Reality v9.0

C API Reference Manual

NECSWS.COM

\Orchestrating a brighter world

Document control

Software Version	Document Status	Document Revision	Issue Date	Reason for Change
9.0	Published	0.1	April, 2002	Final draft

Prepared by

Name	Contact details
Pubali Pramanik	pubali.pramanik@necsws.com
Vijita Patel	vijita.patel@necsws.com

Table of Contents

Section 1: About this manual

This chapter gives a brief overview of the C programming interface, explains the purpose of this guide and the conventions used within it, and references other manuals which provide further information.

1.1 Overview

The Reality Interface described in this manual enables communication between UNIX and Reality or RealityX environments. To be more precise, it allows a C program to:

- Access files in a Reality or RealityX environment.
- Communicate with a DataBasic program running in a Reality or RealityX environment.

To achieve this, the C program must call the appropriate functions from the C function libraries provided:

- For UNIX or Microsoft Windows NT/2000 systems running Reality, the C function libraries are provided as an integral part of Reality.
- For other UNIX systems, the C function libraries are provided as part of the UNIX-Connect product.

UNIX-Connect is the generic name for a family of products which enable communication between NEC supported UNIX and Reality environments. For example, to enable communication between a UNIX system (without Reality) and a Reality 7.0 system, the UNIX-Connect product must be purchased and installed on the UNIX system.

Note
Remote UNIX and Reality environments must be connected via an IEEE 802.3 Local Area Network (Ethernet LAN) or via an X.25 Wide Area Network (X.25 WAN).

1.2 Purpose of this manual

This manual is intended for programmers who wish to do either of the following:

- Write C programs to access Reality or RealityX files.
- Write application programs in C which need to communicate with DataBasic programs in Reality or RealityX environments.

It is assumed that readers of this manual are familiar with the UNIX operating environment and the C programming language and that they have some knowledge of the Reality operating environment and the DataBasic programming language.

1.3 Contents of this manual

The remaining chapters of this guide are organised as follows:

- Chapter 2, Introduction to Reality Interfaces, gives a general overview of UNIX-Connect and Reality Networking, Interactive File Access and Interprocess Communication. It also explains how the Reality interfaces work. It is important that you read this chapter before attempting to use the Reality interfaces.

- Chapter 3, Reality Communications Interface, details the Reality Communications Interface (Rcc) functions.
- Chapter 4, Reality Filing Interface, details the Reality Filing Interface (Rfc) functions.
- Chapter 5, Reality General Services Interface, details the Reality General Services Interface (Rgc) functions.
- Chapter 6, Reality Indexed Access Interface, details the Reality Indexed Access Interface (Risc) functions.
- Chapter 7, Reality List Services Interface, details the Reality List Services Interface (Rlc) functions.
- Appendix A, Error Return Codes, lists the return codes referenced in the body of the manual and gives their meaning.
- Appendix B, Connecting to Multiple Databases, describes how to make connections to multiple Reality databases using the Rfc and Risc interfaces.
- Appendix C, Example Programs, contains example C programs demonstrating Interactive File Access and Inter-process Communication.

1.3.1 Comment sheet

A User Comment Sheet is provided for your comments on this manual.

If you have any comments at all, please let us know - it helps us to improve our documentation.

If your comment sheet has already been used, please write to the Technical Publications Manager at the address on the front cover, or email techpubs@NEC-is.com.

1.3.2 Abbreviations

A glossary of terms and abbreviations used in this manual is included at the end of the manual.

1.3.3 Conventions

This manual uses the following conventions:

Conventions	Description
`Text`	Bold text shown in this typeface is used to indicate input which must be typed at the terminal.
Text	Text shown in this typeface is used to show text output on the screen.
Bold text	Bold text in synopsis descriptions represents characters input exactly as shown. For example: **RccConnect**
text	Characters or words in italics indicate parameters which must be supplied by the programmer. For example, in, RccSend (*Shandle, Buffer, Length*) the arguments *Shandle*, *Buffer* and *Length* are italicized to indicate this is the general form for the **`RccSend`** routine. In the program you must supply specific arguments. Italic text is also used for titles of documents referred to by this

Conventions	Description
	document.
[brackets]	Brackets enclose optional parameters. For example, in, *accountname[,password]* the brackets *around ,password* indicate that this is an optional parameter which, when given, must be separated from *accountname* by a comma.
vertical ellipses...	Vertical ellipses are used in program examples to indicate that a portion of the program has been omitted.
0x*NN*	This denotes a hexadecimal value.

1.3.4 References

The following manuals contain further information:

- UNIX-Connect System Administration Guide
- Reality Reference Manual Volume 3: Administration
- UNIX-Connect User Guide
- DataBasic Reference Manual
- English Reference Manual

Section 2: Introduction to Reality interfaces

This chapter provides an overview of the Reality interfaces. It introduces Interactive File Access (IFA) and Inter-process Communication (IPC) and explains how they work. It is important that you read the information contained in this chapter before attempting to use the Reality interfaces.

2.1 Overview

The Reality IFA and IPC Interfaces enable a C program running in a UNIX or Windows environment to access standard Reality features. A program can:

- Access Reality files using Interactive File Access (IFA).
- Communicate with DataBasic programs running in a Reality environment using Inter-process Communication (IPC).

The implementation of the Reality interfaces is such that a C program can access Reality files or communicate with DataBasic programs using the same methods regardless of whether the Reality environment is local or remote.

2.2 Interactive key files

The Reality Interactive File Access (IFA) Interface enables an application running in the UNIX or Microsoft Windows environment to read from and write to Reality files, manipulate the data within them and use Reality list handling features. IFA comprises four interfaces:

- Reality General Services Interface (Rgc).
- Reality Filing Interface (Rfc).
- Reality List Services Interface (Rlc).
- Reality Index Sequential Services Interface (Risc). This provides a C-ISAM-like interface to Reality files, items and indexes. An application will normally use either the Rfc interface or the Risc interface.

When using these interfaces, an application can access the Reality environment directly or via a Reality server, depending on how the application is linked.

- To access the Reality environment directly, the application must be running on the same physical system (local) as the Reality environment and must be linked with the main Reality libraries.
- To access the Reality environment via a Reality server, the application must be running on UNIX and be linked with the UNIX-Connect library. This allows the application to access a Reality environment on the same physical system (local), or on a different physical system (remote). When using the client-server interface, the application may communicate with the following:
    - A local Reality environment running on UNIX.
    - A remote Reality environment running on UNIX or Windows NT/2000.
    - A remote Reality environment running on NEC proprietary Series 18/19 hardware (provided the client communications support OSI).

> ***Note***
> The Risc Interface is only available if the application is linked with the main Reality libraries.

The Reality IFA Interface API provides a consistent interface to the application, irrespective of the connection mechanism used to communicate with the Reality environment.

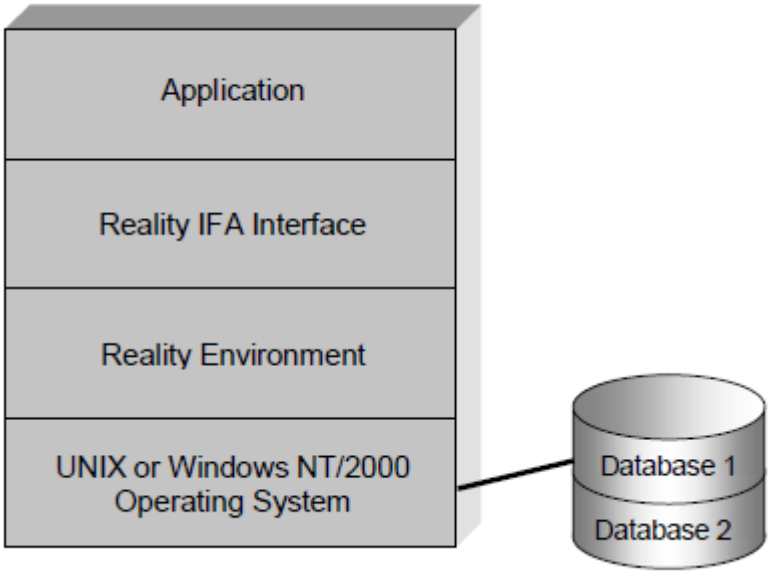


Figure 2-1. Application Using IFA To Access Reality Locally

Figure 2-1 shows a layered model that illustrates an application accessing Reality on a local machine. Communication between the application and Reality is provided by a collection of C API functions that collectively form the Reality IFA Interface. The API functions make calls to the Reality Services provided in the Reality Environment layer, which in turn access the databases via the underlying UNIX or Windows operating system. The Reality Interface and Reality Services are provided as part of Reality.

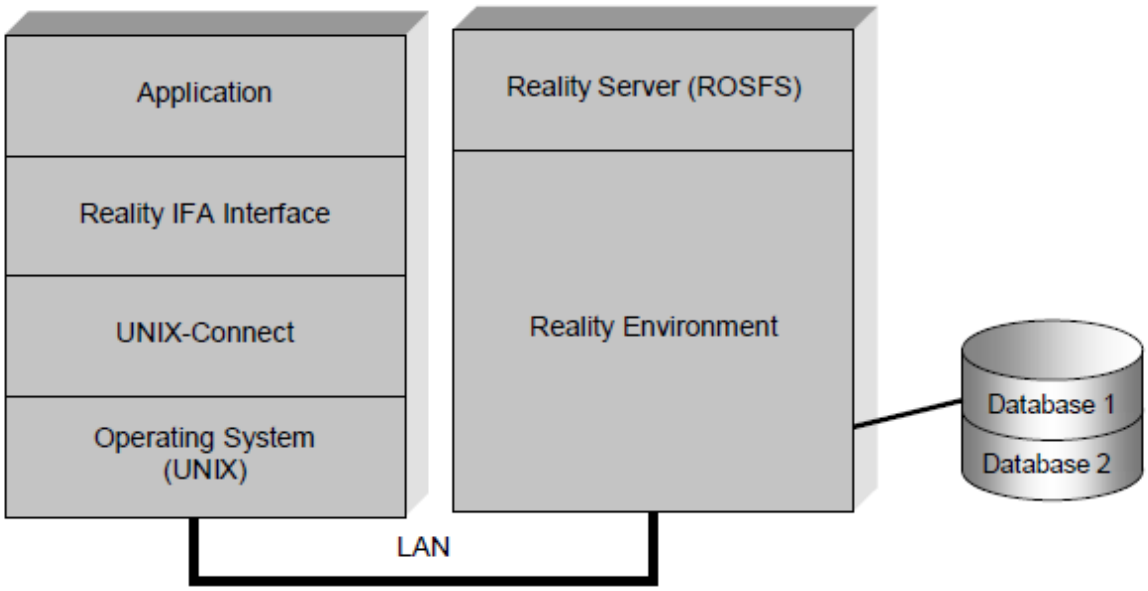


Figure 2-2. Application Using IFA to Access Reality Remotely

Figure 2-2 shows an application accessing a remote Reality environment. As with the local model, the application communicates directly with the Reality IFA Interface by using C API functions. In this case, however, the Reality IFA Interface makes calls to the UNIX–Connect Libraries. (UNIX-Connect is supplied on the Reality CD.) The operating system communicates with the Reality environment remotely via a LAN.

2.2.1 Reality general services

The Reality General Services (Rgc) Interface is a library of functions that allow a C program to do the following:

- Start up and shut down all services
- Manipulate items, attributes, values and subvalues
- Display error messages
- Obtain the time and date in Reality format

The Rgc functions are described in detail in Chapter 5.

