo_client_ip
		Z3950_PREFIX.10.1000.81.3
Scan-Set	USERINFO	yaz_oid_userinfo_scan_set
		Z3950_PREFIX.10.1000.81.4
Facet-1	USERINFO	yaz_oid_userinfo_facet_1
		Z3950_PREFIX.10.1000.81.5
Espec-1	ELEMSPEC	yaz_oid_elemspec_espec_1
		Z3950_PREFIX.11.1
Variant-1	VARSET	yaz_oid_varset_variant_1
		Z3950_PREFIX.12.1
WAIS-schema	SCHEMA	yaz_oid_schema_wais_schema
		Z3950_PREFIX.13.1
GILS-schema	SCHEMA	yaz_oid_schema_gils_schema
		Z3950_PREFIX.13.2
Collections-schema	SCHEMA	yaz_oid_schema_collections_schema
		Z3950_PREFIX.13.3
Geo-schema	SCHEMA	yaz_oid_schema_geo_schema
		Z3950_PREFIX.13.4
CIMI-schema	SCHEMA	yaz_oid_schema_cimi_schema
		Z3950_PREFIX.13.5
Update ES	SCHEMA	yaz_oid_schema_update_es
		Z3950_PREFIX.13.6
Holdings	SCHEMA	yaz_oid_schema_holdings
		Z3950_PREFIX.13.7
Zthes	SCHEMA	yaz_oid_schema_zthes
		Z3950_PREFIX.13.8
thesaurus-schema	SCHEMA	yaz_oid_schema_thesaurus_schema

Name	Class	Constant / OID
		Z3950_PREFIX.13.1000.81.1
Explain-schema	SCHEMA	yaz_oid_schema_explain_schema
		Z3950_PREFIX.13.1000.81.2
TagsetM	TAGSET	yaz_oid_tagset_tagsetm
		Z3950_PREFIX.14.1
TagsetG	TAGSET	yaz_oid_tagset_tagsetg
		Z3950_PREFIX.14.2
STAS-tagset	TAGSET	yaz_oid_tagset_stas_tagset
		Z3950_PREFIX.14.3
GILS-tagset	TAGSET	yaz_oid_tagset_gils_tagset
		Z3950_PREFIX.14.4
Collections-tagset	TAGSET	yaz_oid_tagset_collections_tagset
		Z3950_PREFIX.14.5
CIMI-tagset	TAGSET	yaz_oid_tagset_cimi_tagset
		Z3950_PREFIX.14.6
thesaurus-tagset	TAGSET	yaz_oid_tagset_thesaurus_tagset
		Z3950_PREFIX.14.1000.81.1
Explain-tagset	TAGSET	yaz_oid_tagset_explain_tagset
		Z3950_PREFIX.14.1000.81.2
Zthes-tagset	TAGSET	yaz_oid_tagset_zthes_tagset
		Z3950_PREFIX.14.8
Charset-3	NEGOT	yaz_oid_negot_charset_3
		Z3950_PREFIX.15.3
Charset-4	NEGOT	yaz_oid_negot_charset_4
		Z3950_PREFIX.15.4
Charset-ID	NEGOT	yaz_oid_negot_charset_id
		Z3950_PREFIX.15.1000.81.1
CQL	USERINFO	yaz_oid_userinfo_cql
		Z3950_PREFIX.16.2
UCS-2	GENERAL	yaz_oid_general_ucs_2
		1.0.10646.1.0.2
UCS-4	GENERAL	yaz_oid_general_ucs_4
		1.0.10646.1.0.4
UTF-16	GENERAL	yaz_oid_general_utf_16
		1.0.10646.1.0.5
UTF-8	GENERAL	yaz_oid_general_utf_8
		1.0.10646.1.0.8
OCLC-userInfo	USERINFO	yaz_oid_userinfo_oclc_userinfo
		Z3950_PREFIX.10.1000.17.1
XML-ES	EXTSERV	yaz_oid_extserv_xml_es

Name	Class	Constant / OID
		Z3950_PREFIX.9.1000.105.4

Appendix B

Bib-1 diagnostics

List of Bib-1 diagnostics that are known to YAZ.

