YAZ User's Guide and Reference

Index Data, *info@indexdata.dk* $Revision: 1.5 $

This document is the programmer's guide and reference to the YAZ package. YAZ is a compact toolkit that provides access to the Z39.50/SR protocol, 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.

Contents

1 Introduction

The **YAZ** toolkit offers several different levels of access to the Z39.50 and SR protocols. The level that you need to use depends on your requirements, and the role (server or client) that you want to implement.

The basic level, which is independent of the role, consists of three primary interfaces:

- **ASN**, which provides a C representation of the Z39.50/SR protocol packages (PDUs).
- **ODR**, which encodes and decodes the packages according to the BER specification.
- **COMSTACK**, which exchanges the encoded packages with a peer process over a network.

The ASN module represents the ASN.1 definition of the SR/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 **ASN** 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 `proto.h` (for the types from the protocol), `odr.h` (for the primitive ASN.1 types, or `odr_use.h` (for the ASN.1 *useful* types). The **ASN** 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 **ASN** module, and you are not adding new protocol of your own, the only parts of ODR that you need to worry about are documented in section 5.2 (Using ODR).*

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 (OSI or TCP/IP), and which protocol you want to use (SR or Z39.50). For the remainer 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**, **ASN**, and **COMSTACK** the service level API. It's the API that most closely models the Z39.50/SR 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**/**ASN**, you can use SNACC or BERUtils instead, and still have the benefits of the transparent transport approach of the **COMSTACK** module. Secondly, we realise 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**.

2 Compilation and Installation

The latest version of the software will generally be found at

`http://ftp.indexdata.dk/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. So far, the software has been ported to the following platforms with little or no difficulties.

- Unix systems
  - HP/UX
  - SunOS/Solaris
  - DEC Unix
  - Linux
  - IBM AIX
  - Data General DG/UX (with some CFLAGS tinkering)
  - SGI/IRIX
  - DDE Supermax
- Non-unix systems
  - Apple Macintosh (using the Codewarrior programming environment and the GUSI socket libraries)
  - MS Windows 95/NT (Win32)
  - 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 sending mail to `yaz-request@indexdata.dk`. General questions and problems can be directed at `yaz-help@indexdata.dk`, or the address given at the top of this document.

2.1 UNIX

Note that if your system doesn't have a native ANSI C compiler, you may have to acquire one separately. We recommend gcc.

For UNIX we use GNU configure to create Makefiles for YAZ. Generally it should be sufficient to run configure without options:

```
./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` **path**

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

`--enable-comp`

YAZ will be built using the ASN.1 compiler for YAZ (default). If you wish to use the old decoders (in sub directory asn) use `-disable-comp` instead.

`--enable-threads`

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

When configured, build the software by typing:

```
make
```

Developers that do modifications to **YAZ** may wish to run `make depend` as well to generate object-header file dependencies.

The following files are generated by make

`lib/libyaz.a`

The **YAZ** programmers' library.

`ztest/yaz-ztest`

A test Z39.50 server.

`client/yaz-client`

A command mode Z39.50 client.

`yaz-config`

A Bourne-shell script that holds build settings for **YAZ**.

If you wish to install **YAZ** in system directories such as /usr/local/bin, /usr/local/lib) 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 going to install in anything but /usr/local/.

2.2 WIN32

YAZ is shipped with "makefiles" for the NMAKE tool that comes with Visual C++.

Start an MS-DOS 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 summarises the most important settings in that file:

`NEW_Z3950`

If 1, the auto-generated decoder/encoders for Z39.50 as written by the ASN.1 compiler will be used. If 0, the old decoders for Z39.50 will be used. Note, when 1, the setting TCL should point to the Tcl shell on your system.

`DEBUG`

If set to 1, the software is compiled with debugging libraries. If set to 0, the software is compiled with release (non-debugging) libraries.

When satisfied with the settings in the makefile type

```
nmake
```

The following is generated upon successful compilation:

`bin/yaz.dll`

A multithreaded DLL with everything except the frontend server library.

`lib/yaz.lib`

An import library for `yaz.dll`.

`lib/server.lib`

The frontend server library.

`bin/yaz-ztest.exe`

A test Z39.50 server.

`bin/yaz-client.exe`

A command mode Z39.50 client.

3 The ASN Module

3.1 Introduction

The **ASN** module provides you with a set of C struct definitions for the various PDUs of the 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 `int`. For ASN.1 constructs that have no direct representation in C, such as general octet strings and bit strings, the **ODR** module (see section 5 (ODR)) provides auxiliary definitions.

3.2 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 InitReqest, 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 5.2 (Using ODR) 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. `odr_malloc()` 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 **ASN** 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);
```

eg.:

```
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 toplevel 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 **ASN** module, and you access it through the `proto.h` header file.

3.3 Object Identifiers

When you refer to object identifiers in your application, you need to be aware that SR and Z39.50 use two different set of OIDs to refer to the same objects. To handle this easily, **YAZ** provides a utility module to **ASN** which provides an internal representation of the OIDs used in both protocols. Each oid is described by a structure:

```
typedef struct oident
{
    enum oid_proto proto;
    enum oid_class class;
    enum oid_value value;
    int oidsuffix[OID_SIZE];
    char *desc;
} oident;
```

The `proto` field can be set to either `PROTO_SR` or `PROTO_Z3950`. The `class` might be, say, `CLASS_RECSYN`, and the `value` might be `VAL_USMARC` for the USMARC record format. Functions

