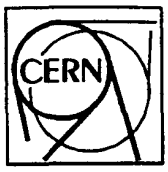
LL



GENÈVE, SUISSE
GENEVA, SWITZERLAND

ИНСТИТУТ ФИЗИКИ ВЫСОКИХ ЭНЕРГИЙ

EUROPEAN LABORATORY FOR PARTICLE PHYSICS

COLLABORATION ON UNK CONTROLS

CERN-PS
September 3, 1994

YAQ - Yet Another eQuipment access package

E.A. Jarkov, V.I. Kovaltsov
Insitute for High Energy Phsics, Protvino, Russia

Abstract

This note describes Version 2.0 of YAQ (Yet Another eQuipment access package). The package provides a unified way to access devices working under different equipment access protocols. It includes tools for maintaining the equipment database and C libraries to transparently access equipment in a distributed heterogeneous networking environment

Contents

1. Introduction
2. Terms and scope
3. Basic data structures
4. YAQ components
5. YAQ database management
6. Examples
7. Portability
8. Current status
9. Reference pages

Contents

1 Introduction **2**

2 Terms and scope **2**

3 Basic data structures **3**

4 YAQ components **3**
4.1 YQ 4
4.2 YH 5
4.3 YR 6
4.4 YD 7
4.5 YM 7
4.6 NCVT 7

5 YAQ database management **8**
5.1 The DBRT control file 8
5.2 dbrtgen 10

6 Examples **10**
6.1 A simple client example 10
6.2 A simple device server example 11
6.3 A client example with the yqRegister subroutines 12
6.4 An example with protocol specific data 13

7 Portability **14**

8 Current status **15**

Reference Pages **16**
ydClose 16
ydCreate 16
ydOpen 16
ydFetch 17
ydStore 17
yqCall 18
yqRegisterServer 19
yqRegisterProtocol 19
yqRegisterFamily 19
yqRegisterDevice 19
yqRegisterAction 19
yqLocateAction 22
yqLocateDevice 22
yrCode 23
yrLog 23
yrString 23

yrSetHandler 25
yrSetMode 25
yhProtocol 26
yhWait 28
Ncvt 29

List of Figures

1 YAQ package context 3
2 YAQ basic data structures 4
3 YAQ components 5
4 The YQ subroutines 6
5 Structure of DBRT 8

List of Tables

1 Introduction

Version 2.0 is based on Version 1.0, which is in one's turn based on ideas of the CERN PS [1] and CERN SPS-LEP [2] equipment access packages. Version 1.0 has been developed in the beginning of 1993 and is used by the UNK control teams at CERN and in Protvino. While Version 2.0 is almost backward compatible with Version 1.0, the source code is completely new. The code has been rewritten with an aim to make it more suitable for different equipment access protocols which are in use at CERN and which probably will be used in the UNK controls infrastructure.

YAQ sources can be found in dxcern.cern.ch:/u/pz/ezharkov/yaq. A postscript copy of this document is in dxcern.cern.ch:/u/pz/ezharkov/yaq/doc/yaq.ps. Any comments, suggestions, etc. are greatly welcomed and should be sent to ezharkov@dxcern.cern.ch.

2 Terms and scope

The YAQ context diagram is shown in Figure 1.

A **device** is a piece of hardware (power supply, for example) or a software object (a database, for example). A device is accessed by means of **actions**. Each device belongs to a **family**. All the devices of one family have the same set of actions.

A set of agreements on the device presentation in a database, inside the application program and the device server is called **equipment access protocol** or simply **protocol**.

YAQ is used by the application program to make a call to a device. YAQ is used by the device server to decode the equipment call message coming from the application program and return back the result. The way how a device server communicates with a device is not covered by YAQ, it is competely up to the author of the device server.

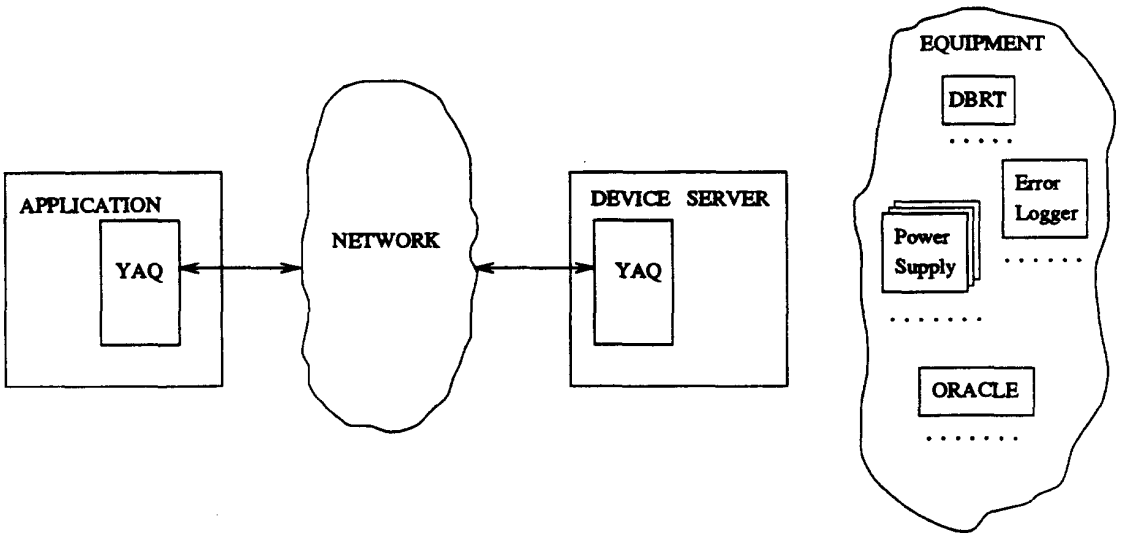


Figure 1: YAQ package context

3 Basic data structures

YAQ basic data structures and their relationships are shown in Figure 2. This is a logical view of the structures. Physically, in the equipment database, inside the application program or the device server, the structures can be somewhat different. For example, in ([1, 3]) numbers are used for the device and action identification on the level of the device server, and there is no need to send device and action names to the device server.

Family, device and action structures consist of two parts, mandatory, which is common for all protocols, and protocol specific. The protocol specific parts are defined by the `family format`, `device format` and `action format` fields in the protocol description. YAQ need not to be rebuild when a specific structure is changing. A device protocol specific structure can include, for example, the member number [3] or a field bus address [2]. A protocol specific structure need not to be defined at the time YAQ is built. Protocol specific data are normally defined in an equipment database, selected and sent transparently to a device server.

4 YAQ components

Physically YAQ is built as one library - `libyaq`, logically it consists of a number of components as shown in Figure 3. The names of the components (and the names of subroutines inluded) changed in Version 2. The YQ component was previously called Eqp, YD - Dbrt, YR - Err, i.e. exactly as in [1], and the reason for changing was to make it possible in principle to use both packages in one application.