Code	Text
1	Permanent system error
2	Temporary system error
3	Unsupported search
4	Terms only exclusion (stop) words
5	Too many argument words
6	Too many boolean operators
7	Too many truncated words
8	Too many incomplete subfields
9	Truncated words too short
10	Invalid format for record number (search term)
11	Too many characters in search statement
12	Too many records retrieved
13	Present request out of range
14	System error in presenting records
15	Record no authorized to be sent intersystem
16	Record exceeds Preferred-message-size
17	Record exceeds Maximum-record-size
18	Result set not supported as a search term
19	Only single result set as search term supported
20	Only ANDing of a single result set as search term supported
21	Result set exists and replace indicator off
22	Result set naming not supported
23	Combination of specified databases not supported
24	Element set names not supported
25	Specified element set name not valid for specified database
26	Only a single element set name supported
27	Result set no longer exists - unilaterally deleted by target
28	Result set is in use
29	One of the specified databases is locked
30	Specified result set does not exist

Code	Text
31	Resources exhausted - no results available
32	Resources exhausted - unpredictable partial results available
33	Resources exhausted - valid subset of results available
100	Unspecified error
101	Access-control failure
102	Security challenge required but could not be issued - request terminated
103	Security challenge required but could not be issued - record not included
104	Security challenge failed - record not included
105	Terminated by negative continue response
106	No abstract syntaxes agreed to for this record
107	Query type not supported
108	Malformed query
109	Database unavailable
110	Operator unsupported
111	Too many databases specified
112	Too many result sets created
113	Unsupported attribute type
114	Unsupported Use attribute
115	Unsupported value for Use attribute
116	Use attribute required but not supplied
117	Unsupported Relation attribute
118	Unsupported Structure attribute
119	Unsupported Position attribute
120	Unsupported Truncation attribute
121	Unsupported Attribute Set
122	Unsupported Completeness attribute
123	Unsupported attribute combination
124	Unsupported coded value for term
125	Malformed search term
126	Illegal term value for attribute
127	Unparsable format for un-normalized value
128	Illegal result set name
129	Proximity search of sets not supported
130	Illegal result set in proximity search
131	Unsupported proximity relation
132	Unsupported proximity unit code
201	Proximity not supported with this attribute combination
202	Unsupported distance for proximity
203	Ordered flag not supported for proximity
205	Only zero step size supported for Scan
206	Specified step size not supported for Scan
207	Cannot sort according to sequence
208	No result set name supplied on Sort
209	Generic sort not supported (database-specific sort only supported)
210	Database specific sort not supported
211	Too many sort keys

Code	Text
212	Duplicate sort keys
213	Unsupported missing data action
214	Illegal sort relation
215	Illegal case value
216	Illegal missing data action
217	Segmentation: Cannot guarantee records will fit in specified segments
218	ES: Package name already in use
219	ES: no such package, on modify/delete
220	ES: quota exceeded
221	ES: extended service type not supported
222	ES: permission denied on ES - id not authorized
223	ES: permission denied on ES - cannot modify or delete
224	ES: immediate execution failed
225	ES: immediate execution not supported for this service
226	ES: immediate execution not supported for these parameters
227	No data available in requested record syntax
228	Scan: malformed scan
229	Term type not supported
230	Sort: too many input results
231	Sort: incompatible record formats
232	Scan: term list not supported
233	Scan: unsupported value of position-in-response
234	Too many index terms processed
235	Database does not exist
236	Access to specified database denied
237	Sort: illegal sort
238	Record not available in requested syntax
239	Record syntax not supported
240	Scan: Resources exhausted looking for satisfying terms
241	Scan: Beginning or end of term list
242	Segmentation: max-segment-size too small to segment record
243	Present: additional-ranges parameter not supported
244	Present: comp-spec parameter not supported
245	Type-1 query: restriction ('resultAttr') operand not supported
246	Type-1 query: 'complex' attributeValue not supported
247	Type-1 query: 'attributeSet' as part of AttributeElement not supported
1001	Malformed APDU
1002	ES: EXTERNAL form of Item Order request not supported
1003	ES: Result set item form of Item Order request not supported
1004	ES: Extended services not supported unless access control is in effect
1005	Response records in Search response not supported
1006	Response records in Search response not possible for specified database (or database combination)
1007	No Explain server. Addinfo: pointers to servers that have a surrogate Explain database for this server
1008	ES: missing mandatory parameter for specified function. Addinfo: parameter