```
int *oid_ent_to_oid(struct oident *ent, int *dst);
struct oident *oid_getentbyoid(int *o);
```

are provided to map between object identifiers and database entries. If you store a member of the `oid_proto` type in your association state information, it's a simple matter, at runtime, to generate the correct OID when you need it. For decoding, you can simply ignore the proto field, or if you're strict, you can verify that your peer is using the OID family from the correct protocol. The `desc` field is a short, human-readable name for the PDU, useful mainly for diagnostic output.

NOTE: The old function oid_getoidbyent still exists but is not thread safe. Use oid_ent_to_oid instead and pass an array of size OID_SIZE.

NOTE: Plans are underway to merge the two protocols into a single definition, with one set of object identifiers. When this happens, the oid module will no longer be required to support protocol independence, but it should still be useful as a simple OID database.

3.4 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 **ASN** 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 **ASN** 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 (eg. 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(oid_value ref);
```

Can be used to retrieve information about the known, external data types. The function return 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
{
    oid_value dref;    /* the direct-reference OID value. */
    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 5.2 (Using ODR)).

If you want to *send* EXTERNALs containing ASN.1-structured values in the occtet-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` arrary 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.

3.5 PDU Contents Table

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

```
Z_InitRequest
-------------
Field                    Type                 Default value

referenceId              Z_ReferenceId        NULL
protocolVersion          Odr_bitmask          Empty bitmask
options                  Odr_bitmask          Empty bitmask
preferredMessageSize     int                  30*1024
maximumRecordSize        int                  30*1024
idAuthentication         Z_IdAuthentication   NULL
implementationId         char*                "YAZ (id=81)"
implementationName       char*                "Index Data/YAZ"
implementationVersion    char*                YAZ_VERSION
userInformationField     Z_UserInformation    NULL
otherInfo                Z_OtherInformation   NULL

Z_InitResponse
--------------
Field                    Type                 Default value

referenceId              Z_ReferenceId        NULL
protocolVersion          Odr_bitmask          Empty bitmask
options                  Odr_bitmask          Empty bitmask
preferredMessageSize     int                  30*1024
maximumRecordSize        int                  30*1024
result                   bool_t               TRUE
implementationId         char*                "YAZ (id=81)"
implementationName       char*                "Index Data/YAZ"
implementationVersion    char*                YAZ_VERSION
userInformationField     Z_UserInformat..     NULL
otherInfo                Z_OtherInformation   NULL

Z_SearchRequest
---------------
Field                    Type                 Default value

referenceId              Z_ReferenceId        NULL
smallSetUpperBound       int                  0
largeSetLowerBound       int                  1
mediumSetPresentNumber   int                  0
replaceIndicator         bool_t               TRUE
resultSetName            char*                "default"
num_databaseNames        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

Z_SearchResponse
----------------
Field                        Type                 Default value

referenceId                  Z_ReferenceId        NULL
resultCount                  int                  0
numberOfRecordsReturned      int                  0
nextResultSetPosition        int                  0
searchStatus                 bool_t               TRUE
resultSetStatus              int                  NULL
presentStatus                int                  NULL
records                      Z_Records            NULL
additionalSearchInfo         Z_OtherInformation   NULL
otherInfo                    Z_OtherInformation   NULL

Z_PresentRequest
----------------
Field                        Type                 Default value

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

Z_PresentResponse
-----------------
Field                        Type                 Default value

referenceId                  Z_ReferenceId        NULL
numberOfRecordsReturned      int                  0
nextResultSetPosition        int                  0
```

```
presentStatus               int                 Z_PRES_SUCCESS
records                     Z_Records           NULL
otherInfo                   Z_OtherInformation  NULL

Z_DeleteResultSetRequest
------------------------
Field                       Type                Default value

referenceId                 Z_ReferenceId       NULL
deleteFunction              int                 Z_DeleteRequest_list
num_ids                     int                 0
resultSetList               char**              NULL
otherInfo                   Z_OtherInformation  NULL

Z_DeleteResultSetResponse
-------------------------
Field                       Type                Default value

referenceId                 Z_ReferenceId       NULL
deleteOperationStatus       int                 Z_DeleteStatus_success
num_statuses                int                 0
deleteListStatuses          Z_ListStatus**      NULL
numberNotDeleted            int                 NULL
num_bulkStatuses            int                 0
bulkStatuses                Z_ListStatus        NULL
deleteMessage               char*               NULL
otherInfo                   Z_OtherInformation  NULL

Z_ScanRequest
-------------
Field                       Type                Default value

referenceId                 Z_ReferenceId       NULL
num_databaseNames           int                 0
databaseNames               char**              NULL
attributeSet                Odr_oid             NULL
termListAndStartPoint       Z_AttributesPlus... NULL
stepSize                    int                 NULL
numberOfTermsRequested      int                 20
preferredPositionInResponse int                 NULL
otherInfo                   Z_OtherInformation  NULL

Z_ScanResponse
--------------
Field                       Type                Default value

referenceId                 Z_ReferenceId       NULL
```

```
stepSize                    int                 NULL
scanStatus                  int                 Z_Scan_success
numberOfEntriesReturned     int                 0
positionOfTerm              int                 NULL
entries                     Z_ListEntris        NULL
attributeSet                Odr_oid             NULL
otherInfo                   Z_OtherInformation  NULL

Z_TriggerResourceControlRequest
-------------------------------
Field                       Type                Default value

referenceId                 Z_ReferenceId       NULL
requestedAction             int                 Z_TriggerResourceCtrl_resou..
prefResourceReportFormat    Odr_oid             NULL
resultSetWanted             bool_t              NULL
otherInfo                   Z_OtherInformation  NULL

Z_ResourceControlRequest
------------------------
Field                       Type                Default value

referenceId                 Z_ReferenceId       NULL
suspendedFlag               bool_t              NULL
resourceReport              Z_External          NULL
partialResultsAvailable     int                 NULL
responseRequired            bool_t              FALSE
triggeredRequestFlag        bool_t              NULL
otherInfo                   Z_OtherInformation  NULL

Z_ResourceControlResponse
-------------------------
Field                       Type                Default value

referenceId                 Z_ReferenceId       NULL
continueFlag                bool_t              TRUE
resultSetWanted             bool_t              NULL
otherInfo                   Z_OtherInformation  NULL

Z_AccessControlRequest
----------------------
Field                       Type                Default value

referenceId                 Z_ReferenceId       NULL
which                       enum                Z_AccessRequest_simpleForm;
u                           union               NULL
otherInfo                   Z_OtherInformation  NULL
```

```
Z_AccessControlResponse
-----------------------
Field                        Type                Default value

referenceId                  Z_ReferenceId       NULL
which                        enum                Z_AccessResponse_simpleForm
u                            union               NULL
diagnostic                   Z_DiagRec           NULL
otherInfo                    Z_OtherInformation  NULL

Z_Segment
---------
Field                        Type                Default value

referenceId                  Z_ReferenceId       NULL
numberOfRecordsReturned      int                 value=0
num_segmentRecords           int                 0
segmentRecords               Z_NamePlusRecord    NULL
otherInfo                    Z_OtherInformation  NULL

Z_Close
-------
Field                        Type                Default value

referenceId                  Z_ReferenceId       NULL
closeReason                  int                 Z_Close_finished
diagnosticInformation        char*               NULL
resourceReportFormat         Odr_oid             NULL
resourceFormat               Z_External          NULL
otherInfo                    Z_OtherInformation  NULL
```