2.2.1.1 Starting-up and shutting-down services

`RgcStartUpServices` is a macro that must be called by a C program that is going to use Rfc, Rgc, Rlc or Risc services.

Note

`RgcStartUpServices` initialises only those services that are used by the program; that is, for which header files have been included. See later in this chapter for details of header files.

The **`RgcShutDownServices`** function must be called to terminate those services initialized by **`RgcStartUpServices`**.

2.2.1.2 Data manipulation

The Rgc functions work on items which have been read into a buffer using the RfcRead function (RfcRead is part of the Rfc services). They allow the construction and manipulation of strings of data containing Reality entities – attributes, values and subvalues.

2.2.1.3 Error handling

Almost all IFA functions return an integer that is a numeric return code. In general, a return value of zero indicates success. Other values can be translated by calling the RgcErrMsg function to access error message text associated with a particular return code.

2.2.2 Reality filing services

The Reality Filing Services (Rfc) Interface is a library of functions that allow a C program to connect to a database and then create, delete, clear, read from and write to Reality files. The RgcStartUpServices macro (see above) must be called to initialise the Rfc services.

The Rfc functions are described in detail in Chapter 4.

2.2.2.1 Connecting to a database

The `RfcConnect` function connects to a specific account on a database. For a connection to a Series 18/19 system, the **database** is the remote system name.

2.2.2.2 Account handles

Once a connection to a database has been established, the account name can be saved to an account handle using the **`RfcGetAccount`** function. Having saved the account handle you can use **`RfcConnect`** to connect to a second database (or another account on the same database) and, subsequently, return to the first by simply referencing the account handle (using **`RfcSetAccount`**).

Note
Account handles only need to be saved for connections to multiple databases.

General rules for connecting to multiple databases are described in Appendix B.

2.2.2.3 File handles

To open a file, the **`RfcOpenFile`** function must be called. The RfcOpenFile function is passed a file name and returns a file handle. This file handle is then used by all functions that perform operations on open files.

2.2.2.4 File names

The file name parameter (used by **`RfcOpenFile`** and other functions) can take one of three forms:

File names	Description
'filename'	Specifies the default data section.
'filename,dataname'	Specifies a particular data section.
'DICT *filename*'	Specifies the dictionary section.

2.2.3 Reality list services

The Reality List Services (Rlc) Interface is a library of functions that allow a C program to use Reality list handling features. The **`RgcStartUpServices`** macro (see page 2-5) must be called to initialise the Rlc services.

Reality lists are lists of item-ids created by list-generating English verbs. A list can be saved in a file item – this can be in POINTER-FILE or another specified file. Alternatively, a list can be dynamically created from the item-ids of an open file. For further details on lists, see *English Reference Manual*.

The Rlc Interface allows C programs to manipulate lists in the Reality environment. Functions are provided to create lists, save and retrieve the created lists to/from files, and use the lists to access data from a specified file.

The Rlc functions are described in detail in Chapter 7.

2.2.3.1 List handles

A list can be created from the item-ids of an open file with the **`RlcMakeList`** function. This returns a list handle. This list handle is used by all functions that perform operations on lists.

2.2.4 Reality index sequential services

The Reality Index Sequential Services (Risc) Interface is a library of functions that allow a C program to use an alternative interface to Reality files and indexes. This interface is more in the style of C-ISAM, and will therefore be easier to use for programmers and applications accustomed to C-ISAM and similar products.

This interface is not a direct replacement for C-ISAM – the intention is to simplify the task of extending or converting existing programs which already use C-ISAM, to be able to use Reality files and indexes. This interface may also prove more appealing to experienced C programmers writing new applications to interface directly with a Reality database.

2.2.4.1 The C-ISAM view of Reality indexed files

The main aim of this interface is to hide the special nature of the Reality item-id from the programmer. It works with records and keys and introduces the concept of a current record.

A record consists of the Reality item-id and the item data, separated by an Attribute Mark (0xFE). The item data consists of several variable length fields separated by Attribute Marks. The Reality item-id appears as the first field in each record.

Although this interface makes the Reality item-id appear as part of the record data, it still has special significance to the underlying Reality File System. It is still the identifier of the record and as such must have a different value in every record (to use relational database terminology, the item-id is always the primary key). A Reality file cannot contain two different records with the same value in the first field.

A key is a Reality Key Value. In the simplest case where the file is indexed on a single field with no special conversions, the key is just the appropriate field value. In an Index defined on several fields (again with no special conversions) the key comprises the appropriate field values separated by Attribute Marks.

With complex Indexes including English conversions, the relationship between the record and the key value is less obvious.

2.2.5 Using IFA functions

IFA provides a large number of file access functions enabling a C program to perform a wide variety of operations on a Reality file. However, it can also be very simple to use.

For example, to alter the contents of an attribute, a C program calls the following functions:

IFA functions	Description
RgcStartUpServices	This is to initialise the interactive file access functions.
RfcConnect	This is to connect to the Reality database.
RfcOpen	This is to open the Reality file.
RfcRead	This is to read the item.

IFA functions	Description
RgcSetAttr	This is to overwrite the attribute.
RfcWrite	This is to write the item to the file.
RfcClose	This is to close the file.
RfcDisconnect	This is to disconnect from the Reality database.
RgcShutDownServices	This is to close the interactive file access functions.

2.2.6 Type definitions

A number of type definitions are provided for use with the IFA functions (see below). The way in which the various type definitions are used is described under the appropriate function descriptions.

Type definitions are provided in the following include files, which must be #included in every program which is to use Rfc, Rgc, Rlc or Risc as follows:

```
#include <ros/rfc.h> /* for Rfc services */
#include <ros/rlc.h> /* for Rlc services */
#include <ros/rgc.h> /* for Rgc services */
#include <ros/risc.h> /* for Risc services */
```

On Windows systems, to allow these files to be included as shown above, the following should be added to the complier's include path:

```
%REALROOT%\include
```

Note

You need only include **rfc.h**, **rlc.h** and **risc.h** if the corresponding services (Rlc, Rfc or Risc) are being used. You must, however, always include **rgc.h**.

2.2.7 Compiling and linking your program

A number of type definitions are provided for use with the IFA functions (see below). The way in which the various type definitions are used is described under the appropriate function descriptions.

2.2.7.1 UNIX

When you compile and link a program that uses IFA, the requisite libraries must be specified. The Reality (local) and UNIX-Connect (client-server) implementations of Interactive File Access use different libraries, though the functions are identical and a program written to use one implementation can be linked to use the other.

- A program using the Reality implementation must be linked to realc.a, reals.a and the curses library.
- A program using the UNIX-Connect implementation must be linked to the IFA library.
- All programs must be compiled and linked to use the following libraries: Reality Communications Services (Rcs), X.25 (regardless of whether the system has an X.25 connection or not), sockets and the transport layer interface (xti on AIX; nsl otherwise).

So, for example, a program called **ifa_eg.c** would be compiled and linked on a UNIX machine (except AIX) as follows:

For Reality IFA:

```
cc ifa_eg.c $REALROOT/lib/realc.a $REALROOT/lib/reals.a -lrcs
$REALROOT/lib/reals.a -lsx25 -lsocket -lnsl -lcurses
```

For UNIX-Connect IFA:

```
cc ifa_eg.c -lifa -lrcs -lsx25 -lsocket -lnsl
```

On AIX, replace the `-lnsl` parameter with `-lxti`.

On site linking:

On a UNIX system, to avoid having to re-compile application programs each time a new version of UNIX-Connect or Reality is released, programs should be compiled and linked separately.

So, for UNIX-Connect IFA, a program called **ifa_eg.c** would be compiled as follows:

```
cc -c ifa_eg.c
```

This generates the file ifa_eg.o, which must then be linked as follows:

```
cc ifa_eg.o -lifa -lrcs -lsx25 -lsocket -lnsl
```

On AIX, replace the `-lnsl` parameter with `-lxti`.

This means that if a new version of UNIX-Connect or Reality is released there is no need to re-compile the program ifa_eg.c although it must be re-linked. This is known as **onsite linking**.

2.2.7.2 Windows

On a Windows platform, you will probably develop applications in an Integrated Development Environment (IDE) such as Microsoft's Visual Studio. The IDE must be set up to include the relevant header files and library files.

For accessing databases locally, the relevant file locations are:

%REALROOT%\lib\realc.dll
%REALROOT%\lib\realc.lib
%REALROOT%\include\ros\rlc.h
%REALROOT%\include\ros\rfc.h
%REALROOT%\include\ros\rgc.h
%REALROOT%\include\ros\risc.h

2.3 Inter-Process Communication (IPC)

The Reality IPC Interface enables an application running in the UNIX environment to access the Reality environment via a Reality server, using low-level communications function calls. A program can connect to another program, send and receive data, and disconnect from the program using the NEC Distributed Data Access (DDA) protocol.

When using IPC, the application must always use the client-server interface, whether the Reality environment is local or remote.

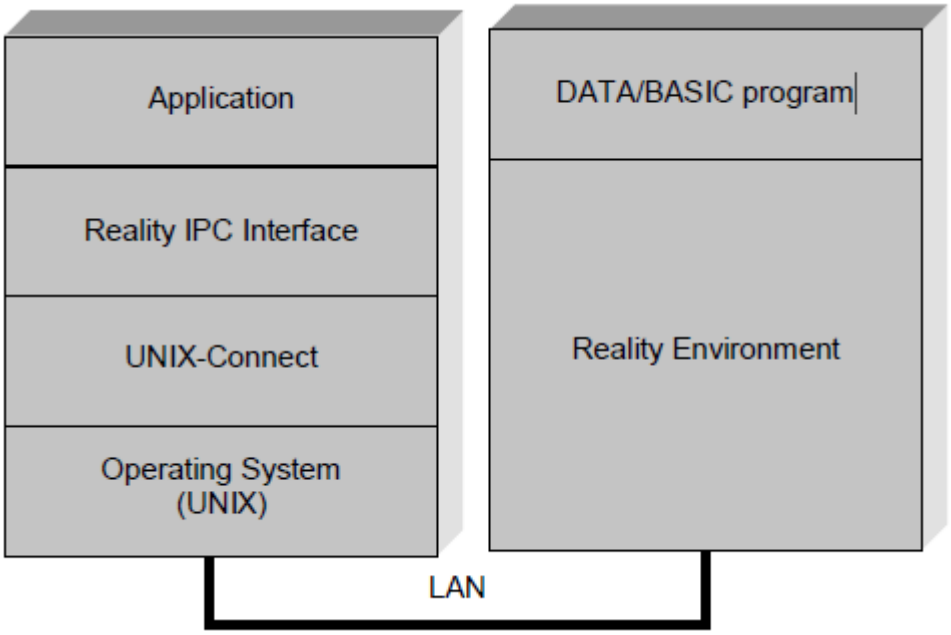


Figure 2-3. Application Using IPC To Access Reality Remotely

Figure 2-3 shows an application accessing a Reality environment remotely. The application communicates directly with the Reality IPC Interface by using C API functions. The Reality IPC Interface makes calls to the UNIX–Connect Libraries. (UNIXConnect is available from NEC as a separate product.) The operating system communicates with the Reality environment remotely via a LAN.

2.3.1 Function libraries

The Reality IPC interface is provided by means of the Rcc library. This is provided as part of the UNIX-Connect Reality Communications Service (Rcs). The relevant library must be declared when the program is linked.