An application program mainly deals with `YQ` (Section 4.1). There are basically two types of calls an applications program makes, the first one is to create a device and an

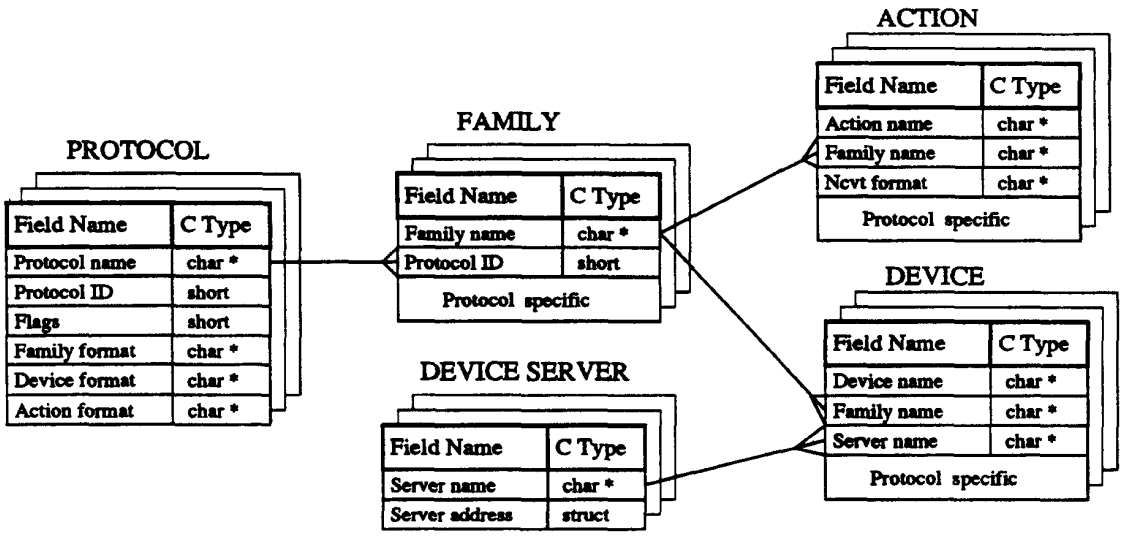


Figure 2: YAQ basic data structures

action data structures, the second one is an actual call to a device.

The device and action descriptions are normally (but not necessarily) stored in a run time database (DBRT, see Section 5). YQ itself maintains a cache of description data, it first checks whether the data requested by the user are already in the cache. If not, it calls YD (Section 4.4) to select data from DBRT. Data in DBRT are in network representation, YQ converts them to host representation using NCVT (Section 4.6).

When an equipment call is made, YQ forms a message to a device server, converts the message to network representation using NCVT and sends it over network using YM (Section 4.5).

A device server is basically in an infinite loop waiting for a message from an application program. A message first comes to YM, then to YH (section Section 4.2). Having received a message the server decodes it, fills up an interface structure and passes control to an appropriate protocol module (one server can support devices working under different protocols).

When protocol module finishes its job, control returns to YH again. YH forms the result message and send it to the application program.

All possible erroneous situations are handled by a call to YR (Section 4.3).

4.1 YQ

An application program mainly deals with YQ.

YQ consists basically of two groups of subroutines, one to create a YQ's data structure, another to make an actual call to a device (see Figure 4). The first group in one's turn is also divided into two subgroups, the **Locate** and **Register** subroutines. A **Locate** subroutine creates a data structure on the base of information, located in the equipment database. A **Register** subroutine creates a data structure on the base of information

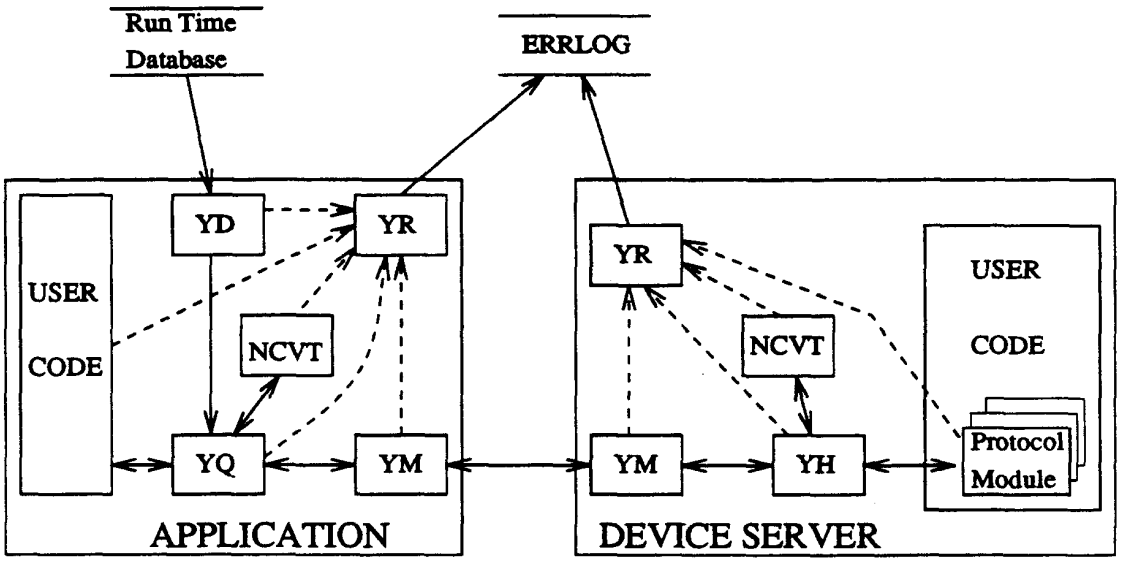


Figure 3: YAQ components

directly supplied by the user. Each of these groups consists of general and protocol specific subroutines. Protocol specific subroutines are included for compatibility with [1, 2, 3].

Refer to yqLocate Reference Page, yqRegister Reference Page and yqCall Reference Page for the complete description of the `YQ` subroutines.

4.2 YH

The idea of `YH` library comes from Message Handler [2]. `YH` easies writing device servers, taking control over all the communication problems between the device server and the application program.

`YH` supports one or more protocol modules in one device server. Each protocol module must be registered. This is done by the `yhRegisterProtocol` subroutine (see yhRegisterProtocol Reference Page). Once all protocol modules are registered, the device server enters an infinite loop waiting for an equipment message. This is done by the `yhWait` subroutine (see yhWait Reference Page).

`YH` interacts with the protocol modules using a C structure. When `YH` receives an equipment call, it fills up a number of fields in this structure and passes control to the appropriate protocol module. When the protocol module exits, it uses the interfacing structure to return the result.

Errors are handled by a call to `YR`. In case of error the error code and error string are sent automatically to the application program.