4 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.

4.1 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 two separate, query-generating tools that may be of use to you.

4.1.1 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.

The PQF is defined by the pquery module in the YAZ library. The `pquery.h` file provides the declaration of the functions

```
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 5 (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`.

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 ::= [ AttSet ] QueryStruct.

AttSet ::= string.

QueryStruct ::= { Attribute } Simple | Complex.

Attribute ::= '@attr' AttributeType '=' AttributeValue.

AttributeType ::= integer.

AttributeValue ::= integer.

Complex ::= Operator QueryStruct QueryStruct.

Operator ::= '@and' | '@or' | '@not' | '@prox' Proximity.

Simple ::= ResultSet | Term.

ResultSet ::= '@set' string.
```

```
Term ::= string | '"' string '"'.

Proximity ::= Exclusion Distance Ordered Relation WhichCode UnitCode.

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

Distance ::= integer.

Ordered ::= '1' | '0'.

Relation ::= integer.

WhichCode ::= 'known' | 'private' | integer.

UnitCode ::= integer.
```

You will note that the syntax above is a fairly faithful representation of RPN, except for the `Attibute`, 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 following are all examples of valid queries in the PQF.

```
dylan

"bob dylan"

@or "dylan" "zimmerman"

@set Result-1

@or @and bob dylan @set Result-1

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

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

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

4.1.2 Common Command Language

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 (or 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.

The EUROPAGATE research project working under the Libraries programme of the European Commission's DG XIII has, amongst other useful tools, implemented a general-purpose CCL parser which produces an output structure that can be trivially converted to the internal RPN representation of YAZ (The `Z_RPNQuery` structure). Since the CCL utility - along with the rest of the software produced by EUROPAGATE - is made freely available on a liberal license, it is included as a supplement to YAZ.

CCL Syntax The CCL parser obeys the following grammar for the FIND argument. The syntax is annotated by in the 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
```

The following queries are all valid:

```
dylan

"bob dylan"

dylan or zimmerman

set=1
```

```
(dylan and bob) or set=1
```

Assuming that the qualifiers `ti`, `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 short-hand notations. You can customize the CCL parser to support a particular set of qualifiers to relect the current target profile. Traditionally, a qualifier would map to a particular use-attribute within the BIB-1 attribute set. However, you could also define qualifiers that would set, for example, the structure-attribute.

Consider a scenario where the target support ranked searches in the title-index. In this case, the user could specify

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

and the `ranked` would map to structure=free-form-text (4=105) and the `ti` would map to title (1=4).

A "profile" with a set predefined CCL qualifiers can be read from a file. The YAZ client reads its CCL qualifiers from a file named `default.bib`. Each line in the file has the form:

qualifier-name type=val type=val ...

where *qualifier-name* is the name of the qualifier to be used (eg. `ti`), *type* is a BIB-1 category type and *val* is the corresponding BIB-1 attribute value. The *type* can be either numeric or it may be either `u` (use), `r` (relation), `p` (position), `s` (structure), `t` (truncation) or `c` (completeness). The *qualifier-name* `term` has a special meaning. The types and values for this definition is used when *no* qualifier is present.

Consider the following definition:

```
ti      u=4 s=1
au      u=1 s=1
term    s=105
```

Two qualifiers are defined, `ti` and `au`. They both set the structure-attribute to phrase (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).

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 eccur, 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.

4.2 Object Identifiers

The basic YAZ representation of an OID is an array of integers, terminated with the value -1. The **ODR** module provides two utility-functions to create and copy this type of data elements:

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

Creates an OID based on a string-based representation using dots (.) to separate elements in the OID.

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

Creates a copy of the OID referenced by the *o* parameter. Both functions 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.

The **OID** module provides a higher-level representation of the family of object identifers which describe the Z39.50 protocol and its related objects. The definition of the module interface is given in the `oid.h` file.