Function libraries	Definition
/usr/lib/librcs.a	Interprocess Communications library
/usr/lib/libsocket.a	Socket Interface library
/usr/lib/libnsl.a	Transport Layer Interface library on SV/88 systems
/usr/lib/libxti.a	Transport Layer Interface library on AIX systems
/usr/lib/libsx25.a	X.25 Interface library

2.3.2 DDA

Distributed Data Access (DDA) is the NEC proprietary protocol for the exchange of messages between inter-connected systems. A DDA message consists of the following fields:

- A function code (2 bytes) – The meanings of function codes sent between user written programs are defined by the programs themselves.

> ***Note***
> Function codes greater than 0x3FFF are reserved for internal use and should not be used by user-written programs.

- A reference number (1 byte) – The meanings of reference numbers sent between user-written programs are defined by the programs themselves.
- Qualifier data (up to 255 bytes) – The use of qualifier data is defined by the communicating programs.
- Qualifier length (1 byte) – Specifies the length of the qualifier data.
- Data – This is the main body of information sent to the other program.
- Data length (4 bytes) – Specifies the length of the data.

A DDA message is constructed my means of a Message Control Block (MCB). This is a structure containing fields for each of the elements listed above. The MCB structure is described in detail in Chapter 3.

> ***Note***
> The function code, reference number and qualifier are all optional. If you are not using these fields, you can use **Simple** functions (see below) that transfer data without using an MCB.

2.3.3 Data transfer functions

The Rcc library provides two types of data transfer function:

- Message functions
  - RccSendMsg
  - RccReceiveMsg
  - RccRecWaitMsg

  These allow a program to send and receive complete DDA messages using a Message Control Block.

- Simple functions
  - RccSend
  - RccReceive
  - RccRecWait

  These allow the user to transfer data without having to explicitly set up an MCB.

> ***Note***
> Although, when you use RccSend, you do not provide values for the function code, reference number and qualifier, the data is transferred in DDA format. The corresponding receive functions (RccReceive and RccRecWait) discard any function code, reference number and qualifier included in a DDA message.

2.3.4 Clients and servers

A program may be either a client (initiates a connection) or a server (responds to a client program).

Typically, a client application starts up a server and sends a command message, the server actions the command and returns a response message. So, a typical client program will execute the following commands:

Connect
    Send
    Receive
Disconnect

and a typical server program will execute the following commands:

Accept
    Receive
    Send
Disconnect

The send/receive sequences may loop as many times as necessary until a disconnect or a timeout occurs.

Client and server programs can be written in C, to run in the UNIX environment, or in DataBasic, to run in the Reality environment. Normally the client-server pair will comprise one C and one DataBasic program but the Rcc functions can be used to enable C programs to communicate as a client-server pair. In C programs, a client program calls the **`RccConnect`** function to initiate a connection, and a server calls the **`RccAccept`** function to respond to a client program.

2.3.5 Session references

Once a connection is established it is accessed by means of a **session reference**. A session reference is simply a number used to indicate to underlying software (which handles all program-to-program connections) which connection the program is accessing.

A client program passes a pointer to a session reference variable to **`RccConnect`**. The session reference is returned by **`RccConnect`** and must be used in all subsequent function calls that apply to the same connection.

Similarly, a server program written in C passes a pointer to a session reference variable to **`RccAccept`**. The session reference is returned by **`RccAccept`** and must be used in all subsequent function calls that apply to the same connection.

2.3.6 Using Rcc

The Rcc functions are held in the Reality Communications Library (/usr/lib/librcs.a).

2.3.6.1 Type definitions

In addition to the functions themselves a number of type definitions are provided for use when calling the functions (for details refer to Chapter 3). Programs that use Rcc should #include the appropriate header file as follows:

```
#include <ros/rcc.h>
```

To allow this file to be included as shown above, the following should be added to the complier's include path:

```
/usr/include
```

2.3.7 Compiling and linking your program

A program that uses Rcc functions must be compiled to use the Reality Communications Services (Rcs) library. All programs written to use IPC must be compiled and linked to use the transport layer interface library (xti on AIX; nsl otherwise) and the socket library.

You must also use the X.25 library if one is available (regardless of whether the system has an X.25 connection or not).

For example, a program called client.c might be compiled as follows:

```
cc client.c -lrcs -lsx25 -lsocket -lnsl
```

Note

1. It is important that the libraries are linked in the order shown above.
2. If no X.25 library is available, omit the **–lsx25** parameter.
3. On AIX, replace the **-lnsl** parameter with **-lxti**.
4. The exact libraries used when linking may vary according to the type of system. Your NEC support representative will be able to tell you which libraries are required on your system.

2.3.6.2 On-site linking

To avoid having to re-compile application programs each time a new version of UNIX-Connect is released, programs should be compiled and linked separately. So, a program called client.c would be compiled as follows:

```
cc -c client.c
```

This generates the file client.o, which might then be linked as follows:

```
cc client.o -lrcs -lsx25 -lsocket -lnsl
```

This means that if a new version of UNIX-Connect is released there is no need to recompile the program client.c, though it must be re-linked.

2.4 Error handling and return codes

Most Reality Interface functions return an integer, which is a numeric return code. This return code will have a value of zero if the function call is successful. If the function call is unsuccessful, the return code will have a non-zero value. A complete list of return codes and their meanings is given in Appendix A.

Return code definitions are #defined in the following header files:

- ros/rfe.h
- ros/rge.h
- ros/rle.h
- ros/risc.h
- ros/rce.h

These can be included as needed in user-written C programs that use the Reality Interface functions. To simplify the inclusion of these in your program, add one of the following to your compiler's include path:

System	Path
UNIX system with Reality	**$REALROOT/include**
Windows NT/2000 system with Reality	**$REALROOT/include**
UNIX system without Reality	**/usr/include**

2.4.1 Interactive file access

Textual messages associated with Interactive File Access and Interprocess Communication return codes can be displayed using the **RgcErrMsg** function.

The **RgcErrMsg** function is passed a return code, which it uses as an index to the error message file, and a pointer to a buffer. **RgcErrMsg** extracts the textual error message and places it in the buffer.

2.4.1.1 Example

In the example below, the if clause is executed if **RetCode** does not equal **SUCCESS**. In these circumstances, **RgcErrMsg** is called to read the associated error message into the supplied buffer, **ErrorString**. The **printf** statement displays the contents of the buffer.

```
if ((RetCode = RfcOpenFile(FileName,&FileHandle)) != SUCCESS) {
       ErrorString = RgcErrMsg(RetCode);
       (void) printf("%s\n", ErrorString);
       exit(2);
}
```

2.4.2 InterProcess communication

Textual messages associated with InterProcess Communication function return codes can be displayed using the **RccError** function (if you are using Interactive File Access as well, however, you must use **RgcErrMsg**).

The **RccError** function is passed a return code, which it uses as an index to the error message file, and a pointer to a buffer. **RccError** extracts the textual error message and places it in the buffer.

2.4.2.1 Example

In the example below, the if clause is executed if **RetCode** is not equal to SUCCESS. In these circumstances **RccError** is called to read the associated error message into the supplied buffer, **ErrorStr**. The **printf** statement displays the contents of the buffer.

```
if ((RetCode = RccSendMsg(Reference,&Msg)) != SUCCESS) {
       RccError(RetCode, ErrorStr);
       printf("RccSendMsg Error : %s\n", ErrorStr);
       exit(1);
}
```

Section 3: Reality Communications Interface functions

The Reality Communications Interface (Rcc) functions enable a C program in a UNIX environment to communicate with a DataBasic program in a Reality environment or another C program in a UNIX environment.

3.1 Rcc functions

The Reality Communications Interface for the C Language allows C programs running in a UNIX environment to communicate with DataBasic programs running in a Reality environment. In fact, Rcc is a library of C functions which allows a C program to connect to another program, send and receive data and disconnect from the program using the NEC Distributed Data Access (DDA) protocol.

A program may be either a client (initiates a connection) or a server (responds to a client program). Typically, a client application starts up a server and sends a command message, the server actions the command and returns a response message.

For further details of Interprocess Communication and how it works, refer to Chapter 2.

Rcc functions	Definition
RccConnect	Sets up a connection.
RccSetConnect Options	Allows the default connection settings to be altered.
RccAccept	Accepts an incoming connection.
RccSetAcceptO ptions	Allows the default acceptance settings to be altered.
RccSend	Sends a buffer of data.
RccSendMsg	Sends a formatted DDA message.
RccReceive	Receives a buffer of data (returns immediately).
RccReceiveMsg	Receives a formatted DDA message (returns immediately).
RccRecWait	Receives a buffer of data (waits for data).
RccRecWaitMsg	Receives a formatted DDA message (waits for data).
RccDisconnect	Terminates the connection.
RccError	Reads an error message.

3.1.1 Message Control Block

The Rcc message mode functions (**`RccSendMsg`**, **`RccReceiveMsg`** and **`RccRecWaitMsg`**) must be given a pointer to a DDA Message Control Block (MCB). This is a structure of type **`RCS_MCB`**:

```
typedef struct mcb {
RCS_FUNCTION Function;
RCS_REF Reference;
```

```
int QualLength;
int DataLength;
unsigned char * QualBuffer;
unsigned char * DataBuffer;
int MaxQualLength;
int MaxDataLength;
} RCS_MCB;
```

3.1.1.1 RccSendMessage

When calling **`RccSendMsg`** you must set the elements of the MCB to the following:

MCB elements	Definition
Mcb.Function	The DDA function code.
Mcb.Reference	The DDA reference number.
Mcb.QualBuffer	A pointer to a buffer containing the DDA qualifier. The length of the qualifier must not exceed 255 bytes.
Mcb.QualLength	The length of the DDA qualifier.
Mcb.DataBuffer	A pointer to a buffer containing the DDA data.
Mcb.DataLength	The length of the DDA data.
Mcb.MaxQualLength	Unused
Mcb.MaxQualLength	Unused

3.1.1.2 RccReceiveMsg and RccRecWaitMsg

When calling **`RccReceiveMsg`** or **`RccRecWaitMsg`** you must set the following elements of the MCB:

MCB elements	Definition
Mcb.QualBuffer	A pointer to a buffer in which to return the DDA qualifier. The length of this buffer must not exceed 255 bytes.
Mcb.DataBuffer	A pointer to a buffer in which to return the DDA data.
Mcb.MaxQualLength	The size of the qualifier buffer.
Mcb.MaxDataLength	The size of the data buffer.
On return, the elements of the MCB will be set to the following:	
Mcb.Function	The DDA function code.
Mcb.Reference	The DDA reference number.
Mcb.QualLength	The number of bytes received in Mcb.QualBuffer.
Mcb.DataLength	The number of bytes received in Mcb.DataBuffer.
Mcb.QualBuffer	The pointer to the qualifier buffer. The buffer will be filled with the DDA qualifier data.
Mcb.DataBuffer	The pointer to the DDA data buffer. The buffer will be filled with the DDA data.
Mcb.MaxQualLength	Unchanged

MCB elements	Definition
Mcb.MaxDataLength	Normally unchanged but see note below. ***Note*** If the length of the DDA data exceeds that of the data buffer, the function will return the error **RCE_MOREDATA** or **RCE_QUALTRUNC_MOREDATA**. Under these circumstances, the **MaxDataLength** element will be set to the total length of the data sent. To receive the remaining data, save the data received by the first call, and then call **RccReceiveMsg** or **RccRecWaitMsg** (as appropriate) again with the same MCB, repeating as necessary until you have received all the data.