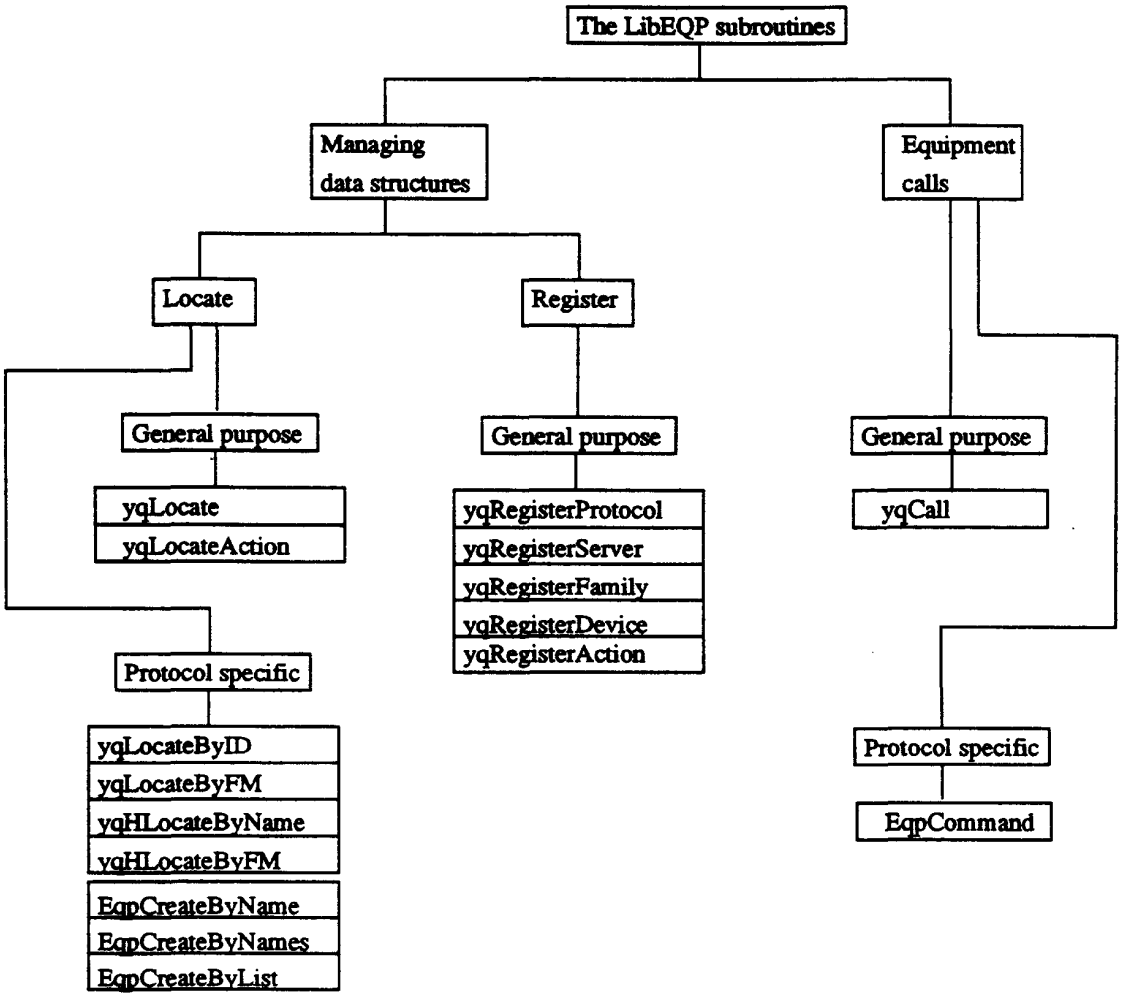


Figure 4: The YQ subroutines

4.3 YR

Error handling in YAQ is based on use of C macros. For example, a call to YR may look like:

```
if (strlen(name) >= bufsize) xyzERROR_NameTooLong(name, bufsize);
```

where xyzERROR_NameTooLong is defined in an include file as:

```
#define xyzERROR_NameTooLong(name,size) yrLog1M(xyzERRNO_NameTooLong,\
      "Name %s is too long. Buffer size %d", name, size)
```

Macros allow to switch quite easily to a different way of error handling. For instance, very often errors are subdivided into categories by severity: warning, fatal, etc. The macro from the previous example can be redefined to something like:

```
#define xyzERR_NameTooLong(name,size) yrLog1M(xyzERRNO_NameTooLong, FATAL, \
      "Name %s is too long. Buffer size %d", name, size)
```

The error strings in YAQ can be kept separately from the source code, in a sort of message files, and selected from these files using the error code as a key. Keeping error messages separately from the source code allows to have different message files for several national languages. The xyzERROR_NameTooLong macro takes the form:

```
#define XyzERR_NameTooLong(name,size) yrLog1(xyzERRNO_NameTooLong, name, size)
```

When control passes to YR, its behaviour is defined by the mode currently set, the modes are PRINT - print error message, EXIT - exit on receiving an error call, LOG - log the error information to the errlog archive. In addition to the mode, a user can set up his own error handler.

Refer to YR reference pages for detailed information.

4.4 YD

YD is used by YQ to get a device description from DBRT, by the **dbrtgen** program (Section 5.2) to create a **gdbm** based DBRT, and it can also be used by an application program directly.

YD consists of **ydOpen** to open a DBRT database, **ydFetch** to get a record from DBRT, **ydStore** to put a record to the **gdbm** based DBRT. Refer to the corresponding reference pages for detailed information.

4.5 YM

YM is the communication component. YM is not supposed to be used directly by an application program or a device server programmer, it is used internally by YQ. It consists of subroutines to open a connection and to send/receive a block of data over network. YM can work on the top of different software interfaces, CERN RPC, UDP sockets, TCP sockets etc., presently it works via the UDP sockets.

4.6 NCVT

Different computers in the distributed heterogeneous networking environment have different representation and alignment rules of theirs basic data types. To be able to speak to each other computers must use a common machine independent representation of data (network representation). The NCVT subroutines are to convert data from the local machine representation (host representation) to network representation and vise versa. NCVT is based on ideas of the DTM package [4]. As distinct from DTM, NCVT supports pointers, variable arrays, structures, etc.

NCVT also supports conversion of variable C-call argument list.

Refer to Ncvt Reference Page for the complete description of the NCVT subroutines.

5 YAQ database management

The device and action description in YAQ is normally stored in a run time database (DBRT), which can be considered as a one-table single-key database. Different types of records are stored in the database with a key prefixed by the record type. All the information is in network format. Primary source of data is normally in an ORACLE database. As an alternative to ORACLE ordinary ASCII files can be used too.

A DBRT database can be organized directly under ORACLE, or in an intermediate storage built on the base of GNU **gdbm**. A **gdbm**-based database is created from an ORACLE database (or from a set of ASCII files) by the **dbrtgen** program (Section 5.2).

In case of **gdbm** an application program can be linked with the **gdbm** library and read data directly from a **gdbm** file. A **gdbm** file can be accessed from a number of machines via NFS, but all these machines should be compatible from the point of view of data representation. The more universal way is to have a DBRT server. In YAQ such a server, called **dbrtora**, made as an ordinary device server, selects data directly from an ORACLE database (of course it can select data from a **gdbm** database too).

The **dbrtgen** and **dbrtora** programs are driven by a special control file (Section 5.1), which contains information on what and how is to be selected from ORACLE.