The interface is mainly based on the `oident` structure. The definition of this structure looks like this:

```
typedef struct oident
{
    oid_proto proto;
    oid_class oclass;
    oid_value value;
    int oidsuffix[OID_SIZE];
    char *desc;
} oident;
```

The *proto* field takes one of the values

```
PROTO_Z3950
PROTO_SR
```

If you don't care about talking to SR-based implementations (few exist, and they may become fewer still if and when the ISO SR and ANSI Z39.50 documents are merged into a single standard), you can ignore this field on incoming packages, and always set it to PROTO_Z3950 for outgoing packages.

The *oclass* field takes one of the values

```
CLASS_APPCTX
CLASS_ABSYN
CLASS_ATTSET
CLASS_TRANSYN
CLASS_DIAGSET
CLASS_RECSYN
CLASS_RESFORM
CLASS_ACCFORM
CLASS_EXTSERV
CLASS_USERINFO
CLASS_ELEMSPEC
CLASS_VARSET
CLASS_SCHEMA
CLASS_TAGSET
CLASS_GENERAL
```

corresponding to the OID classes defined by the Z39.50 standard.

Finally, the *value* field takes one of the values

```
VAL_APDU
VAL_BER
VAL_BASIC_CTX
VAL_BIB1
VAL_EXP1
VAL_EXT1
VAL_CCL1
VAL_GILS
VAL_WAIS
VAL_STAS
VAL_DIAG1
VAL_ISO2709
VAL_UNIMARC
VAL_INTERMARC
VAL_CCF
VAL_USMARC
VAL_UKMARC
VAL_NORMARC
VAL_LIBRISMARC
VAL_DANMARC
VAL_FINMARC
VAL_MAB
```

```
VAL_CANMARC
VAL_SBN
VAL_PICAMARC
VAL_AUSMARC
VAL_IBERMARC
VAL_EXPLAIN
VAL_SUTRS
VAL_OPAC
VAL_SUMMARY
VAL_GRS0
VAL_GRS1
VAL_EXTENDED
VAL_RESOURCE1
VAL_RESOURCE2
VAL_PROMPT1
VAL_DES1
VAL_KRB1
VAL_PRESSET
VAL_PQUERY
VAL_PCQUERY
VAL_ITEMORDER
VAL_DBUPDATE
VAL_EXPORTSPEC
VAL_EXPORTINV
VAL_NONE
VAL_SETM
VAL_SETG
VAL_VAR1
VAL_ESPEC1
```

again, corresponding to the specific OIDs defined by the standard.

The *desc* field contains a brief, mnemonic name for the OID in question.

The function

```
struct oident *oid_getentbyoid(int *o);
```

takes as argument an OID, and returns a pointer to a static area containing an `oident` structure. You typically use this function when you receive a PDU containing an OID, and you wish to branch out depending on the specific OID value.

The function

```
int *oid_ent_to_oid(struct oident *ent, int *dst);
```

Takes as argument an `oident` structure - in which the *proto*, *oclass*, and *value* fields are assumed to be set correctly - and returns a pointer to a the buffer as given by *dst* containing the base representation of the corresponding OID. The function returns NULL and the array dst is unchanged if a mapping couldn't place. The array *dst* should be at least of size `OID_SIZE`.

The `oid_ent_to_oid()` function can be used whenever you need to prepare a PDU containing one or more OIDs. The separation of the *protocol* element from the remainer of the OID-description makes it simple to write applications that can communicate with either Z39.50 or OSI SR-based applications.

The function

```
oid_value oid_getvalbyname(const char *name);
```

takes as argument a mnemonic OID name, and returns the *value* field of the first entry in the database that contains the given name in its *desc* field.

Finally, the module provides the following utility functions, whose meaning should be obvious:

```
void oid_oidcpy(int *t, int *s);
void oid_oidcat(int *t, int *s);
int oid_oidcmp(int *o1, int *o2);
int oid_oidlen(int *o);
```

*NOTE: The **OID** module has been criticized - and perhaps rightly so - for needlessly abstracting the representation of OIDs. Other toolkits use a simple string-representation of OIDs with good results. In practice, we have found the interface comfortable and quick to work with, and it is a simple matter (for what it's worth) to create applications compatible with both ISO SR and Z39.50. Finally, the use of the `oident` database is by no means mandatory. You can easily create your own system for representing OIDs, as long as it is compatible with the low-level integer-array representation of the ODR module.*

4.3 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 5.2 (Using ODR)). 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, int size);
void nmem_reset(NMEM n);
int nmem_total(NMEM n);
void nmem_init(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.

Note that the nibble memory pool is shared amonst threads. Posix mutex'es and WIN32 Critical sections are introduced to keep the module thread safe. On WIN32 function `nmem_init()` initialises the Critical Section handle and should be called once before any other nmem function is used.

5 The ODR Module

5.1 Introduction

ODR is the BER-encoding/decoding subsystem of **YAZ**. Care as 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 5.2 (Using ODR)). 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 5.3 (Programming with ODR)). 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 5.3 (Programming with ODR) 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.

5.2 Using ODR

5.2.1 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.

5.2.2 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, int 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, int size);
```

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

```
int 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 freelist 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 maintaing 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.

5.2.3 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, signalling 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 realise 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 signalling 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 (eg, `odr_integer()` odr `z_APDU()`).