3.2 RccAccept

Purpose	The **RccAccept** function is called by a server program to accept an incoming connection from a client program. The function will wait until connection is established or the connection timeout (see **RccSetAcceptOptions**) has expired. ***Note*** **RccAccept** is only available on UNIX systems.	
Synopsis	int RccAccept(*PtrShandle*, *Account*, *Server*, *ClientId*, *Plid*) **RCS_PSREF** *PtrShandle*; **char** * *Account*; **char** * *Server*; **char** * *ClientId*; **char*** *Plid*;	
Parameters	*PtrShandle*	A pointer to a variable in which the session reference will be returned. The session reference uniquely identifies the connection established and must be used with all subsequent function calls that make use of this connection.
	Account	A pointer to a string that can contain the account name if required. This must match the account name specified by the client program. In most cases, this can be set to a null string or a null pointer.
	Server	A pointer to a string containing the server name. This must match the server name specified by the client program.
	ClientId	A pointer to a buffer (at least 51 bytes in length) in which the client's identification (system-name*user-id) will be returned.

	Plid	A pointer to a buffer (at least 51 bytes in length) in which the client's PLId will be returned.
Return value	The `RccAccept` function returns `SUCCESS` for successful completion, or one of the return codes listed in Appendix A. The following are likely errors:	
	RCE_INSUFFMEM	System error: insufficient memory.
	RCE_PLID_LENGTH	The PLId is too long for supplied buffer. The PLId has been truncated to 50 characters, but otherwise the function completed successfully.
	RCE_SERVER	Invalid server name.
	RCE_SND_IPC_MSG	Send IPC message failure.
	RCE_TIMEOUT	Operation timed out.
	RCE_USERID	Invalid user-id/password.
Remarks	Client and server matching	The account and server names specified in the call to `RccAccept` are matched against the account and server names specified by the client program. Similarly, the user-id specified by the client program is matched against the UNIX user-id from which the server program that calls the `RccAccept` function is being run. Unless the client program specifies otherwise, when a client program requests a connection, if the required server program is not already running, it will be started automatically by the session manager. A server program started by the Session Manager runs under the UNIX user-id specified by the client in the CONNECT statement or `RccConnect` call; and **stdin**, **stdout** and **stderr** are directed to **/dev/null**. If the client program has specified that the server program should not be started automatically and no matching server is already running, the connection request will be queued until either a matching server is started or the connection timeout expires. Before starting, a server program the session manager executes **/etc/rcsprofile** if it exists. If it does not exist, **/etc/profile** is executed. If **$HOME/.rcsprofile** exists it is also executed, after **/etc/rcsprofile** or **/etc/profile**.
	Server environment	The environment variables set up by the session manager are **HOME**, **PATH**, **SHELL** and **MAIL**. **HOME** and **SHELL** are set up according to the

		UNIX password file entry. **PATH** is set to **$HOME/bin:/usr/bin** and **MAIL** is set to **/usr/mail/***UserId*.
	Session reference	The value returned in the *PtrShandle* parameter is a unique session reference number that is used to identify subsequent transfers over the same connection. The client's identification (system-name*user-id) and physical location identifier (PLId) are also returned (in the *ClientId* and *Plid* parameters respectively) – these can be used for further security checking.
Example		

```
#include <ros/rcc.h>
.
.
.
main() {
        char Server[] = "abc"; /* The name of the server */
        char Account[] = "xyz"; /* The name of the account */
        char ClientId[51]; /* Buffer to receive client id */
        char Plid[51]; /* Buffer to receive PLId */
        RCS_SREF Shandle; /* To hold session reference */
        RCS_PSREF PtrShandle; /* Pointer to session reference */
        int RetCode; /* To hold returned value */
        char ErrorStr[80]; /* Buffer to receive error description */
        PtrShandle = &Shandle; /* Point to the session reference */
        .
        .
        .
        /* Tell the user what's going on */
        printf("Accepting...\n");
        /* Wait for an incoming connection.
                If an error occurred ... */
if ((RetCode = RccAccept(PtrShandle, Account, Server,
                        ClientId, Plid)) != SUCCESS) {
                /* Get the error description */
                RccError(RetCode, ErrorStr);
                /* Display it */
                printf("RccAccept Error :%s\n", ErrorStr);
                exit(); /* quit */
        }
        .
        .
        .
}
```

In this example a server program accepts a connection from a client specifying account name 'xyz' and server name 'abc'. Shandle is used to store the session reference, *ClientId* the client-id and *Plid* the PLId.

See also **RccSetAcceptOptions** for details of setting a timeout.

3.3 RccConnect

Purpose	The **RccConnect** function is called by a client program to initiate a connection to a server.
Synopsis	int **RccConnect**(*PtrShandle*, *System*, *Userid*, *Account*, *Server*)

	RCS_PSREF *PtrShandle*; **char *** *System*; **char *** *Userid*; **char *** *Account*; **char** * *Server;*	
Parameters	*PtrShandle*	A pointer to a variable in which the session reference will be returned. The session reference uniquely identifies the connection established and must be used with all subsequent function calls that make use of this connection.
	System	A pointer to a string which identifies the environment to which a connection is required; that is, the name of an entry in **/etc/ROUTE-FILE**. A null string specifies the local environment.
	User	A pointer to a string containing the user-id or the user-id and password, in the form: *UserId*[,*Password*] If *Userid* is null, the USERS-FILE entry for *System* under the local user- or group-id is used.
	Account	A pointer to a string containing the account name or the account name and password, in the form: *Account* [,*Password*] The *Account* parameter identifies the Reality account that holds the server program. If the account points to a null string or is a null pointer, it will match with any account specified by the server.
	Server	A pointer to a string that identifies the server program on the remote system.
Return value	The **RccConnect** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors:	
	RCE_PLID	Invalid Physical Location Identifier.
	RCE_SERVER	Invalid server name.
	RCE_SND_IPC_MSG	Send IPC message failure.
	RCE_SYSTEM	Invalid system name.
	RCE_THOSTDISC	Transport: circuit disconnected.
	RCE_TIMEOUT	Operation timed out.
	RCE_USERID	Invalid user-id/password.

Example	`#include <ros/rcc.h>`
	`.`
	`.`
	`.`
	`main()`
	`{`
	`  char[] System = "MDIS"; /* Name of system to connect to */`	

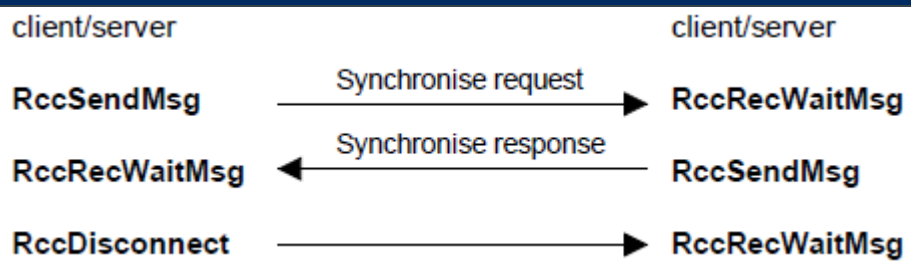
```
        char[] Userid = "ROSEMARY"; /* User ID */
        char[] Account = "PROGS"; /* Account to connect to */
        char[] Server = "PROGA"; /* Name of server program */
        RCS_SREF Shandle; /* To hold session reference */
        RCS_PSREF PtrShandle; /* Pointer to session reference */
        int RetCode; /* To hold returned value */
        char ErrorStr[80]; /* Buffer to receive error description */
        PtrShandle = &Shandle; /* Point to the session reference */
        .
        .
        .
        /* Tell the user what's going on */
        printf ("Connecting ......\n");
        /* Try to connect. If an error occurred ... */
        if ((RetCode = RccConnect(PtrShandle, System, Userid, Account,
              Server)) != SUCCESS) {
              /* Get the error description */
              RccError(RetCode, ErrorStr);
              /* Display it */
              printf("RccConnect Error: %s\n", ErrorStr);
        }
        .
        .
        .
}
```

In the above example the client program makes a connection to Northgate with a user-id of ROSEMARY and starts up a server program called PROGA in the account PROGS. The session reference is placed in *Shandle*.

See also **RccSetConnectOptions**, **RccDisconnect**.

3.4 RccDisconnect

Purpose	The **RccDisconnect** function terminates a connection established by **RccAccept** or **RccConnect**.	
Synopsis	int **RccDisconnect**(*Shandle*) **RCS_SREF** *Shandle*;	
Parameters	*Shandle*	The session reference of the required connection, returned by **RccAccept** or **RccConnect**.
Return value	The **RccDisconnect** function returns SUCCESS for successful completion, or one of the return codes listed in Appendix A. The following are likely errors:	
	RCE_ILLSREF	Illegal session reference.
	RCE_SND_IPC_MSG	Send IPC message failure.
Remarks	To minimise the risk of losing data and achieve an orderly disconnect it is recommended that the procedure below is followed:	

	 If **RccRecWaitMsg** or **RccSendMsg** returns the error code **RCE_THOSTDISC**, the receiving program should also issue an **RccDisconnect** to clean up the connection resources. If this is not done, and the process terminates, a **process has died** message appears in the session log. Alternatively, if you know that the underlying network will be TCP/IP, you can set the environment variable **UC_USE_ORDERLY_REL to 1** – this will ensure that TCP orderly release is used and guarantee that all data is sent before disconnection.

Example	`#include <ros/rcc.h>`
	`RCS_SREF Shandle; /* Holds the session reference */`
	`int RetCode; /* To hold returned value */`
	`char ErrorStr[80]; /* Buffer to receive error description */`
	`.`
	`.`
	`.`
	`/* Tell the user what's going on */`
	`printf("Disconnecting...\n");`
	`/* Try to disconnect. If an error occurred ... */`
	`if ((RetCode = RccDisconnect(Shandle)) != SUCCESS) {`
	`  /* Get the error description */`
	`  RccError(RetCode, ErrorStr);`
	`  /* Display it */`
	`  printf("RccConnect Error: %s\n", ErrorStr);`
	`  exit(); /* quit */`
	`}`

See also **RccAccept**, **RccConnect**.

3.5 RccError

Purpose	The **RccError** function returns the description associated with a specified error number (return code). Each Rcc function returns **SUCCESS** for successful completion – any other value indicates an error. For each error code, there is a description of the error.	
Synopsis	int **RccError**(*ErrorNumber*, *Message*) **int** *ErrorNumber*; **char *** *Message*;	
Parameters	*ErrorNumber*	The value returned by an Rcc function.

	Message	A pointer to a buffer (at least 80 bytes in length) in which **RccError** will return the error description.
Return value	The **RccError** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors:	
	RCE_ERRNUM_READ	Cannot read error number from ERRMSG-FILE.
	RCE_ERRMSG_LOCATE	Cannot locate error message in ERRMSG-FILE.
	RCE_ERRMSG_READ	Cannot read error message from ERRMSG-FILE.
Remarks	Additional diagnostic information can be obtained from the global integers **TliReason**, **t_errno** and **errno**, where:	
	TliReason	It is the reason code for the most recent disconnect received through the TLI (Transport Layer Interface). **TliReason** is set to -1 if no disconnect has been received.
	t_errno	It is a **TLI** error number.
	errno	It is a standard UNIX error number.
	TliReason is declared in the file `rcc.h`, and **t_errno** and **errno** in `errno.h` and tiuser.h. If these files are #included in your program, the variables need not be explicitly declared. These error numbers are not always relevant but may be useful if problems are being caused by underlying transport errors – contact your NEC support representative for details.	