The structure of DBRT is shown in Figure 5. In most cases adding new record type

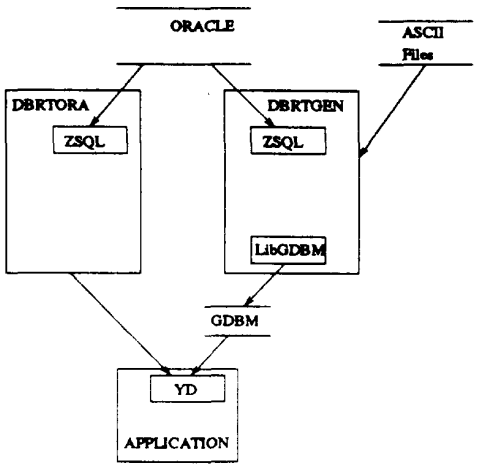


Figure 5: Structure of DBRT

to DBRT, or modifying the structure of an old one does not require the **dbrtgen** code to be changed or rebuilt.

5.1 The DBRT control file

For each DBRT record type there is a statement in the control file that defines the record structure and the way data are extracted from ORACLE (or from an ASCII file). Each of these statements has the following syntax:

```
  record {
    comment     "string";
    keyid       "string" or number;
    ncvt        "string";
    sqlgen      "string";
    sqlget      "string";
    file        "string";
};
```

where:

keyid is a record type;

ncvt is an ncvt format string, that defines the record format and is used to convert data to network representation;

sqlgen is a SQL statement used by the **dbrtgen** program to select data from ORACLE;

sqlget is a SQL statement used by **dbrtora** server to directly select data from ORACLE;

file is a name of an ASCII file (if any) containing data to be put to **gdbm**.

Consider an example:

```
record {
  comment  "Device By name";
  keyid    "device";
  ncvt     "zzi";
  sqlgen   "select name, family_name, server_name, device_id "
           "from equip.device ";
  sqlget   "select family_name, server_name, device_id "
           "from equip.device "
           "where name='%s'";
  file     "device.dat";
}
```

In this example the record type is "device by name", all the records of this type will be stored in DBRT with a key equal to **name** (the first item in the sqlgen select list), prefixed by the string "device". For instance, if the device name is "mydevice" then the key in **gdbm** will be "device.mydevice". The structure of this record is defined by the ncvt string "zzi", that corresponds to the C structure:

```
struct {
  char  *family_name;
  char  *server_name;
  int   device_id;
};
```

The control file is normally passed through the C preprocessor, this allows to use include files and avoid explicit specification for **keyid** and **ncvt**.

5.2 dbrtgen

The **dbrtgen** program generates a **gdbm** based DBRT. **dbrtgen** is driven by a control file (Section 5.1). Depending on the description in the control file it selects data either from ORACLE or from an ASCII files.

6 Examples

6.1 A simple client example

The simplest case for equipment access client is when the device and action descriptions are in an equipment database. Consider, for example, a device called "Calculator", which is able to add, subtract, multiply and divide integer numbers, a client program may look like:

```
#include <stdio.h>
#include <yaqh.h>

main() {
  int sum, difference, product, quotient;
  int item1 = 4, item2 = 2;
  yqCall("Calculator", "Add", item1, item2, &sum);
  yqCall("Calculator", "Sub", item1, item2, &difference);
  yqCall("Calculator", "Mul", item1, item2, &product);
  yqCall("Calculator", "Div", item1, item2, &quotient);
  printf("Result = %d, %d, %d, %d\n",sum,difference,product,quotient);
}
```

In the example above device and action are specified by name, the yqCall subroutine internally calls yqLocateDevice and yqLocateAction. This may be not good by performance or some other reasons, so one can rewrite the above example as follows:

```
#include <stdio.h>
#include <yaqh.h>

main() {
  int sum, difference, product, quotient;
  int item1 = 4, item2 = 2;
  yqDevice_t *calculator;
  yqAction_t *add, *sub, *mul, *div;
  calculator = yqLocateDevice("Calculator");
  add = yqLocateAction("Add", "Calculator_family");
  sub = yqLocateAction("Sub", "Calculator_family");
  mul = yqLocateAction("Mul", "Calculator_family");
  div = yqLocateAction("Div", "Calculator_family");
  yqCall(calculator, add, item1, item2, &sum);
  yqCall(calculator, sub, item1, item2, &difference);
  yqCall(calculator, mul, item1, item2, &product);
  yqCall(calculator, div, item1, item2, &quotient);
  printf("Result = %d, %d, %d, %d\n",sum,difference,product,quotient);
}
```

6.2 A simple device server example

Our server works via the UDP sockets. In case of the UDP sockets the address is specified by a structure of type **ymUDP_t** consisting of a hostname and a UDP port (hostname, in fact, is not used by the yhRegisterServer subroutine).

We define our protocol as not having specific data for neither family, nor device, nor action, this is specified by the NULL pointers in the call to the yhRegisterProtocol subroutine.

We register our protocol with the **yqPF_MHDEVNAM** and **yqPF_MHACTNAM** flags, thus requesting the device and action names to be sent from the client to the server (by default family, device and action names are not sent to a server). Having the device and action names we check if this is a right device and we make a switch by the action name to a particular procedure.

We register our protocol with the **yqPF_MHACTFMT** and **yqPF_ARGHOST** flags, thus requesting the action format string to be sent from the client to the server and the action parameters to be converted automatically to host representation. We just cast the **h->arg** pointer to a data type corresponding to the action's parameter list.

```
#include <stdio.h>
#include <yaqr.h>
#include <yaqm.h>
#include <yaqq.h>
#include <yaqh.h>

#define FLAGS yqPF_ARGHOST|yqPF_MHDEVNAM|yqPF_MHACTNAM|yqPF_MHACTFMT
static ymUDP_t srvaddr = { NULL, 1994 };

int myprotocol();

main() {
  yrSetMode(yrMODE_PRINT|yrMODE_EXIT);
  yhRegisterServer(ymCMT_UDP,&srvaddr);
  yhRegisterProtocol(myprotocol, 1, FLAGS, NULL,NULL,NULL);
  yhWait(0,1000);
}

int myprotocol(yhHeader_t *h) {
  if (strcmp(h->device->name,"Calculator") == 0) return calculator(h);
  return yrLog("Unknown device");
}

calculator(yhHeader_t *h) {
  yhAction_t *a = h->action;
  struct params {
    int i1;
    int i2;
    int *res; } *p = (struct params *)h->arg;
  if (strcmp(a->name, "Add") == 0) add(p->i1, p->i2, p->res);
  else if (strcmp(a->name, "Sub") == 0) sub(p->i1, p->i2, p->res);
  else if (strcmp(a->name, "Mul") == 0) mul(p->i1, p->i2, p->res);
  else if (strcmp(a->name, "Div") == 0) div(p->i1, p->i2, p->res);
  else yrLog("Unknown action");
```

```
  return 0;
}

add(int i1, int i2, int *res) { *res = i1 + i2; }
sub(int i1, int i2, int *res) { *res = i1 - i2; }
mul(int i1, int i2, int *res) { *res = i1 * i2; }
div(int i1, int i2, int *res) { *res = i1 / i2; }
```

6.3 A client example with the yqRegister subroutines

In Section 6.1 we showed an example of a simple client which locates device and action descriptions in an equipment database. Alternatively all this information can be directly specified by the user in the form of parameters to the yqRegister subroutines. There is at least one interesting moment in using of the yqRegister subroutines - the example we provide below can be easily compiled, linked and hopefully it will even work. The above example with the yqLocate subroutines is less easy to run because it require a database.