Examples of encoding/decoding functions:

```
int odr_integer(ODR o, 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.

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(int value)
{
    ODR encode, decode;
    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);
    printf("length of encoded data is %d\n", len);

    /* now let's decode the thing again */
    odr_setbuf(decode, bufferp, len);
    if (odr_integer(decode, &resvalp, 0, 0) == 0)
    {
        printf("decoding went bad\n");
        return;
    }
    printf("the value is %d\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.

5.2.4 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:

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.

The character string array

```
char *odr_errlist[]
```

can be indexed by the error code to obtain a human-readable representation of the problem.

5.2.5 Summary and Synopsis

```
#include <odr.h>

ODR odr_createmem(int direction);

void odr_destroy(ODR o);

void odr_reset(ODR o);

char *odr_getbuf(ODR o, int *len);

void odr_setbuf(ODR o, char *buf, int len);

void *odr_malloc(ODR o, int size);

ODR_MEM odr_extract_mem(ODR o);

void odr_release_mem(ODR_MEM r);

int odr_geterror(ODR o);

void odr_perror(char *message);

extern char *odr_errlist[];
```

5.3 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.

The 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 datatypes, 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. Although no ASN.1 compiler is supplied with **ODR** at this time, it shouldn't be too difficult to write one, or perhaps even to adapt an existing compiler to output **ODR** routines (not surprisingly, writing encoders/decoders using **ODR** turns out to be boring work).

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 memebers which don't exist in XDR.

5.3.1 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, int **p, int optional, const char *name);
```

(we don't allow values that can't be contained in a C 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, bool_t **p, int optional, const char *name);
```

REAL Not defined.

NULL

```
int odr_null(ODR o, bool_t **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;
    int size;
} 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, while the `size` field gives the size of the allocated array (not of interest to you, in most cases). 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 (eg. VisibleString):

```
int odr_cstring(ODR o, char **p, int optional, const char *name);
```

Which encoded or decodes between OCTETSTRING representations and null-terminates C strings.

Functions are provided for the derived string types, eg:

```
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, eg. 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 modelled 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 represenation is simply an array of integers, terminated by the value -1 (the `Odr_oid` type is synonymous with the `int` type). We suggest that you use the OID database module (see section 3.3 (Object Identifiers)) to handle object identifiers in your application.

5.3.2 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, 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()` and `odr_implicit()` 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`.

5.3.3 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. `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
{
    int *intval;
    bool_t *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()` or `odr_explicit()` 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 behaviour of the new type is compatible with the interface of the primitive types.

5.3.4 Tagging Constructed Types

NOTE: See section 5.3.2 (Tagging Primitive types) 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()` 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 were 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()` 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.

5.3.5 SEQUENCE OF

To handle sequences (arrays) of a apecific 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;
    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);
}
```

5.3.6 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.

class, tag

The class and tag of the type (-1 if no tagging is used).

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
    {
        int *untagged;
        int *tagged;
        bool_t *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 specifictions naturally requires 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.

5.4 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.

6 The COMSTACK Module

6.1 Introduction

The **COMSTACK** subsystem provides a transparent interface to different types of transport stacks for the exchange of BER-encoded data. At present, the RFC1729 method (BER over TCP/IP), and Peter Furniss' XTImOSI 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.

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.

6.2 Common Functions

6.2.1 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 values `tcpip_type` and `mosi_type` are recognized. The function returns a null-pointer if a system error occurs. The `blocking` parameter should be one if you wish the association to operate in blocking mode, zero otherwise. The `protocol` field should be one of `PROTO_SR` or `PROTO_Z3950`.

```
int cs_close(COMSTACK handle);
```

Closes the connection (as elegantly as the lower layers will permit), and releases the resouces 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?

6.2.2 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. -1 indicates an error condition (see below).

```
int cs_get(COMSTACK handle, char **buf, int *size);
```

Receives a PDU 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 fist 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). -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).

6.3 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) 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 occured.

6.4 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 finalises 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

```
char *cs_addrstr(COMSTACK);
```

on an established connection to retrieve the hostname of the remote host.

NOTE: You may need to use this function with some care if your name server service is slow or unreliable

6.5 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.

```
struct sockaddr_in *tcpip_strtoaddr(char *str);

struct netbuf *mosi_strtoaddr(char *str);
```

The format for TCP/IP addresses is straightforward:

```
<host> [ ':' <portnum> ]
```

The `hostname` can be either a domain name or an IP address. The port number, if omitted, defaults to 210.

For OSI, the format is

```
[ <t-selector> '/' ] <host> [ ':' <port> ]
```

The transport selector is given as an even number of hex digits.

You'll note that the address format for the OSI mode are just a subset of full presentation addresses. We use presentation addresses because xtimosi doesn't, in itself, allow access to the X.500 Directory service. We use a limited form, because we haven't yet come across an implementation that used more of the elements of a full p-address. It is a fairly simple matter to add the rest of the elements to the address format as needed, however: Xtimosi *does* support the full P-address structure.

In both transport modes, the special hostname "@" is mapped to any local address (the manifest constant INADDR_ANY). It is used to establish local listening endpoints in the server role.

When a connection has been established, you can use

```
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.

NOTE: We have left the issue of X.500 name-to-address mapping open, for the moment. It would be a simple matter to provide a table-based mapping, if desired. Alternately, we could use the X.500 client-function that is provided with the ISODE (although this would defeat some of the purpose of using ThinOSI in the first place. We have been told that it should be within the realm of the possible to implement a lightweight implementation of the necessary X.500 client capabilities on top of ThinOSI. This would be the ideal solution, we feel. On the other hand, it still remains to be seen just what role the Directory will play in a world populated by ThinOSI and other pragmatic solutions.

6.6 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.

When a function (including the data exchange functions) reports an error condition, use the function `cs_errno()` to determine the cause of the problem. The function

```
void cs_perror(COMSTACK handle char *message);
```

works like `perror`(2) and prints the `message` argument, along with a system message, to `stderr`. Use the character array

```
extern const char *cs_errlist[];
```

to get hold of the message, if you want to process it differently. The function