Code	Text
1009	ES: Item Order, unsupported OID in itemRequest. Addinfo: OID
1010	Init/AC: Bad Userid
1011	Init/AC: Bad Userid and/or Password
1012	Init/AC: No searches remaining (pre-purchased searches exhausted)
1013	Init/AC: Incorrect interface type (specified id valid only when used with a particular access method or client)
1014	Init/AC: Authentication System error
1015	Init/AC: Maximum number of simultaneous sessions for Userid
1016	Init/AC: Blocked network address
1017	Init/AC: No databases available for specified userId
1018	Init/AC: System temporarily out of resources
1019	Init/AC: System not available due to maintenance
1020	Init/AC: System temporarily unavailable (Addinfo: when it's expected back up)
1021	Init/AC: Account has expired
1022	Init/AC: Password has expired so a new one must be supplied
1023	Init/AC: Password has been changed by an administrator so a new one must be supplied
1024	Unsupported Attribute
1025	Service not supported for this database
1026	Record cannot be opened because it is locked
1027	SQL error
1028	Record deleted
1029	Scan: too many terms requested. Addinfo: max terms supported
1040	ES: Invalid function
1041	ES: Error in retention time
1042	ES: Permissions data not understood
1043	ES: Invalid OID for task specific parameters
1044	ES: Invalid action
1045	ES: Unknown schema
1046	ES: Too many records in package
1047	ES: Invalid wait action
1048	ES: Cannot create task package -- exceeds maximum permissable size
1049	ES: Cannot return task package -- exceeds maximum permissable size
1050	ES: Extended services request too large
1051	Scan: Attribute set id required -- not supplied
1052	ES: Cannot process task package record -- exceeds maximum permissible record size for ES
1053	ES: Cannot return task package record -- exceeds maximum permissible record size for ES response
1054	Init: Required negotiation record not included
1055	Init: negotiation option required
1056	Attribute not supported for database
1057	ES: Unsupported value of task package parameter
1058	Duplicate Detection: Cannot dedup on requested record portion
1059	Duplicate Detection: Requested detection criterion not supported
1060	Duplicate Detection: Requested level of match not supported

Code	Text
1061	Duplicate Detection: Requested regular expression not supported
1062	Duplicate Detection: Cannot do clustering
1063	Duplicate Detection: Retention criterion not supported
1064	Duplicate Detection: Requested number (or percentage) of entries
1065	Duplicate Detection: Requested sort criterion not supported
1066	CompSpec: Unknown schema, or schema not supported.
1067	Encapsulation: Encapsulated sequence of PDUs not supported
1068	Encapsulation: Base operation (and encapsulated PDUs) not executed based on pre-screening analysis
1069	No syntaxes available for this request
1070	user not authorized to receive record(s) in requested syntax
1071	preferredRecordSyntax not supplied
1072	Query term includes characters that do not translate into the target character set
1073	Database records do not contain data associated with access point
1074	Proxy failure

Appendix C

SRU diagnostics

List of SRU diagnostics that are known to YAZ.