We use the same parameters for yqRegisterServer and yqRegisterProtocol as in the server example above (this is a must for a client and a server that want to speak to each other).

```
#include <stdio.h>
#include <yaqr.h>
#include <yaqm.h>
#include <yaqq.h>
#include <yaqh.h>

#define FLAGS yqPF_ARGHOST|yqPF_MHDEVNAM|yqPF_MHACTNAM|yqPF_MHACTFMT
static ymUDP_t srvaddr = { "localhost", 1994 };

main() {
  int sum, difference, product, quotient;
  int item1 = 4, item2 = 2;

  yrSetMode(yrMODE_PRINT|yrMODE_EXIT);

  yqRegisterServer("myserver", ymCMT_UDP, &srvaddr);
  yqRegisterProtocol("myprotocol", 1, FLAGS, NULL, NULL, NULL);
  yqRegisterFamily("Calculator_family", "myprotocol", NULL);

  yqRegisterDevice("Calculator", "Calculator_family", "myserver", NULL);
  yqRegisterAction("Add", "Calculator_family", "<ii >*i", NULL);
  yqRegisterAction("Sub", "Calculator_family", "<ii >*i", NULL);
  yqRegisterAction("Mul", "Calculator_family", "<ii >*i", NULL);
  yqRegisterAction("Div", "Calculator_family", "<ii >*i", NULL);

  yqCall("Calculator", "Add", item1, item2, &sum);
  yqCall("Calculator", "Sub", item1, item2, &difference);
  yqCall("Calculator", "Mul", item1, item2, &product);
  yqCall("Calculator", "Div", item1, item2, &quotient);
  printf("Result = %d, %d, %d, %d\n",sum,difference,product,quotient);
}
```

6.4 An example with protocol specific data

In the above examples we use a protocol without protocol specific data for family, device and action. In the example below we use a bit different protocol. According to this protocol, each device and action has an additional parameter, an integer identifier for the device or action. Device and action names are not sent to a server.

```
/* A client example using protocol specific data */
#include <stdio.h>
#include <yaqr.h>
#include <yaqm.h>
#include <yaqq.h>
#include <yaqh.h>

#define FLAGS yqPF_ARGHOST|yqPF_MHACTFMT
static ymUDP_t srvaddr = { "localhost", 1994 };

struct protocol_specific_device {
  int id;
} devid = { 1 };

struct protocol_specific_action {
  int id;
} addid = { 1 }, subid = { 2 }, mulid = { 3 }, divid = { 4 };

main() {
  int sum, difference, product, quotient;
  int item1 = 4, item2 = 2;

  yrSetMode(yrMODE_PRINT|yrMODE_EXIT);

  yqRegisterServer("myserver", ymCMT_UDP, &srvaddr);
  yqRegisterProtocol("myprotocol", 1, FLAGS, NULL, "i", "i");
  yqRegisterFamily("Calculator_family", "myprotocol", NULL);

  yqRegisterDevice("Calculator", "Calculator_family", "myserver", &devid);
  yqRegisterAction("Add", "Calculator_family", "<ii >*i", &addid);
  yqRegisterAction("Sub", "Calculator_family", "<ii >*i", &subid);
  yqRegisterAction("Mul", "Calculator_family", "<ii >*i", &mulid);
  yqRegisterAction("Div", "Calculator_family", "<ii >*i", &divid);

  yqCall("Calculator", "Add", item1, item2, &sum);
  yqCall("Calculator", "Sub", item1, item2, &difference);
  yqCall("Calculator", "Mul", item1, item2, &product);
  yqCall("Calculator", "Div", item1, item2, &quotient);
  printf("Result = %d, %d, %d, %d\n",sum,difference,product,quotient);
}

/* A server example with protocol specific data */
#include <stdio.h>
#include <yaqr.h>
#include <ncvt.h>
#include <yaqm.h>
#include <yaqq.h>
```

```
#include <yaqh.h>

#define FLAGS yqPF_ARGHOST|yqPF_MHACTFMT
static ymUDP_t srvaddr = { NULL, 1994 };

int myprotocol();

typedef struct {
  char *name;
  int id;
} MyDevice_t;

typedef struct {
  char *name;
  char *format;
  int id;
} MyAction_t;

main() {
  yrSetMode(yrMODE_PRINT|yrMODE_EXIT);
  yhRegisterServer(ymCMT_UDP,&srvaddr);
  yhRegisterProtocol(myprotocol, 1, FLAGS, NULL,"i","i");
  yhWait(0,1000);
}

int myprotocol(yhHeader_t *h) {
  MyDevice_t *device = (MyDevice_t *)h->device;
  if (device->id == 1) return calculator(h);
  return yrLog("Unknown device");
}

add(int i1, int i2, int *res) { *res = i1 + i2; }
sub(int i1, int i2, int *res) { *res = i1 - i2; }
mul(int i1, int i2, int *res) { *res = i1 * i2; }
div(int i1, int i2, int *res) { *res = i1 / i2; }

int (*actions[])() = { add, sub, mul, div };

calculator(yhHeader_t *h) {
  MyAction_t *a = (MyAction_t *)h->action;
  int id = a->id;
  if (id < 1 || id > 4) return yrLog("Unknown action");
  return NcvtCall(actions[id-1], h->arg, a->format);
}
```

7 Portability

YAQ has been ported and tested in one or other way on the following platforms:

- DEC Ultrix V4.3
- IBM AIX Risc System/6000
- SunOS V4.x

- Alpha/OSF
- LynxOS/M68K
- Alpha/VMS
- VAX/VMS
- MSDOS
- MS Windows 3.1

The least portable part in YAQ is the NcvtCall subroutine. To be sure that a piece of code will be easily ported to a new system it is better to avoid using floats, doubles and structures passed by value. All these data types are supported for the platforms listed above (except of floats or doubles for Alpha/OSF), but porting to a new system usually require to pay some attention to these data types and to add a number of `#ifdef` directives.

8 Current status

- The protocol specific subroutines is not implemented.
- Only communication via the UDP sockets is currently supported. Packet splitting and retransmission in case of error is not yet done.

References

[1] Franck Di Maio, Alessandro Risso. The CERN-PS Equipment Access Library, *Software Specifications, Version 3*. PS/CO/Note 93-87 (Spec.), CERN, February 1994.

[2] Pierre Charrue, Accessing Equipment in the SPS-LEP Controls Infrastructure: The "SL-EQUIP package", *Software user manual (SUM)*, SL/Note 93-86 (CO), CERN, September 1993.