Example	<pre>#include <ros/rcc.h>
	int RetCode; /* To hold returned value */
	/* Global error variables */
	extern int TliReason;
	extern int t_errno;
	extern int errno;
	char ErrorStr[80]; /* Buffer to receive error description */
	.
	.
	.
	/* Try to do something. If an error occurred ... */
	if ((RetCode = RccSend(Shandle, Buffer, Length)) != SUCCESS) {
	  /* Get the error description */
	  RccError(RetCode, ErrorStr);
	  /* Display the error details */
	  printf("RccSend Error:%s\n TliReason:%d t_errno:%d
	  errno:%d\n", ErrorStr, TliReason, t_errno, errno);
	}</pre>
	In the above example, if the **RccSend** function call fails, **RetCode** is used to access the associated error description. In addition, the settings of **TliReason**, **t_errno** and **errno** are displayed.	

3.6 RccReceive

Purpose	The **RccReceive** function receives data from a remote environment. If no data is available, the function returns immediately.

<table>
<tr><td></td><td colspan="2"></td></tr>
<tr><td>Synopsis</td><td colspan="2">int RccReceive(Shandle, Buffer, BufferLength, RcvLength)

RCS_SREF Shandle;
unsigned char * Buffer;
int BufferLength;
int * RcvLength;</td></tr>
<tr><td rowspan="4">Parameters</td><td>Shandle</td><td>The session reference of the required connection, returned by RccAccept or RccConnect.</td></tr>
<tr><td>Buffer</td><td>A pointer to a buffer in which RccReceive will return the received data.</td></tr>
<tr><td>BufferLength</td><td>The size of the receive buffer in bytes.</td></tr>
<tr><td>RcvLength</td><td>A pointer to a variable in which the length of the data returned in Buffer will be returned.</td></tr>
<tr><td rowspan="6">Return value</td><td colspan="2">The RccReceive function returns SUCCESS for successful completion, or one of the return codes listed in Appendix A. The following are likely errors:</td></tr>
<tr><td>RCE_ILLSREF</td><td>Illegal session reference.</td></tr>
<tr><td>RCE_MOREDATA</td><td>The data received is longer than the size of the supplied data buffer (see below).</td></tr>
<tr><td>RCE_NODATA</td><td>No data is available.</td></tr>
<tr><td>RCE_THOSTDISC</td><td>Transport: circuit disconnected.</td></tr>
<tr><td>RCE_TRCV</td><td>Transport: receive failure.</td></tr>
<tr><td>Remarks</td><td colspan="2">The RccReceive function discards any function code, reference number and qualifier included in a DDA message – see Chapter 2 for more details.

If the length of the DDA data exceeds that of the data buffer, RccReceive will return the error RCE_MOREDATA. To receive the remaining data, save the data received by the first call, and then call RccReceive again, repeating as necessary until you have received all the data.</td></tr>
<tr><td>Example</td><td colspan="2"><pre>
#include <ros/rcc.h>
#define BUFSIZE 1024
.
.
.
main() {
        .
        .
        .
        unsigned char Buffer[BUFSIZE]; /* Receive buffer */
        int BufferLength = BUFSIZE; /* Buffer length */
        int Length; /* Length of received data */
        int RetCode; /* To hold returned value */
        char ErrorStr[ERRSIZE]; /* Buffer for error message */
        .
        .
        .
        /* Loop until there is data available... */
        while ((RetCode = RccReceive(Shandle,
                Buffer,
                BufferLength,
</pre></td></tr>
</table>

```
                &Length)) == RCE_NODATA) {
        .
        . /* Do something while waiting for the data */
        .
}
/* If an error occurred... */
if (RetCode != SUCCESS) {
        /* Get the error description */
        RccError(RetCode, ErrorStr);
        /* Display an error message */
        printf("RccReceiveMsg error:%s\n ", ErrorStr);
}
else {
                .
                . /* Do something with the data */
                .
        }
        .
        .
        .
}
```

In the above example, the received data is placed in *Buffer* and the length of the received data is placed in *Length*.

See also **RccRecWait**, **RccSend**.

3.7 RccReceiveMsg

Purpose	The **RccReceiveMsg** function receives a DDA message. If no data is available, the function returns immediately.	
Synopsis	int **RccReceiveMsg**(*Shandle*, *Message*) **RCS_SREF** *Shandle*; **RCS_PMCB** *Message*;	
Parameters	*Shandle*	The session reference of the required connection, returned by **RccAccept** or **RccConnect**.
	Message	A pointer to a message control block (MCB) into which **RccReceiveMsg** will place the data received. For details of the message control block, see page 3-2.
Return value	The **RccReceiveMsg** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors:	
	RCE_ILLSREF	Illegal session reference.
	RCE_MOREDATA	The data received is longer than the size of the supplied data buffer (see below).
	RCE_NODATA	No data is available.
	RCE_QUALOVFL	The qualifier buffer is longer than 255 bytes.
	RCE_QUALTRUNC	The qualifier received is longer than the size of the supplied qualifier buffer. The qualifier is truncated.

	RCE_QUALTRUNC_MOREDATA	Neither the data buffer nor the qualifier buffer is large enough for the returned data.
	RCE_THOSTDISC	Transport: circuit disconnected.
	RCE_TRCV	Transport: receive failure.
Remarks	On return from **RccReceiveMsg**, the elements of the MCB will be set to the following:	
	Mcb.**Function**	The DDA function code.
	Mcb.**Reference**	The DDA reference number.
	Mcb.**QualBuffer**	The pointer to the qualifier buffer. The buffer will be filled with the DDA qualifier data.
	Mcb.**QualLength**	The number of bytes received in *Mcb*.**QualBuffer.** *Mcb*.**MaxQualLength** Unchanged.
	Mcb.**MaxQualLength**	Unchanged.
	Mcb.**DataBuffer**	The pointer to the DDA data buffer. The buffer will be filled with the DDA data.
	Mcb.**DataLength**	The number of bytes received in *Mcb*.**DataBuffer**.
	Mcb.**MaxDataLength**	Normally unchanged but see below.
	If the length of the DDA data exceeds that of the data buffer, **RccReceiveMsg** will return the error **RCE_MOREDATA** or **RCE_QUALTRUNC_MOREDATA**. Under these circumstances, the **MaxDataLength** element will be set to the total length of the data sent. To receive the remaining data, save the data received by the first call, and then call **RccReceiveMsg** again with the same MCB, repeating as necessary until you have received all the data. Once one byte of data has been received, **RccReceiveMsg** will wait until it has filled the supplied buffer or the end of the message has been reached. This is not normally a problem, but in exceptional circumstances, network problems could cause the transfer to take longer than usual.	
Example		

```
#include <ros/rcc.h>
#define BUFSIZE 1024
#define QUALSIZE 255
#define ERRSIZE 128
.
.
.
main () {
        .
        .
        .
        unsigned char QualBuf[QUALSIZE]; /* Qualifier buffer */
        unsigned char RcvBuf[BUFSIZE]; /* Data buffer */
        RCS_MCB Msg; /* Message Control Block */
        RCS_PMCB PtrMsg; /* Pointer to the MCB */
        int RetCode; /* To hold returned value */
        char ErrorStr[ERRSIZE]; /* Buffer for error message */
        .
        .
        .
        /* Initialise the MCB */
        Msg.Function = 0; /* DDA function code */
```

```
        Msg.Reference = 0; /* DDA reference number */
        Msg.QualLength = 0;
        Msg.DataLength = 0;
        Msg.QualBuffer = QualBuf; /* Pointer to qualifer buffer */
        Msg.DataBuffer = RcvBuf; /* Pointer to data buffer */
        /* Size of qualifier buffer */
        Msg.MaxQualLength = sizeof(QualBuf);
        /* Size of data buffer */
        Msg.MaxDataLength = sizeof(RcvBuf);
        PtrMsg = &Msg; /* Set a reference to the MCB */
        .
        .
        .
        /* Loop until there is data available... */
        while ((RetCode = RccReceiveMsg(Shandle, PtrMsg))
        == RCE_NODATA) {
                .
                . /* Do something while waiting for the data */
                .
        }
        /* If an error occurred... */
        if (RetCode != SUCCESS) {
        /* Get the error description */
        RccError(RetCode, ErrorStr);
        /* Display an error message */
        printf("RccReceiveMsg error:%s\n ", ErrorStr);
        }
        else {
                .
                . /* Do something with the data */
                .
        }
}
```

In the above example, *Msg* is declared as a message control block and initialised. A formatted DDA message will be returned in *Msg* by **RccReceiveMsg**.

See also **RccRecWaitMsg**, **RccSendMsg**.

3.8 RccRecWait

Purpose	The **RccRecWait** function receives data from a remote environment. If no data is available, the function waits.	
Synopsis	int **RccRecWait**(*Shandle*, *Buffer*, *BufferLength*, *RcvLength*) **RCS_SREF** *Shandle*; **unsigned char *** *Buffer*; **int** *BufferLength*; **int *** *RcvLength*;	
Parameters	*Shandle*	The session reference of the required connection, returned by **RccAccept** or **RccConnect**.
	Buffer	A pointer to a buffer in which **RccRecWait** will return the received data.

	BufferLength	An integer specifying the size of the receive buffer in bytes.
	RcvLength	A pointer to a variable in which the length of the data returned in *Buffer* will be returned.
Return value	The **RccRecWait** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors:	
	RCE_ILLSREF	Illegal session reference.
	RCE_MOREDATA	The data received is longer than the size of the supplied data buffer (see below).
	RCE_THOSTDISC	Transport: circuit disconnected.
	RCE_TRCV	Transport: receive failure.
Remarks	If the length of the DDA data exceeds that of the data buffer, **RccRecWait** will return the error **RCE_MOREDATA**. To receive the remaining data, save the data received by the first call, and then call **RccRecWait** again, repeating as necessary until you have received all the data.	

Example	<pre>#include <ros/rcc.h>
	#define BUFSIZE 80
	.
	.
	.
	main()
	  .
	  .
	  .
	  unsigned char Buffer[BUFSIZE];
	  int BufferLength = BUFSIZE;
	  int Length;
	  int RetCode; /* To hold returned value */
	  char ErrorStr[ERRSIZE]; /* Buffer for error message */
	  .
	  .
	  .
	  if ((RetCode = RccRecWait(Shandle, Buffer, BufferLength,
	  &Length)) != SUCCESS) {
	  /* Get the error description */
	  RccError(RetCode, ErrorStr);
	  /* Display an error message */
	  printf("RccRecWait Error :%s\n", ErrorStr);
	  }
	  .
	  .
	  .
	}</pre> In the above example, the received data is placed in *Buffer* and the length of the data is placed in *Length*.	

See also **RccReceive**, **RccSend**.