```
const char *cs_stackerr(COMSTACK handle);
```

Returns an error message from the lower layer, if one has been provided.

6.7 Enabling OSI Communication

6.7.1 Installing Xtimosi

Although you will have to download Peter Furniss' XTI/mOSI implementation for yourself, we've tried to make the integration as simple as possible.

The latest version of xtimosi will generally be under

```
ftp://pluto.ulcc.ac.uk/ulcc/thinosi/xtimosi/
```

When you have downloaded and unpacked the archive, it will (we assume) have created a directory called `xtimosi`. We suggest that you place this directory *in the same directory* where you unpacked the **YAZ** distribution. This way, you shouldn't have to fiddle with the makefiles of `YAZ` beyond uncommenting a few lines.

Go to `xtimosi/src`, and type `"make libmosi.a"`. This should generally create the library, ready to use.

CAVEAT

The currently available release of xtimosi has some inherent problems that make it disfunction on certain platforms - eg. the Digital OSF/1 workstations. It is supposedly primarily a compiler problem, and we hope to see a release that is generally portable. While we can't guarantee that it can be brought to work on your platform, we'll be happy to talk to you about problems that you might see, and relay information to the author of the software. There are some signs that the **gcc** *compiler is more likely to produce a fully functional library, but this hasn't been verified (we think that the problem is limited to the use of hexadecimal escape-codes used in strings, which are silently ignored by some compilers).*

A problem has been encountered in the communication with ISODE-based applications. If the ISODE presentation-user calls `PReadRequest()` with a timeout value different from `OK` or `NOTOK`, he will get an immediate TIMEOUT abort when receiving large (>2041 bytes, which is the SPDU-size that the ISODE likes to work with) packages from an xtimosi-based implementation (probably most other implementations as well, in fact). It seems to be a flaw in the ISODE API, and the workaround (for ISODE users) is to either *not use an explicit timeout (switching to either blocking or nonblocking mode), or to check that the timer really has expired before closing the connection.*

The next step in the installation is to modify the makefile in the toplevel **YAZ** directory. The place to change is in the top of the file, and is clearly marked with a comment.

Now run `make` in the **YAZ** toplevel directory (do a `"make clean"` first, if the system has been previously made without OSI support). Use the **YAZ ztest** and **client** demo programs to verify that OSI communication works OK. Then, you can go ahead and try to talk to other implementations.

NOTE: Our interoperability experience is limited to version 7 of the Nordic SR-Nett package, which has had several protocol errors fixed from the earlier releases. If you have problems or successes in interoperating with other implementations, we'd be glad to hear about it, or to help you make things work, as our resources allow.

If you write your own applications based on **YAZ**, and you wish to include OSI support, the procedure is equally simple. You should include the `xmosi.h` header file in addition to `comstack.h`. `xmosi.h` will define the manifest constant `mosi_type`, which you should pass to the `cs_create()` function. In addition, you should use the function `mosi_strtoaddr()` rather than `tcpip_strtoaddr()` when you need to prepare an address.

When you link your application, you should include (after the `libyaz.a` library) the `libmosi.a` library, and the `librfc.a` library provided with **YAZ** (for OSI transport).

As always, it can be very useful, if not essential, to have a look at the example applications to see how things are done.

6.7.2 OSI Transport

Xtimosi requires an implementation of the OSI transport service under the X/OPEN XTI API. We provide an implementation of the RFC1006 encapsulation of OSI/TP0 in TCP/IP (through the Berkeley Sockets API), as an independent part of **YAZ** (it's found under the `rfc1006` directory). If you have access to an OSI transport provider under XTI, you should be able to make that work too, although it may require tinkering with the `mosi_strtoaddr()` function.

6.7.3 Presentation Context Management

To simplify the implementation, we use Peter Furniss' alternative (PRF) option format for the Control of the presentation negotiation phase. This format is enabled by default when you compile xtimosi.

The current version of **YAZ** does *not* support presentation-layer negotiation of response record formats. The primary reason is that we have had access to no other SR or Z39.50 implementations over OSI that used this method. Secondarily, we believe that the EXPLAIN facility is a superior mechanism for relaying target capabilities in this respect. This is not to say that we have no intentions of supporting presentation context negotiation - we have just hitherto given it a lower priority than other aspects of the protocol.

One thing is certain: The addition of this capability to **YAZ** should have only a minimal impact on existing applications, and on the interface to the software in general. Most likely, we will add an extra layer of interface to the processing of EXPLAIN records, which will convert back and forth between `oident` records (see section 3.3 (Object Identifiers)) and direct or indirect references, given the current association setup. Implementations based on any of the higher-level interfaces will most likely not have to be changed at all.

6.8 Summary and Synopsis

```
#include <comstack.h>

#include <tcpip.h>      /* this is for TCP/IP support   */
#include <xmosi.h>      /* and this is for mOSI support */

COMSTACK cs_create(CS_TYPE type, int blocking, int protocol);

COMSTACK cs_createbysocket(int s, CS_TYPE type, int blocking,
                              int protocol);