[3] A.Elin et al., Control protocols for the UNK control system, ICALEPS'93, Berlin, 1993.

[4] Julian Lewis, Distributed shared memory Table Manager, PS/CO.

[5] P.Anderson, V.Fremmery, G.Morpurgo, User Guide to the Network Compiler Remote Procedure Call (NC/RPC), LEP Controls note 97, May 1989.

Name

ydOpen, ydCreate, ydClose - subroutines to open and close a DBRT database

Syntax

#include <yaqd.h>

ydOpen(int *mode*)

ydCreate(char ***name*)

ydClose()

Description

The **ydOpen** subroutine opens a DBRT database specified by either the DBRTFILE or DBRTHOST environment variables, that corresponds to either a local file, accessible via the gdbm library, or to a remote host with a DBRT device server running. The *mode* is a character string having one of the following values:

"r" Open for reading

"w" Open for writing (this works only for a local file)

The **ydCreate** creates a new DBRT database in a local file with the name *name*.

The **ydClose** subroutine closes the DBRT database opened by **ydOpen** or **ydCreate**.

Return values

Upon successful completion, a 0 is returned. Otherwise, a -1 is returned, and more specific error code is available via **yrCode**.

See Also

yrLog(3l), yrCode(3l), yrString(3l)

Name

ydFetch, ydStore - get/put a record from/to a DBRT database

Syntax

#include <yaqd.h>

int ydFetch(char ****buf*, int** *bufsize*, **int ****datasize*, **char *dbrtkey)**

int ydStore(char ****dbrtkey*, **void ****data*, **int datasize)

Description

The `ydFetch` subroutine fetches a record from the DBRT database opened by `ydOpen`. If the DBRT database is not opened, `ydFetch` makes a call to `ydOpen`.

The `ydStore` stores a record to the DBRT database opened by `ydOpen` or `ydCreate`.

Arguments

buf is a pointer to a buffer the selected record is placed to.

bufsize is the size of *buf*.

datasize After a call to `ydFetch` contains the number of bytes actually placed to *buf*. Before a call to should contains the size of the record to be stored.

dbrtkey is a pointer to 0-terminated string containing a key. The key is used by `ydFetch` subroutine to find a record in the database. The record is placed under this key by `ydStore`.

data is a pointer to a buffer containing the record to be stored.

datasize is the size of re to be stored.

Return values

Upon successful completion, a 0 is returned. Otherwise, a -1 is returned, and the error code is available via yrCode.

See Also

ydOpen(3l), ydCreate(3l), yrCode(3l), yrString(3l)

Name

yqCall – equipment access call

Syntax

#include <yaqq.h>

int yqCall(void ****device*, void ****action*** [, ***arg***] ...)**

Description

The **yqCall** subroutines performs an equipment call to the device specified by *device* with the action specified by *action.*

Arguments

device is a pointer to the device name or to the structure returned by yqRegisterDevice or yqLocateDevice.

action is a pointer to the action name or to the structure returned by yqRegisterAction or yqLocateAction.

Upon succesful completion, a 0 is returned. Otherwise a –1 is returned and more detailed information about the error is accessible via **yrCode**.

See Also

yqRegister(3l), yqLocate(3l), yrCode(3l), yrString(3l)

Name

yqRegisterServer, yqRegisterProtocol, yqRegisterFamily, yqRegisterDevice, yqRegisterAction – register a server, a protocol, a family, a device, an action

Syntax

#include <yaqm.h>

#include <yaqq.h>

yqServer_t *yqRegisterServer(char *_name_, int _commtype_, void *_address_)

yqProtocol_t *yqRegisterProtocol(char *_name_, int _id_, int _flags_, char *_family_format_, char *_device_format_, char *_action_format_)

yqFamily_t *yqRegisterFamily(char _name_, yqProtocol_t *_protocol_, void *_specific_)

yqDevice_t *yqRegisterDevice(char *_name_, yqFamily_t *_family_, yqServer_t *_server_, void *_specific_)

yqAction_t *yqRegisterAction(char *_name_, yqFamily_t *_family_, char *_format_, void *_specific_)

Description

These subroutines are normally used by the application program that wants to access devices not described in the equipment database.

The **yqRegisterServer** subroutine registers the device server specified by the communication type *commtype* and the communication address *address*. The server is assigned the name *name*. The *name* must be unique for a given application. Communication types are the UDP sockets (ymCMT_UDP), the TCP sockets (ymCMT_TCP), CERN RPC (ymCMT_CERN_RPC). Communication address is different for different communication types. For the UDP sockets the communication address is represented by a structure of type ymUDP_t:

```
typedef struct {
  char *hostname;
  unsigned short port;
} ymUDP_t;
```

Upon successful completion **yqRegisterServer** returns a pointer to an opaque structure that can be later used in subsequent calls to **yqRegisterDevice**.

The **yqRegisterProtocol** subroutine registers an equipment access protocol. The protocol is assigned the name *name* that must be unique for a given application. The

id argument is an integer protocol identifier that is sent to the device server when an equipment call is made. If the protocol defines a protocol specific data structure for family, device, action, than the corresponding *family_format*, *device_format*, *action_format* argument points to a null–terminated ncvt format string, that desribes this structure. If the protocol specific data are not defined, the corresponding pointer must be set to a NULL value.

The *flags* argument is formed by ORing of the following values:

yqPF_MHFAMNAM family name is sent to a device server.

yqPF_MHDEVNAM device name is sent to a device server.

yqPF_MHACTNAM action name is sent to a device server.

yqPF_MHACTFMT action format is sent to a device server.

yqPF_ARGHOST a device server automatically (before calling a user protocol module) converts action parameters to host representation and after the call converts result to network representation.

Upon successful completion **yqRegisterProtocol** returns a pointer to an opaque data structure that can be later used in subsequent calls to **yqRegisterFamily**.

The **yqRegisterFamily** subroutine registers the family *name* of devices, working under protocol *protocol*. The *protocol* argument is either a pointer returned by **yqRegisterProtocol** or a name of an already registered protocol. If the protocol defines protocol specific family data structure, the *specific* argument of **yqRegisterFamily** points to this family's protocol specific data.

Upon successful completion **yqRegisterFamily** returns a pointer to an opaque data structure that can be later used in subsequent calls to **yqRegisterDevice** or **yqRegisterAction**.

The **yqRegisterDevice** subroutine registers the device identified by *name*, of family *family*, controlled by the server *server*. If the protocol defines protocol specific device data structure, the *specific* argument of **yqRegisterFamily** points to this device's protocol specific data.

Upon successful completion **yqRegisterDevice** returns a pointer to an opaque data structure that can be later used in subsequent equipment calls.

The **yqRegisterAction** subroutine registers the action *name* of family *family*. The *format* is a pointer to an ncvt conversion string that describes the format of the action parameters. If the protocol defines protocol specific action data structure, the *specific* argument of **yqRegisterFamily** points to this action's protocol specific data.

Upon successful completion **yqRegisterAction** returns a pointer to an opaque data structure that can be later used in subsequent equipment calls.