3.9 RccRecWaitMsg

Purpose	The **RccRecWaitMsg** function receives a DDA message. If no data is available, the function waits.	
Synopsis	int **RccRecWaitMsg**(*Shandle*, *Message*) **RCS_SREF** *Shandle*; **RCS_PMCB** *Message*;	
Parameters	*Shandle*	The session reference of the required connection, returned by **RccAccept** or **RccConnect**.
	Message	A pointer to a message control block (MCB) into which **RccRecWaitMsg** will place the data received. For details of the message control block, see page 3-2.
Return value	The **RccRecWaitMsg** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors:	
	RCE_ILLSREF	Illegal session reference.
	RCE_MOREDATA	The data received is longer than the size of the supplied data buffer (see below).
	RCE_QUALOVFL	The qualifier buffer is longer than 255 bytes.
	RCE_QUALTRUNC	The qualifier received is longer than the size of the supplied qualifier buffer. The qualifier is truncated.
	RCE_QUALTRUNC_MOREDATA	Neither the data buffer nor the qualifier buffer is large enough for the returned data.
	RCE_THOSTDISC	Transport: circuit disconnected.
	RCE_TRCV	Transport: receive failure.
Remarks	On return from **RccRecWaitMsg**, the elements of the MCB will be set to the following:	
	Mcb.**Function**	The DDA function code.
	Mcb.**Reference**	The DDA reference number.
	Mcb.**QualLength**	The number of bytes received in *Mcb*.**QualBuffer.**
	Mcb.**DataLength**	The number of bytes received in *Mcb*.**DataBuffer**.
	Mcb.**QualBuffer**	The pointer to the qualifier buffer. The buffer will be filled with the DDA qualifier data.
	Mcb.**DataBuffer**	The pointer to the DDA data buffer. The buffer will be filled with the DDA data.
	Mcb.**MaxQualLength**	Unchanged.
	Mcb.**MaxDataLength**	Normally unchanged but see below.
	If the length of the DDA data exceeds that of the data buffer, **RccRecWaitMsg** will return the error **RCE_MOREDATA**. Under these circumstances, the **MaxDataLength** element will be set to the *total* length of the data sent. To receive the remaining data, save the data received by the	

	first call, and then call **RccRecWaitMsg** again with the same MCB, repeating as necessary until you have received all the data.

Example	`#include <ros/rcc.h>`
	`#define BUFSIZE 1024`
	`#define QUALSIZE 255`
	`#define ERRSIZE 128`
	`.`
	`.`
	`.`
	`main () {`
	`.`
	`.`
	`.`
	`unsigned char QualBuf[QUALSIZE]; /* Qualifier buffer */`
	`unsigned char RcvBuf[BUFSIZE]; /* Data buffer */`
	`RCS_MCB Msg; /* Message Control Block */`
	`RCS_PMCB PtrMsg; /* Pointer to the MCB */`
	`int RetCode; /* To hold returned value */`
	`char ErrorStr[ERRSIZE]; /* Buffer for error message */`
	`.`
	`.`
	`.`
	`/* Initialise the MCB */`
	`Msg.Function = 0; /* DDA function code */`
	`Msg.Reference = 0; /* DDA reference number */`
	`Msg.QualLength = 0;`
	`Msg.DataLength = 0;`
	`Msg.QualBuffer = QualBuf; /* Pointer to qualifer buffer */`
	`Msg.DataBuffer = RcvBuf; /* Pointer to data buffer */`
	`/* Size of qualifier buffer */`
	`Msg.MaxQualLength = sizeof(QualBuf);`
	`/* Size of data buffer. If we set this to 0, no data will`
	`initially be returned, but Msg.MaxDataLength will be`
	`returned set to the total length of the data. We can then`
	`allocate a buffer of the correct size and call RccWaitMsg`
	`again to fetch the data. */`
	`Msg.MaxDataLength = 0;`
	`PtrMsg = &Msg; /* Set a reference to the MCB */`
	`.`
	`.`
	`.`
	`/* Wait for data to become available */`
	`RetCode = RccRecWaitMsg(Shandle, PtrMsg);`
	`/* If there is data available... */`
	`if (RetCode == RCE_MOREDATA`
	`|| RetCode == RCE_QUALTRUNC_MOREDATA) {`
	`/* Allocate a buffer to receive the data */`
	`Msg.DataBuffer = (`
	`unsigned char *)calloc(Msg.MaxDataLength,`
	`sizeof(unsigned char));`
	`/* Get the data */`
	`RetCode = RccRecWaitMsg(Shandle, PtrMsg);`
	`}`
	`/* If an error occurred... */`
	`if (RetCode != SUCCESS) {`
	`/* Get the error description */`
	`RccError(RetCode, ErrorStr);`
	`/* Display an error message */`
	`printf("RccRecWaitMsg error:%s\n ", ErrorStr);`
	`}`
	`else {`
	`.`
	`. /* Do something with the data */`

	.
	}
	/* Free up the buffer memory. */
	free(Msg.DataBuffer);
	}
	
	In the above example, *Msg* is declared as a message control block and initialised. A formatted DDA message is placed in *Msg* by **RccRecWaitMsg**.

See also **RccReceiveMsg**, **RccSendMsg**.

3.10 RccSend

Purpose	The **RccSend** function sends data to a remote host.	
Synopsis	int **RccSend**(*Shandle*, *Buffer, Length*) **RCS_SREF** *Shandle*; **unsigned char** * *Buffer*; **int** *Length*;	
Parameters	*Shandle*	The session reference of the required connection, returned by **RccAccept** or **RccConnect**.
	Buffer	A pointer to a buffer containing the data to be sent.
	Length	The number of bytes in the buffer.
Return value	The **RccSend** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors:	
	RCE_ILLSREF	Illegal session reference.
	RCE_THOSTDISC	Transport: circuit disconnected.
Remarks	Although, when you use **RccSend**, you do not provide values for the function code, reference number and qualifier, the data is transferred in DDA format.	
Example	#include <ros/rcc.h> #define BUFSIZE 1024 . . . main() . . . unsigned char Buffer[BUFSIZE]; /* Data buffer */ int Length; /* Data length */ int RetCode; /* To hold returned value */ char ErrorStr[ERRSIZE]; /* Buffer for error message */ . . . /* Prompt the user to enter some data */ printf("Enter a line of data : "); /* Fetch the data */ fgets(Buffer, BUFSIZE, stdin); Length = strlen(Buffer);	

	/* Send the data. If unsuccessful... */
	if ((RetCode = RccSend(Shandle, Buffer, Length)) != SUCCESS) {
	/* Get the error description */
	RccError(RetCode, ErrorStr);
	/* Display an error message */
	printf("RccSend Error :%s\n", ErrorStr);
	}
	.
	.
	.
	}
	
	In the above example, the data in *Buffer* (of length, *Length*) is sent across the connection referenced by *Shandle*.

See also **RccReceive**, **RccRecWait**.

3.11 RccSendMsg

Purpose	The **RccSendMsg** function sends a DDA message to a remote host.	
Synopsis	int **RccSendMsg**(*Shandle*, *Message*) **RCS_SREF** *Shandle*; **RCS_PMCB** *Message*;	
Parameters	*Shandle*	The session reference of the required connection, returned by **RccAccept** or **RccConnect**.
	Message	Pointer to a message control block (MCB) containing the DDA message to send. For details of the message control block, see page 3-2.
Return value	The **RccSendMsg** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors:	
	RCE_ILLSREF	Illegal session reference.
	RCE_QUALOVFL	The qualifier buffer is longer than 255 bytes.
	RCE_THOSTDISC	Transport: circuit disconnected.
Example	#include <ros/rcc.h> #define BUFSIZE 1024 #define QUALSIZE 255 #define ERRSIZE 128 . . . main () { . . . unsigned char QualBuf[QUALSIZE]; /* Qualifier buffer */ unsigned char SndBuf[BUFSIZE]; /* Data buffer */ RCS_MCB Msg; /* Message Control Block */ RCS_PMCB PtrMsg; /* Pointer to the MCB */ int RetCode; /* To hold returned value */ char ErrorStr[ERRSIZE]; /* Buffer for error message */	

```
.
.
.
/* Initialise the MCB */
/* We are sending only data, so the Function number,
Reference number and Qualifier length are all set
to zero */
Msg.Function = 0; /* DDA function code */
Msg.Reference = 0; /* DDA reference number */
Msg.QualLength = 0; /* Qualifier length */
Msg.QualBuffer = QualBuf; /* Pointer to qualifier buffer */
PtrMsg = &Msg; /* Set a reference to the MCB */
.
.
.
/* Prompt the user to enter some data */
printf("Enter a line of data: ");
fgets(SndBuf, BUFSIZE, stdin);
/* Set the pointer to data buffer */
Msg.DataBuffer = SndBuf;
/* Set the MCB data length */
Msg.DataLength = strlen(SndBuf);
/* Send the data */
RetCode = RccSendMsg(Shandle, PtrMsg);
/* If an error occurred... */
if (RetCode != SUCCESS) {
/* Get the error description */
RccError(RetCode, ErrorStr);
/* Display an error message */
printf("RccSendMsg error:%s\n ", ErrorStr);
}
.
.
.
}
```

In the above example, *Msg* is declared as a message control block and initialised. Data is read from the terminal into *SndBuf*, the length of *SndBuf* is calculated and written to *Msg.DataLength* and the pointer *Msg.DataBuffer* is set to *SndBuf*. The formatted DDA message is then sent across the connection referenced by *Shandle*.

See also **`RccReceiveMsg`**, **`RccRecWaitMsg`**.

3.12 RccSetAcceptOptions

Purpose	The **`RccSetAcceptOptions`** function is called to change the default settings of accept options.	
Synopsis	int **`RccSetAcceptOptions`**(*Flags*, *Timeout*) **RCS_FLAGS** *Flags*; **RCS_TIMEOUT** *Timeout*;	
Parameters	*Flags*	Must be set to one of the following: • **0**: The Timeout parameter is interpreted as minutes.

		• **RCS_SECONDS**: The Timeout parameter is interpreted as seconds.
	Timeout	A value within the range 0 to 255 (see flags above) within which a connection must be made, where 0 indicates that control is returned immediately if a client program is not awaiting this connection.
Return value	The **RccSetAcceptOptions** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A.	
Remarks	If changes to the default settings are required, **RccSetAcceptOptions** must be called before calling the **RccAccept** function. The default condition is that Flags and Timeout are set to 0. If **RccSetAcceptOptions** is used to change the accept options, the new settings become the default settings for all further accepts.	

Example	`#include <ros/rcc.h>`
	`int RetCode; /* To hold returned value */`
	`.`
	`.`
	`.`
	`if ((RetCode = RccSetAcceptOptions(0, 5)) != SUCCESS) {`
	`// Handle the error.`
	`}`
	
	In the above example, the timeout is altered to 5 minutes.	

See also **RccAccept**.

3.13 RccSetConnectOptions

Purpose	The **RccSetConnectOptions** function changes the default setting of connection options. The default condition is that Flags is set to 0 and Timeout is set to 1.	
Synopsis	int **RccSetConnectOptions**(*Flags*, *Timeout*) **RCS_FLAGS** *Flags*; **RCS_TIMEOUT** *Timeout*;	
Parameters	*Flags*	**RCS_SERVER_NOSTART** or 0. Setting Flags to 0 indicates that the remote server process will be started up automatically by the remote session manager on receipt of a **connect request**. Setting Flags to **RCS_SERVER_NOSTART** indicates that the server which responds to this client is not to be started up automatically by the session manager following a **connect request**. In other words, it must either be already running (and have performed an RccAccept) or start running within the period specified by Timeout

	Timeout	A value within the range 0 to 255 (in units of one minute) within which the server process must issue an **RccAccept**. This timeout applies regardless of whether the server is started automatically by the session manager or not. Setting timeout to 0 indicates that control is returned immediately if the server program is not already running (has been pre-started).
Return value	The **RccSetConnectOptions** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A.	
Remarks	If **RccSetConnectOptions** is used to change the connection options, the new settings become the default settings for all further connections.	