Code	Text
1	Permanent system error
2	System temporarily unavailable
3	Authentication error
4	Unsupported operation
5	Unsupported version
6	Unsupported parameter value
7	Mandatory parameter not supplied
8	Unsupported parameter
10	Query syntax error
11	Unsupported query type
12	Too many characters in query
13	Invalid or unsupported use of parentheses
14	Invalid or unsupported use of quotes
15	Unsupported context set
16	Unsupported index
17	Unsupported combination of index and context set
18	Unsupported combination of indexes
19	Unsupported relation
20	Unsupported relation modifier
21	Unsupported combination of relation modifers
22	Unsupported combination of relation and index
23	Too many characters in term
24	Unsupported combination of relation and term
25	Special characters not quoted in term
26	Non special character escaped in term
27	Empty term unsupported
28	Masking character not supported
29	Masked words too short
30	Too many masking characters in term
31	Anchoring character not supported

Code	Text
32	Anchoring character in unsupported position
33	Combination of proximity/adjacency and masking characters not supported
34	Combination of proximity/adjacency and anchoring characters not supported
35	Term contains only stopwords
36	Term in invalid format for index or relation
37	Unsupported boolean operator
38	Too many boolean operators in query
39	Proximity not supported
40	Unsupported proximity relation
41	Unsupported proximity distance
42	Unsupported proximity unit
43	Unsupported proximity ordering
44	Unsupported combination of proximity modifiers
45	Prefix assigned to multiple identifiers
46	Unsupported boolean modifier
47	Cannot process query; reason unknown
48	Query feature unsupported
49	Masking character in unsupported position
50	Result sets not supported
51	Result set does not exist
52	Result set temporarily unavailable
53	Result sets only supported for retrieval
54	Retrieval may only occur from an existing result set
55	Combination of result sets with search terms not supported
56	Only combination of single result set with search terms supported
57	Result set created but no records available
58	Result set created with unpredictable partial results available
59	Result set created with valid partial results available
60	Result set not created: too many matching records
61	First record position out of range
62	Negative number of records requested
63	System error in retrieving records
64	Record temporarily unavailable
65	Record does not exist
66	Unknown schema for retrieval
67	Record not available in this schema
68	Not authorised to send record
69	Not authorised to send record in this schema
70	Record too large to send
71	Unsupported record packing
72	XPath retrieval unsupported
73	XPath expression contains unsupported feature
74	Unable to evaluate XPath expression
80	Sort not supported
81	Unsupported sort type
82	Unsupported sort sequence

Code	Text
83	Too many records to sort
84	Too many sort keys to sort
85	Duplicate sort keys
86	Cannot sort: incompatible record formats
87	Unsupported schema for sort
88	Unsupported path for sort
89	Path unsupported for schema
90	Unsupported direction value
91	Unsupported case value
92	Unsupported missing value action
93	Sort ended due to missing value
100	Explain not supported
101	Explain request type not supported (SOAP vs GET)
102	Explain record temporarily unavailable
110	Stylesheets not supported
111	Unsupported stylesheet
120	Response position out of range
121	Too many terms requested
235	Database does not exist
236	Access to specified database denied
1015	Init/AC: Maximum number of simultaneous sessions for Userid
1074	Proxy failure

Appendix D

License

Index Data Copyright

Appendix E

About Index Data

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
Amagerfælledvej 56
2300 Copenhagen S
Denmark
Email info@indexdata.dk
```


The Hacker's Jargon File has the following to say about the use of the prefix "YA" in the name of a software product.

[**?**, **?**]

Appendix F

Credits

This appendix lists individuals that have contributed in the development of YAZ. Some have contributed with code, while others have provided bug fixes or suggestions. If we're missing somebody, of if you, for whatever reason, don't like to be listed here, let us know.

- Gary Anderson
- Dimitrios Andreadis
- Morten Bøgeskov
- Rocco Carbone
- Matthew Carey
- Hans van Dalen
- Irina Dijour
- Larry E. Dixson
- Hans van den Dool
- Mads Bondo Dydensborg
- Franck Falcoz
- Kevin Gamiel
- Morten Garkier Hendriksen
- Morten Holmqvist
- Ian Ibbotson
- Shigeru Ishida
- Heiko Jansen

- David Johnson
- Oleg Kolobov
- Giannis Kosmas
- Kang-Jin Lee
- Pieter Van Lierop
- Stefan Lohrum
- Ronald van der Meer
- Thomas W. Place
- Peter Popovics
- Jacob Chr. Poulsen
- Ko van der Sloot
- Mike Taylor
- Rustam T. Usmanov
- Charles Woodfield
- Tom André Øverland