On failure, all these subroutines return a NULL pointer. A more detailed information about the error can be found via **yrCode**.

See Also

yqCall(3l), Ncvt(3l), yrCode(3l), yrString(3l)

Name

yqLocateDevice, yqLocateAction – locate description of a device or an action in an equipment database

Syntax

#include <yaqq.h>

yqDevice_t *yqLocateDevice(char *_name_)

yqAction_t *yqLocateAction(char *_name_, void *family)

Description

The `yqLocateDevice` subroutine checks first whether the device specified by *name* is already registered in the calling program's local cache. If this is a new device, the description of the device is located in the equipment database.

The `yqLocateAction` subroutines checks first whether the action specified by *name* for the family specified by *family* is already registered in the calling program's cache. If this is a new action, the description of the action is located in the equipment database. The *family* parametes pointer to the name of a family or to a family reference returned by `yqRegisterFamily`.

Return values

Upon successful completion a pointer to an opaque data structure inside the calling program is returned.

On failure, a NULL pointer is returned.

Diagnostics

See Also

yqCall(3l), yqRegister(3l), yrCode(3l), yrString(3l)

yrLog

yrLog, yrLog1, yrLog1M, yrLog2, yrLog2M – error handling subroutines

Syntax

#include <yaqr.h>

**int yrLog(char ****format* [,*arg*] ...)

int yrLog1M(int *errcode*, **char ****format* [,*arg*] ...)
int yrLog1(int *errcode* [,*arg*] ...)

int yrLog2M(int *errcode*, **int errcode2, char ****format* [,*arg*] ...)
int yrLog2(int *errcode* **int errcode2,** [,*arg*] ...)

int yrCode()
int yrCode2()
int yrString()

Description

The `yrLog1M` subroutine first forms an error message string using the printf–style *format* and all successive arguments. A pointer to the error string and *errcode* are then stored in errlog's internal variables, that can be accessed later via the `yCode` and `yrString` subroutines.

If an user error handler is set (see `yrSetHandler`), the `yrLog1M` subroutine calls the user handler. Upon return from the handler, the return value is checked. A 0 return value causes no additional actions to be taken, otherwise `yrLog1M` continues error processing as it is specified by the yrLog mode.

The `yrLog1` subroutine is similar to `yrLog1M`, except that it does not have the *format* parameter. The `format` string is selected from a message file, using *errcode* as a key. Message files are normally found in directories specified by the YAQRPATH environment variable. There may be message files in different national languages, this can be chosen by the `yrSetLanguage` subroutine. An application program can have its own message file(s), that can be specified by the `yrSetMsgfile`.

The `yrLog2` subroutines are similar to the `yrLog1` ones, except that they have the additinal *errcode2* argument. Like *errcode*, *errcode2* is also saved in an errlog's internal variable, that can later be accessed using the `yrCode2` subroutine. The *errcode2* is also sent to the errlog server. There is no predefined usage for *errcode2*, this is completely up to the user to decide what it stands for.

The `yrLog` subroutine is similar to `yrLog1M`, except that in does not have the *errcode* argument and the *errcode* value is always set to –1.

Return values

A 0 is returned if the user error handler have returned 0. Otherwise a -1 is returned.

See Also

yrCode(3l), yrCode2(3l), yrString(3l) yrSetHandler(3l), yrSetMode(3l)

yrSet

yrSetMode, yrSetHandler - set errlog mode and handler

Syntax

#include <yaqr.h>

int yrSetMode(int *mode*)

yrHandler yrSetHandler(int (**handler*)())

Description

The **yrSetMode** subroutines sets the errlog mode to the value *mode* and returns the old value of mode. The mode is formed by ORing of the following values:

ErrMODE_PRINT Print error message on stderr.

ErrMODE_EXIT Exit on receiving an error call.

ErrMODE_LOG Send error information to the error log server. This is default mode.

The **yrSetHandler** subroutine specifies the user error handler routine, that will be called in the case error occurs. It returns the address of the previously set user handler.

See Also

yrLog(3l)

Name

yhRegisterProtocol – register a protocol module entry point

Syntax

#include <yaqh.h>

yhProtocol_t *yhRegisterProtocol(int (*_entry_)(yhHeader_t*), int *protocol_id*, **int** *flags*, **char *_family_format_**, **char *_device_format_**, **char *_action_format_**)

Description

The **yhRegisterProtocol** subroutine is similar to the client's subroutine yqRegisterProtocol, additinally is has the *entry* parameter, a user protocol module entry point.

On receiving an equipment call to a device working under protocol *protocol_id*, the *entry* subroutine is called with one parameter, a structure of type yhHeader_t by address. The yhHeader_t structure is defined as follows:

```
typedef struct {
        yhDevice_t *device;
        yhFamily_t *family;
        yhAction_t *action;
        char    *arg;
        int     argsize;
        char    *res;
        int     ressize;
        int     resbufsize;
        int     mode;
} yhHeader_t;
```

where yhDevice_t, yhFamily_t and yhAction_t are:

```
typedef struct {
        char    *name;
} yhDevice_t;

typedef struct {
        char    *name;
} yhFamily_t;

typedef struct {
        char    *name;
        char    *ncvt_format;
} yhAction_t;
```

The *device*, *family*, *action* fields in the yhHeader_t structure are, in fact, pointers to structures, that may contain protocol specific fields, depending on the equipment access protocol. If so, the desirable data type can be set by casting. For example:

```
int my_protocol_module(yhHeader_t *h) {
        MyDevice_t *device = (MyDevice_t*) h->device;
        MyFamily_t *family = (MyFamily_t*) h->family;
        MyAction_t *action = (MyAction_t*) h->action;
```

Other fields in the yhHeader_t structure are: *arg* is a pointer to the input parameters of the action. The parameters are in network representation unless the yqPF_ARGHOST flags is set. The *argsize* is the size of *arg*.

At the time a user protocol module starts, the *res* points to a block of memory, result can be put to, the *resbufsize* is the size of this block. On exit, the protocol module should put the actial result size into *ressize*.

See Also

yqRegisterProtocol(3l), Ncvt(3l), yrCode(3l)

Name

yhWait - wait for an equipment call

Syntax

#include <yaqh.h>

yhWait(int *mode*, **int** *loc_size*)

Description

The **yhWait** subroutine listens for an equipment call from a client. Upon received an equipment call, the protocol identifier is decoded, and the corresponding user protocol module is called.

Arguments

mode specifies the level of verbosity for the device server, this is used mainly for debugging purpose. The *mode* is formed by ORing the following values:

yhTRACE_CALL Print a message like "Equipment call received".

yhTRACE_ACTION Print action.

yhTRACE_ERRORS. Print error messages.

loc_size is the size of a memory buffer to be allocated. This buffer is used to store an incoming equipment call message and the equipment call result.