Example	`#include <ros/rcc.h>`
	`int RetCode; /* To hold returned value */`
	`.`
	`.`
	`.`
	`if ((RetCode = RccSetConnectOptions(RCS_SERVER_NOSTART, 0))`
	`!= SUCCESS) {`
	`// Handle the error.`
	`}`
	`.`
	`.`
	`.`
	
	In the above example, the connection options are set so that that the server must be running.	

See also **RccConnect**.

Section 4: Reality Filing Interface

The Reality Filing Interface (Rfc) enables a UNIX program to connect to a database and subsequently create, delete, read and write to Reality files.

4.1 Rfc functions

The Rfc functions allow a C program to connect to a database and then create, delete, clear, read from and write to Reality files.

The **`RgcStartUpServices`** macro which is part of the Rgc services must be called to initialise the Rfc services.

4.1.1 Establishing and terminating connections

Term or abbreviation	Definition
RfcConnect	Establishes a connection between the application program and a database.
RfcDisconnect	Terminates a connection established by RfcConnect.
RfcGetAccount	Saves the handle of the current account.
RfcSetAccount	Changes the account handle to that of a previously saved connection.

4.1.2 File operations

Term or abbreviation	Definition
RfcSetFileOptions	Sets options for various filing operations.
RfcOpenFile	Opens a file for reading and writing.
RfcClose	Closes a previously opened file.
RfcCreateFile	Creates a file.
RfcDeleteFile	Deletes a file.
RfcClear	Clears the contents of an open file.
RfcClearFile	Clears the contents of a file.
RfcRenameFile	Renames a file.
RfcSetRetUpdLocks	Sets retrieval and update locks for file creation.

4.1.3 Item reading and writing

Term or abbreviation	Definition
RfcRead	Reads an item from a file.

Term or abbreviation	Definition
RfcReadRest	Retrieves data which was too long to fit into a receive buffer.
RfcReadAttr	Reads an attribute from a file item.
RfcLockRead	Locks and then reads an item from a file.
RfcLockReadAttr	Locks a file item and then reads an attribute.
RfcGetHeader	Returns the header from the last item read.
RfcWrite	Writes data to a file item.
RfcWriteUnlock	Writes data to a file item. On completion, unlock the item.
RfcInsert	Inserts an item into a file.
RfcInsertUnlock	Inserts an item into a file. On completion, unlock the item.
RfcWriteAppend	Appends data to a file item.
RfcWriteAttr	Writes data to one attribute of a file item.
RfcWriteAttrUnlock	Writes data to one attribute of a file item. On completion, unlock the item.
RfcSetHeader	Sets the header for the next item written.
RfcDelete	Deletes an item from a file.

4.1.4 Locks

Term or abbreviation	Definition
RfcUnlock	Unlocks a file item.
RfcUnlockAll	Unlocks all the items in a file.
RfcSetLockMode	Sets lock control flags.

4.2 Using the Rfc functions

4.2.1 Connecting to a database

The **`RfcConnect`** function connects to a specific account on a database. For a connection to a Series 18/19 system, the **database** is the remote system name.

4.2.2 File handles

To open a file, the **`RfcOpenFile`** function must be called. The **`RfcOpenFile`** function is passed a file name and returns a file handle. This file handle is then used by all functions which perform operations on open files.

4.2.3 File names

The file name parameter (used by **RfcOpenFile** and other functions) can take one of three forms:

- '*filename*': Specifies the default data section.
- '*filename*,*dataname*': Specifies a particular data section.
- 'DICT *filename*': Specifies the dictionary section.

4.2.4 Account handles

Once connected to a database the account name can be saved to an account handle using the **RfcGetAccount** function. Having saved the account handle it is possible to use **RfcConnect** to connect to another database (or another account on the same database) and, subsequently, return to the first by simply referencing the account handle (using **RfcSetAccount**).

Note
Account handles are only required for multiple connections to databases. General rules for connecting to multiple databases are provided in Appendix B.

The Rfc functions can be divided into logical groups.

4.3 RfcClear

Purpose	Deletes all the items in an open Reality file.	
Synopsis	int **RfcClear**(*FileHandle*) **RFC_FILE** *FileHandle*;	
Parameters	*FileHandle*	The handle of the required Reality file, returned by **RfcOpenFile**.
Return value	The **RccSetConnectOptions** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A.	
	RFE_DONTKNOW	An error occurred in the underlying operating system.
	RFE_NOACCESS	Insufficient access rights.
Remarks	The file to be cleared must be open. To clear a file which is not open, use **RfcClearFile**. If the file handle references a dictionary, all items are deleted except for D pointers and self-referencing Q pointers. If the file handle references a data section, only the data section concerned will be cleared. Note that item locks are not checked nor released.	

See also **RfcClearFile**.

4.4 RfcClearFile

Purpose	Deletes all the items in a Reality file.	
Synopsis	int **RfcClearFile**(*FileName*) **char** * *FileName*;	
Parameters	*FileHandle*	A pointer to a string containing the file dictionary and/or data names. The following forms of filename may be used as required to clear dictionary or data sections, or both: • **DICT** *filename*: Clear dictionary. • [**DATA**] *filename*: Clear default data section. • *filename*,*dataname*: Clear named data section.
Return value	The **RfcClearFile** function returns SUCCESS for successful completion, or one of the return codes listed in Appendix A. The following are likely errors:	
	RFE_INVDPTR	Invalid 'D' pointer.
	RFE_NOACCESS	Insufficient access rights.
	RFE_NOACCOUNT	No current account.
	RFE_NOFILE	No file found.
Remarks	The file to be cleared should not be open. To clear a file which is open, use **RfcClear**. If a dictionary is specified, all items are deleted except D pointers and self-referencing Q pointers. Note that any item locks that may be set are ignored.	

See also **RfcClear**, **RfcSetFileOptions**.

4.5 RfcClose

Purpose	Closes a previously opened Reality file.	
Synopsis	int **RfcClose**(*FileHandle*) **RFC_FILE** *FileHandle*;	
Parameters	*FileHandle*	The handle of the required Reality file, returned by **RfcOpenFile**.
Return value	The **RfcClose** function returns SUCCESS for successful completion, or one of the return codes listed in Appendix A. The following are likely errors:	
	RFE_DONTKNOW	An error occurred in the underlying operating system.
Remarks	Any item locks held by the file server will be released.	

See also **`RfcOpenFile`**, **`RfcSetFileOptions`**.

4.6 RfcClose

Purpose	Closes a previously opened Reality file.	
Synopsis	int **`RfcClose`**(*DatabaseName, User, UserPasswd, Account, AcctPasswd*) **char *** *DatabaseName*; **char *** *User*; **char *** *UserPasswd*; **char *** *Account*; **char *** *AcctPasswd*;	
Parameters	*DatabaseName*	A pointer to a string containing the name of the database. • For programs linked with the Reality libraries, this must be the name of a RealityX entry in the ROUTE-FILE or the full UNIX pathname of the database. • For programs linked with the UNIX-Connect or PCSNI libraries, this must be the system name of an outgoing entry in the ROUTE-FILE.
	User	A pointer to a string containing the user-id or the user-id and password, in the form: *UserId*[,*Password*] If this parameter is a null string, the UNIX user-id from which the program is being run is used. For remote connections, this user-id is used to access the USERS-FILE and obtain the user-id to be used when logging on to the remote database.
	UserPasswd	A pointer to a string containing the password for the user-id specified in the *User* parameter. This parameter must be a null pointer, or point to a null string in case of the following criteria: • The password is specified in the *User* parameter. • The *User* parameter is null. • The specified user-id does not have a password.
	Account	A pointer to a string containing the account name or the account name and password, in the form: *Account [,Password]* If this parameter is a null string, the default account for the specified user-id will be used.
	AcctPasswd	A pointer to a string containing the password for the account specified in the *Account* parameter. This

		parameter must be a null pointer, or point to a null string in case of the following criteria: • The password is specified in the *Account* parameter. • The *Account* parameter is null. • The specified account does not have a password.
Return value	The **RfcConnect** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors:	
	RFE_ACCTACTIVE	Account handle has not been saved.
	RFE_DONTKNOW	An error occurred in the underlying operating system.
	RFE_INVACCPASS	Invalid logon attempt.
	RFE_INVALID	Invalid database name.
Remarks	When you connect to a database, an account handle is assigned and stored internally. If you need concurrent connections to two or more databases (or to different accounts on the same database), you can fetch the account handle for the current connection by calling **RfcGetAccount** and store it for later use. You can then use **RfcConnect** to connect to another database (or another account on the same database), without losing your connection to the first. If you subsequently need to access the first database, you can re-establish the connection by calling the **RfcSetAccount** function, specifying the saved account handle. In a program that will establish connections to two or more databases, the first connection must be a dummy outer connection. **RfcGetAccount** is used to fetch the account handle for this outer connection, which must be kept open until all subsequent connections have been closed. A more detailed description of connecting to multiple databases is provided in Appendix B.	

See also **RfcDisconnect**.

4.7 RfcCreateFile

Purpose	Creates a Reality file in the current account.
Synopsis	int **RfcCreateFile**(*FileType, Options, FileName, CreateString*) **RFC_FILE_TYPE** *FileType*; **RFC_CREATE_OPTS** *Options*; **char *** *FileName*; **char *** *CreateString*;

Parameters	*FileType*	Specifies the file type – currently only **RFC_DEFAULT_FILE** is supported.
	Options	This is a bit-significant parameter that must be set to a combination of the following: • **RFC_OPT_DICT**: See *FileName*. • **RFC_OPT_NOT_LOGGED**: Inhibits transaction logging. • **RFC_OPT_MOD_SEP**: Modulo/separation. Currently **RFC_OPT_MOD_SEP** must be selected – the modulo and separation values are specified in the *CreateString* parameter.
	FileName	Points to a string containing the file dictionary and/or data names. The following forms of filename may be used as required to create dictionary or data sections, or both: • *filename*: Create dictionary and/or default data section. ○ If **RFC_OPT_DICT** is not selected in the *Options* parameter, specifying the filename in this format creates the dictionary section of *filename* if it does not already exist and then creates the default data section. An error occurs if the default data section already exists. ○ If **RFC_OPT_DICT** is selected in the *Options* parameter, specifying the filename in this format creates the dictionary section of *filename*. An error occurs if the dictionary already exists. • *filename,dataname*: Creates the named data section, provided the dictionary section *filename* exists. • **DICT** *filename*: Creates the dictionary for the file *filename*. An error occurs if the dictionary already exists.
	CreateString	A pointer to a string containing the modulo and separation for the file in the form *Modulo,Separation* . If both dictionary and data sections are created, the dictionary is created with modulo and separation both set to 1.
Return value	The **`RfcCreateFile`** function returns **`SUCCESS`** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors:	
	RFE_DONTKNOW	An error occurred in the underlying operating system.
	RFE_NOACCESS	Unable to create Reality file.
	RFE_NOACCOUNT	No current account.
	RFE_NOFILE	Dictionary file does not exist.

	RFE_SECTEXISTS	Dictionary or data section already exists.

See also **RfcDeleteFile, RfcSetFileOptions, RfcSetRetUpdLocks**.