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);

int cs_close(COMSTACK handle);

int cs_look(COMSTACK handle);

struct sockaddr_in *tcpip_strtoaddr(char *str);

struct netbuf *mosi_strtoaddr(char *str);

extern int cs_errno;

void cs_perror(COMSTACK handle char *message);

const char *cs_stackerr(COMSTACK handle);

extern const char *cs_errlist[];
```

7 Making an IR Interface for Your Database with YAZ

7.1 Introduction

NOTE: If you aren't into documentation, a good way to learn how the backend interface works is to look at *the backend.h file. Then, look at the small dummy-server in server/ztest.c. Finally, you can have a look at the seshigh.c file, which is where most of the logic of the frontend server is located. 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/SR, **YAZ** basically offers your two options. You can use the APIs provided by the **ASN**, **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 realise 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 toplevel 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.

7.2 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 prototypes and structure definitions. You are required to provide the **main()** routine for the server (which can be quite simple), as well as functions 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 functions will perform the tasks of:

- Initialization.
- Searching.
- Fetching records.
- Scanning the database index (if you wish to implement SCAN).

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

Because the model where pipes or sockets are used to access the backend database is a fairly common one, we have added a mechanism that allows this communication to take place asynchronously. In this mode, the frontend server doesn't have to block while the backend database is processing a request, but can wait for additional PDUs from the client.

7.3 The Backend API

The headers files that you need to use the interface are in the include/ directory. They are called `statserv.h` and `backend.h`. They will include other files from the `include` 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 toplevel **YAZ** directory, everything you need to create your server is put the lib/libyaz.a library. If you want OSI as well, you'll also need to link in the `libmosi.a` library from the xtimosi distribution (see the mosi.txt file), a well as the `lib/librfc.a` library (to provide OSI transport over RFC1006/TCP).

7.4 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);
```

Statserv_main will establish listening sockets according to the parameters given. When connection requests are received, the event handler will typically **fork()** to handle the new request. If you do use global variables, you should be aware, then, that these cannot be shared between associations, unless you explicitly disallow forking by command line parameters (we advise against this for any purposes except debugging, as a crash or hang in the server process will affect all users currently signed on to the server).

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.

int loglevel

Set this by ORing the constants defined in include/log.h.

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_SR` or `PROTO_Z3950`. Default is `PROTO_Z39_50`.

int idle_timeout;

Maximum session idletime, in minutes. Zero indicates no (infinite) timeout. Default is 120 minutes.

int maxrecordsize;

Maximum permissible record (message) size. Default is 1Mb. 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.

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 before you call

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

Note that you should generally update this structure *before* calling `statserv_main()`.

7.5 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.

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

This function is called once for each new connection request, after a new process has been forked, and an initRequest has been received from the client. The parameter and result structures are defined as

```
typedef struct bend_initrequest
{
    char *configname;
} 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;
```

The configname of bend_initrequest is currently always set to "default-config". We haven't had use for putting anything special in the initrequest yet, but something might go there if the need arises (account/password info would be obvious).

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.

```
bend_searchresult *bend_search(void *handle, bend_searchrequest *r,
                               int *fd);
bend_searchresult *bend_searchresponse(void *handle);

typedef struct bend_searchrequest
{
    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_Query *query;       /* query structure */
} bend_searchrequest;

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

The first thing to notice about the search request interface (as well as all of the following requests), is that it consists of two separate functions. The idea is to provide a simple facility for asynchronous communication with the backend server. When a searchrequest comes in, the server frontend will fill out the bend_searchrequest tructure, and call the bend_search function. The fd argument will point to an integer variable. If you are able to do asynchronous I/O with your database server, you should set *fd to the file descriptor you use for the communication, and return a null pointer. The server frontend will then select() on the *fd, and will call bend_searchresult when it sees that data is available. If you don't support asynchronous I/O, you should return a pointer to the bend_searchresult immediately, and leave *fd untouched. This construction is common to all of the bend_ functions (except bend_init). Note that you can choose to support this facility in none, any, or all of the bend_ functions, and you can respond differently on each request at run-time. The server frontend will adapt accordingly.

The bend_searchrequest is a fairly close approximation of a protocol searchRequest PDU. 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/proto.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_getentbyoid` function provided by **YAZ**.

The result 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.

```
bend_fetchresult *bend_fetch(void *handle, bend_fetchrequest *r,
                             int *fd);
bend_fetchresult *bend_fetchresponse(void *handle);

typedef struct bend_fetchrequest
{
    char *setname;       /* set name */
    int number;          /* record number */
    oid_value format;
} bend_fetchrequest;

typedef struct bend_fetchresult
{
    char *basename;      /* name of database that provided record */
    int len;             /* length of record */
    char *record;        /* record */
    int last_in_set;     /* is it?  */
    oid_value format;
    int errcode;         /* 0==success */
    char *errstring;     /* system error string or NULL */
} bend_fetchresult;
```

NOTE: The `bend_fetchresponse()` function is not yet supported in this version of the software. Your implementation of `bend_fetch()` should always return a pointer to a `bend_fetchresult`.

The frontend server calls `bend_fetch` when it needs database records to fulfill a searchRequest or a presentRequest. 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 3.3 (Object Identifiers)). The value `VAL_NONE` 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.

In the result 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 currently be interpreted as a global error pertaining to the set, and will be returned in a nonSurrogateDiagnostic.

NOTE: This is silly. Add a flag to say which is which.

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.*

```
bend_deleteresult *bend_delete(void *handle, bend_deleterequest *r,
                               int *fd);
bend_deleteresult *bend_deleteresponse(void *handle);

typedef struct bend_deleterequest
{
    char *setname;
} bend_deleterequest;

typedef struct bend_deleteresult
{
    int errcode;         /* 0==success */
    char *errstring;     /* system error string or NULL */
} bend_deleteresult;
```

NOTE: The "delete" function is not yet supported in this version of the software.

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?