See Also

yhRegisterProtocol(3l), yqCall(3l), yrCode(3l)

Name

Ncvt – subroutines to convert data to and from network representation

Syntax

#include <ncvt.h>

int NcvtGetArgNetworkSize(int ****size*, char ****format***, void ****hostptr***)**
int NcvtArgToNetwork(void ****netptr*, char ****format***, void ****hostptr***)**
int NcvtGetArgHostSize(int ****size*, char ****format***, void ****netptr***)**
int NcvtArgToHost(void ****hostptr*, char ****format***, void ****netptr***)**
int NcvtGetResNetworkSize(int ****size*, char ****format***, void ****hostptr***)**
int NcvtResToNetwork(void ****netptr*, char ****format***, void ****hostptr***)**
int NcvtResToHost(void ****hostptr*, char ****format***, void ****netptr***)**

int NcvtGetNetworkSize(int ****size*, char ****format***, void ****hostptr***)**
int NcvtToNetwork(void ****netptr*, char ****format***, void ****hostptr***)**
int NcvtGetHostSize(int ****size*, char ****format***, void ****netptr***)**
int NcvtToHost(void ****hostptr*, char ****format***, void ****netptr***)**

int NcvtFGetNetworkSize(int ****size*, char ****format***)**
int NcvtFGetHostSize(int ****size*, char ****format***)**

int NcvtVaToHost(void *dest, char *format, va_list va_alist);
int NcvtCall(void *func, void *alist, char *format);

Description

The *format* argument controls how each of these subroutines calculates sizes and converts. The *format* is a charater string, containing a conversion specification. A conversion specification is composed of a number of items possibly separated by blanks. Simple items are 'c', 's', 'i', 'l', 'f', 'd', that correspond to the char, short, int, long, float and double C data type, respectively. For example, the format string "ifc" corresponds to the structure:

```
struct {
        int i;
        float f;
        char c; }
```

A simple item can be preceded by 'u', that stands for 'unsigned'. For example, 'ui' means 'unsigned int'.

Arrays are specified by '[]', the *format* "i[10]" stands for an array of integers of size 10. Structures are specified by '{}'. For example, the structure:

```
struct {
        int i;
        struct {
                float f;
                char c; }}
```

is described by the format "{i{fc}}", or simply "i{fc}" (the outermost brackets may be omitted).

Pointers are specified by '*', "*i stands for a pointer to an integer. Pointers have no impact on network representation, they are not sent over network, network representation of 'i' and '*****i', for example, is the same.

Variable arrays are specified somewhat like "*(d[i])", that corresponds to a C structure:

```
struct {
        double *d;
        int i; }
```

Note that the '()' brackets are mandatory, the following example shows why:

```
int array[10];
struct {
        int array[10];
        int *array_of_pointers[10];
        int *pointer_to_array;
} example;
example.pointer_to_array = array;
/* The format is "i[10] *i[10] *(i[10])" */
```

A special kind of variable length array is a 0-terminated charater string, it is specified by 'z'.

Data to be converted can be of one of four following categories: input, output, input and output, neither input nor output. These categories are specified in the *format* string by the symbols '<', '>', '=' and '#', respectively. If nothing is specified, '=' is assumed. Input data are included into network representation by the `NcvtArg` subroutines, output data by the `NcvtRes` subroutines. An input/output specification remains valid until the next specification is found.

```
int arg1 = 10;
int arg2 = 20;
int res;
my_add(arg1, arg2, &res);
/* The format is "<ii >*i" */
```

The following diagram contains a formal syntax description of the conversion specification.

```
conversion_specification := one_item
                          | conversion_specification ' '
                          | conversion_specification one_item
              one_item := io_attribute const data_specification
```

```
          io_attribute := | '<' | '=' | '>' | '#'
     data_specification := complex_type | pointer_to_complex_type
pointer_to_complex_type := '*' '(' complex_type ')'
                         | '*' pointer_to_complex_type
          complex_type := non_pointer_type
                         | '*' complex_type
       non_pointer_type := simple_type
                         | simple_type '[' array_specification ']'
            simple_type := 'c' | 'f' | 'd' | 'z'
                         | signed_single_type
                         | 'u' signed_simple_type
                         | {conversion_specification}
     signed_simple_type := 's' | 'i' | 'l'
    array_specification := io_attribute array_data_specification
array_data_specification := array_simple_type
                         | const
                         | '*' array_data_specification
      array_simple_type := 's' | 'i' | 'l'
                 const := digit
                         | const digit
                 digit := 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

To make the description of the **Ncvt** subroutines more clear consider the following context: a client application wants a remote procedure to be executed on a server. The client sends input parameters to the server, the server returns output parameters to the client.

Before any data can be sent over network, they must be converted to network representation. The conversion usually requires an intermediate buffer data in network representation will be placed to. The **NcvtGetArgNetworkSize** subroutine returns the *size* of an intermediate buffer required to store network representation of input data pointed to by *hostptr* and described by the *format* string. The **NcvtArgToNetwork** subroutine converts input data pointed to by *hostptr* to network representation and places output in a buffer pointed to by *netptr*.

Upon receiving input data from the client, the server must convert them to host representation using the **NcvtGetArgHostSize** and **NcvtArgToHost** subroutines. The **NcvtGetArgHostSize** subroutine returns the *size* of an intermediate buffer required to store host representation of client's data pointed to by *netptr* and described by the *format* control string. The **NcvtArgToHost** subroutine converts input data pointed to by *netptr* to host representation and places output in a buffer pointed to by *hostptr*. The **NcvtArgToHost** re-creates the exact copy of client's data structure, including pointers and data that are not input ones.

Once input data are converted to host representation, the server can call a procedure requested by the client. On return from that procedure the server must send result to the client. To be sent the output data are converted to network representation using the **NcvtGetResNetworkSize** and **NcvtResToNetwork** subroutines. These subroutines are similar to the **NcvtGetArgNetworkSize** and **NcvtResToNetwork**, except that they process output data instead of input ones.

Having received output data from the server, the client must convert them back to host representation. The **NcvtResToHost** subroutine converts output data pointed to by *netptr* to host representation and places output into a data structure pointed to by *hostptr*. Our client have to specify the *hostptr* as a pointer to exactly the same data structure that was used in **NcvtArgToNetwork**.

The **NcvtGetNetworkSize**, **NcvtToNetwork**, **NcvtGetHostSize** and **NcvtToHost** are just other names for the **NcvtArgGetNetworkSize**, **NcvtArgToNetwork**, **NcvtArgGetHostSize** and **NcvtArgToHost** subroutines. Normally they should be used in a context when data are not subdivided into input/output categories, i.e. the *io_attribute* is not specified.

The **NcvtFGetNetworkSize** and **NcvtFGetHostSize** subroutines are similar to the ones without G, except that these work only for simple data structures, without pointers, when *size* can be calculated using the *format* only.

The **NcvtVaToHost** subroutine converts a C variable argument list, pointed to by *va_alist* to an ordinary structure and places the result in a buffer pointed to by *dest*.

The NcvtCall subroutine calls the subroutine specified by *func* with the parameters pointed to by *alist* and described by *format*.

Return values

Upon successful completion all these routines return 0. On failure a -1 is returned, and a more detailed information can be obtained via **yrCode**.

See Also

yrCode(3l), yrString(3l)