```
bend_scanresult *bend_scan(void *handle, bend_scanrequest *r,
    int *fd);
bend_scanresult *bend_scanresponse(void *handle);

typedef struct bend_scanrequest
{
    int num_bases;      /* number of elements in databaselist */
    char **basenames;   /* databases to search */
    Z_AttributesPlusTerm *term;
    int term_position;  /* desired index of term in result list */
    int num_entries;    /* number of entries requested */
} bend_scanrequest;

typedef struct bend_scanresult
{
    int num_entries;
```

```
    struct scan_entry
    {
        char *term;
        int occurrences;
    } *entries;
    int term_position;
    enum
    {
        BEND_SCAN_SUCCESS,
        BEND_SCAN_PARTIAL
    } status;
    int errcode;
    char *errstring;
} bend_scanresult;
```

NOTE: The `bend_scanresponse()` function is not yet supported in this version of the software. Your implementation of `bend_scan()` should always return a pointer to a `bend_scanresult`.

7.6 Application Invocation

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

```
appname [-szSu -a apdufile -l logfile -v loglevel]
[listener ...]
```

The options are

-a

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

-S

Don't fork 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.

-s

Use the SR protocol.

-z

Use the Z39.50 protocol (default). These two options complement eachother. 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

The logfile.

-v

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

-u

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

Working directory.

-i

Use this when running from the `inetd` server.

-t

Idle session timeout, in minutes.

-k

Maximum record size/message size, in kilobytes.

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

For TCP, an address has the form

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

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

For osi, the address form is

```
[t-selector /] hostname | IP-number [: portnumber]
```

The transport selector is given as a string of hex digits (with an even number of digits). The default port number is 102 (RFC1006 port).

Examples

```
tcp:dranet.dra.com

osi:0402/dbserver.osiworld.com:3000
```

In both cases, the special hostname "@" is mapped to the address INADDR_ANY, which causes the server to listen on any local interface. To start the server listening on the registered ports for Z39.50 and SR over OSI/RFC1006, and to drop root privileges once the ports are bound, execute the server like this (from a root shell):

```
my-server -u daemon tcp:@ -s osi:@
```

You can replace `daemon` with another user, eg. your own account, or a dedicated IR server account. `my-server` should be the name of your server application. You can test the procedure with the `ztest` application.

7.7 Summary and Synopsis

```
#include <backend.h>

bend_initresult *bend_init(bend_initrequest *r);

bend_searchresult *bend_search(void *handle, bend_searchrequest *r,
                                 int *fd);

bend_searchresult *bend_searchresponse(void *handle);

bend_fetchresult *bend_fetch(void *handle, bend_fetchrequest *r,
                               int *fd);

bend_fetchresult *bend_fetchresponse(void *handle);

bend_scanresult *bend_scan(void *handle, bend_scanrequest *r, int *fd);

bend_scanresult *bend_scanresponse(void *handle);

bend_deleteresult *bend_delete(void *handle, bend_deleterequest *r,
                                  int *fd);

bend_deleteresult *bend_deleteresponse(void *handle);

void bend_close(void *handle);
```

8 Future Directions

The software has been successfully ported to the Mac as well as Windows NT/95 - we'd like to test those ports better and make sure they work as they should.

We have a new and better version of the frontend server on the drawing board. Resources and external commitments will govern when we'll be able to do something real with it. Fetures should include greater flexibility, greter support for access/resource control, and easy support for Explain (possibly with Zebra as an extra database engine).

We now support all PDUs of Z39.50-1995. If there is one of the supporting structures that you need but can't find in the prt*.h files, send us a note; it may be on its way.

The 'retrieval' module needs to be finalized and documented. We think it can form a useful resource for people dealing with complex record structures, but for now, you'll mostly have to chew through the code yourself to make use of it. Not acceptable.

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.

9 License

9.1 Index Data Copyright

Copyright © 1995-2000 Index Data.

Permission to use, copy, modify, distribute, and sell this software and its documentation, in whole or in part, for any purpose, is hereby granted, provided that:

1. This copyright and permission notice appear in all copies of the software and its documentation. Notices of copyright or attribution which appear at the beginning of any file must remain unchanged.

2. The names of Index Data or the individual authors may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT WARRANTY OF ANY KIND, EXPRESS, IMPLIED, OR OTHERWISE, INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT SHALL INDEX DATA BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT OR CONSEQUENTIAL DAMAGES OF ANY KIND, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER OR NOT ADVISED OF THE POSSIBILITY OF DAMAGE, AND ON ANY THEORY OF LIABILITY, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

9.2 Additional Copyright Statements

The optional CCL query language interpreter is covered by the following license:

Copyright © 1995, the EUROPAGATE consortium (see below).

The EUROPAGATE consortium members are:

University College Dublin Danmarks Teknologiske Videnscenter An Chomhairle Leabharlanna Consejo Superior de Investigaciones Cientificas

Permission to use, copy, modify, distribute, and sell this software and its documentation, in whole or in part, for any purpose, is hereby granted, provided that:

1. This copyright and permission notice appear in all copies of the software and its documentation. Notices of copyright or attribution which appear at the beginning of any file must remain unchanged.

2. The names of EUROPAGATE or the project partners may not be used to endorse or promote products derived from this software without specific prior written permission.

3. Users of this software (implementors and gateway operators) agree to inform the EUROPAGATE consortium of their use of the software. This information will be used to evaluate the EUROPAGATE project and the software, and to plan further developments. The consortium may use the information in later publications.

4. Users of this software agree to make their best efforts, when documenting their use of the software, to acknowledge the EUROPAGATE consortium, and the role played by the software in their work.

THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT WARRANTY OF ANY KIND, EXPRESS, IMPLIED, OR OTHERWISE, INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MER-

10 About Index Data

Index Data is a consulting and software-development enterprise that specialises 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 hypermedia 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
Ryesgade 3
DK-2200 Copenhagen N

Phone: +45 3536 3672
Fax  : +45 3536 0449
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.

Yet Another. adj. 1. Of your own work: A humorous allusion often used in titles to acknowledge that the topic is not original, though the content is. As in "Yet Another AI Group" or "Yet Another Simulated Annealing Algorithm". 2. Of others' work: Describes something of which there are already far too many.