4.8 RfcDelete

Purpose	Deletes an item from a Reality file.	
Synopsis	int **RfcDelete**(*FileHandle, ItemId, ItemIdLen*) **RFC_FILE** *FileHandle*; **char *** *ItemId*; **int** *ItemIdLen*;	
Parameters	*FileHandle*	The handle of the required Reality file, returned by **RfcOpenFile**.
	Itemid	Points to a buffer containing the item-id of the item to be deleted.
	ItemIdLen	The length of the item-id.
Return value	The **RfcDelete** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors:	
	RFE_DONTKNOW	An error occurred in the underlying operating system.
	RFE_IDEXCEED	Item-id too long.
	RFE_NOITEM	Item not found.

See also **RfcInsert, RfcInsertUnlock, RfcWrite, RfcWriteAppend, RfcWriteAttr, RfcWriteAttrUnlock, RfcWriteUnlock**.

4.9 RfcDeleteFile

Purpose	Deletes all or part of a Reality file.	
Synopsis	int **RfcDeleteFile**(*FileName*) **char *** *FileName*;	
Parameters	*FileName*	Points to a string containing the file dictionary and/or data names. The following forms of filename may be used as required to delete dictionary or data sections, or both: • *filename*: Delete all data sections including the default. • **DICT** *filename:* Delete dictionary (fails if there are any data sections). • *filename,dataname***:** Delete specified data section.

Return value	The **RfcDeleteFile** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors:	
	RFE_DATA_EXISTS	Attempt to delete dictionary while data sections.
	RFE_INVDPTR	Invalid 'D' pointer.
	RFE_NOACCESS	Insufficient access rights.
	RFE_NOACCOUNT	No current account.
	RFE_NOFILE	No file found.

See also **RfcCreateFile**, **RfcSetFileOptions**.

4.10 RfcDisconnect

Purpose	Closes any open files and terminates the current connection.	
Synopsis	int **RfcDisconnect**()	
Return value	The **RfcDisconnect** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors:	
	RFE_NODATABASE	There is no current database.
Remarks	You should always close all connections before terminating your program. Note that if you have multiple concurrent connections, you must use **RfcSetAccount** to make a connection the current connection before you can close it. At the end of a program that has made connections to multiple databases, the final **RfcDisconnect** is used to close the dummy outer connection (see Appendix B).	

See also **RfcConnect**.

4.11 RfcGetAccount

Purpose	Returns the current account handle.	
Synopsis	int **RfcGetAccount**(*AccountHandle*) **RFC_ACCOUNT** * *AccountHandle*;	
Parameters	*AccountHandle*	A pointer to a variable in which to return the account handle.

Return value	The **RfcGetAccount** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors:
	RFE_NOACCOUNT — Not logged on to an account.
Remarks	When you connect to a database, an account handle is assigned and stored internally. If you need concurrent connections to two or more databases (or to different accounts on the same database), you can fetch the account handle for the current connection by calling **RfcGetAccount** and store it for later use. You can then use **RfcConnect** to connect to another database (or another account on the same database), without losing your connection to the first. If you subsequently need to access the first database, you can re-establish the connection by calling the **RfcSetAccount** function, specifying the saved account handle. If you save the account handle for each connection you make, you can switch between connections as necessary. You should always close all connections (with **RfcDisconnect**) before terminating your program. ***Note*** You should always save a connection's account handle before making another connection. If you do not, you will be unable to return to it to disconnect. In a program that will establish connections to two or more databases, **RfcGetAccount** must always be used to save the account handle for the dummy outer connection (see Appendix B).

See also **RfcConnect**, **RfcSetAccount**.

4.12 RfcGetHeader

Purpose	Returns the date and flags information from the header of the last item read.	
Synopsis	void **RfcGetHeader**(*Flags, Date*) **RFC_IFLAGS** * *Flags*; **RGC_DATE** * *Date*;	
Parameters	*Flags*	A variable in which to return the flags setting. The value returned will be one of the following: • **RFC_IFLAG_DPTR**: The item is a D pointer. • **RFC_IFLAG_BINARY**: The item is a binary item.

	Date	A variable in which to return the item date. The date will be in internal Reality format.

See also **RfcSetHeader**.

4.13 RfcInsert

Purpose	Inserts an item into a Reality file.	
Synopsis	int **RfcInsert**(*FileHandle, ItemId, ItemIdLen, Item, ItemLen*) **RFC_FILE** *FileHandle*; **char *** *ItemId*; **int** *ItemIdLen*; **char *** *Item*; **int** *ItemLen*;	
Parameters	*FileHandle*	The handle of the required Reality file, returned by **RfcOpenFile**.
	ItemId	A pointer to a buffer containing the item-id of the item to be inserted.
	ItemIdLen	The length of the item-id in ItemId.
	Item	A pointer to a buffer containing the data to be stored in the inserted item.
	ItemLen	The length of the item data.
Return value	The **RfcInsert** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors:	
	RFE_DONTKNOW	An error occurred in the underlying operating system.
	RFE_IDEXCEED	Item-id too long.
	RFE_IEXISTS	The item already exists.
Remarks	**RfcInsert** does not change the states of any item locks – if the item is locked when the function is called, on completion it will remain locked. Items are normally written as standard Reality textual items. Other types of items may be written by calling the **RfcSetHeader** function to set the appropriate header flags before calling **RfcInsert**.	

See also **RfcDelete**, **RfcInsertUnlock**, **RfcWrite**, **RfcWriteAppend**, **RfcWriteAttr**, **RfcWriteAttrUnlock**, **RfcWriteUnlock**.

4.14 RfcInsertUnlock

Purpose	Inserts an item into a Reality file. On completion, the item is unlocked (cf. **RfcInsert**).	
Synopsis	int **RfcInsertUnlock**(*FileHandle, ItemId, ItemIdLen, Item, ItemLen*) **RFC_FILE** *FileHandle*; **char *** *ItemId*; **int** *ItemIdLen*; **char *** *Item*; **int** *ItemLen*;	
Parameters	*FileHandle*	The handle of the required Reality file, returned by **RfcOpenFile**.
	ItemId	A pointer to a buffer containing the item-id of the item to be inserted.
	ItemIdLen	The length of the item-id in ItemId.
	Item	A pointer to a buffer containing the data to be stored in the inserted item.
	ItemLen	The length of the item data.
Return value	The **RfcInsertUnlock** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors:	
	RFE_DONTKNOW	An error occurred in the underlying operating system.
	RFE_IDEXCEED	Item-id too long.
	RFE_IEXISTS	The item already exists.
Remarks	Items are normally written as standard Reality textual items. Other types of items may be written by calling the **RfcSetHeader** function to set the appropriate header flags before calling **RfcInsertUnlock**.	

See also **RfcDelete**, **RfcInsert**, **RfcWrite**, **RfcWriteAppend**, **RfcWriteAttr**, **RfcWriteAttrUnlock**, **RfcWriteUnlock**.

4.15 RfcLockRead

Purpose	Locks an item in a Reality file and then returns the contents.
Synopsis	int **RfcLockRead**(*FileHandle, ItemId, ItemIdLen, Item, ItemMaxLen, ItemLen*) **RFC_FILE** *FileHandle*; **char *** *ItemId*; **int** *ItemIdLen*; **char *** *Item*; **int** *ItemMaxLen*; **int** *ItemLen*;

Parameters	*FileHandle*	The handle of the required Reality file, returned by **RfcOpenFile**.
	ItemId	A pointer to a buffer containing the item-id of the item to be inserted.
	ItemIdLen	The length of the item-id in *ItemId*.
	Item	A pointer to a buffer containing the data to be stored in the inserted item.
	ItemMaxLen	The length of the *Item* buffer.
	ItemLen	A pointer to a variable in which the length of the item data will be returned. If the complete item was too long to fit into the buffer, this variable will be returned set to the total length of the item if known, or to zero.
Return value	The **RfcLockRead** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors:	
	RFE_DONTKNOW	An error occurred in the underlying operating system.
	RFE_IDEXCEED	Item-id too long.
	RFE_LOCKED	Item is locked.
	RFE_NOITEM	Item not found.
	RFE_READEXCEED	Item too long for buffer (see below).
Remarks	The operation of **RfcLockRead** depends on the flags set with the **RfcSetLockMode** function. • If the lock mode has not been set, or is set to **RFC_OPT_NONE**, **RfcLockRead** will wait for a locked item to be released and will not lock a non-existent item. • If the RFC_OPT_NO_WAIT option is set, if the item is locked, **RfcLockRead** will return immediately with the error **RFE_LOCKED**. • If the **RFC_OPT_HOLD** option is set and the item does not exist, **RfcLockRead** will set an item lock. If the length of the item is greater than the length of the Item buffer, the data is truncated and the error **RFE_READEXCEED** is returned. The **RfcReadRest** function must then be called to read the remainder of the item.	

See also **RfcLockReadAttr**, **RfcRead**, **RfcReadRest**, **RfcSetLockMode**.

4.16 RfcLockReadAttr

Purpose	Locks an item in a Reality file and then returns the contents of a specified attribute.

Synopsis	int **RfcLockReadAttr**(*FileHandle, ItemId, ItemIdLen, AttrNum, Attr, AttrMaxLen, AttrLen*) **RFC_FILE** *FileHandle*; **char *** *ItemId*; **int** *ItemIdLen*; **int** *AttrNum*; **char *** *Attr*; **int** *AttrMaxLen*; **int *** *AttrLen*;	
Parameters	*FileHandle*	The handle of the required Reality file, returned by **RfcOpenFile**.
	ItemId	A pointer to a buffer containing the item-id of the item to be inserted.
	ItemIdLen	The length of the item-id in *ItemId*.
	AttrNum	The number of the required attribute.
	Attr	A pointer to a buffer in which the contents of the attribute will be returned.
	AttrMaxLen	The length of the *Attr* buffer.
	AttrLen	A pointer to a variable in which the length of the attribute data will be returned. If the complete attribute was too long to fit into the buffer, the value returned in this variable will be undefined.
Return value	The **RfcLockReadAttr** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors:	
	RFE_DONTKNOW	An error occurred in the underlying operating system.
	RFE_IDEXCEED	Item-id too long.
	RFE_LOCKED	Item is locked.
	RFE_NOITEM	Item not found.
	RFE_READEXCEED	Item too long for buffer (see below).
Remarks	The operation of **RfcLockReadAttr** depends on the flags set with the **RfcSetLockMode** function. • If the lock mode has not been set, or is set to **RFC_OPT_NONE**, **RfcLockReadAttr** will wait for a locked item to be released and will not lock a non-existent item. • If the **RFC_OPT_NO_WAIT** option is set, if the item is locked, **RfcLockReadAttr** will return immediately with the error **RFE_LOCKED**. • If the **RFC_OPT_HOLD** option is set and the item does not exist, **RfcLockReadAttr** will set an item lock. If the length of the item is greater than the length of the Item buffer, the data is truncated and the error **RFE_READEXCEED** is returned. The	

	RfcReadRest function must then be called to read the remainder of the item.

See also **RfcLockRead, RfcReadAttr, RfcReadRest, RfcSetLockMode.**

4.17 RfcOpenFile

Purpose	Opens a Reality file in the current account and returns a file handle.	
Synopsis	int **RfcOpenFile**(*FileName*, *FileHandle*) **char *** *FileName*; **RFC_FILE *** *FileHandle*;	
Parameters	*FileName*	A pointer to a string containing the file dictionary and/or data names. The following forms of filename may be used as required to open dictionary or data sections: • **DICT** *filename*: Open dictionary. • *filename*: Open default data section. • *filename,dataname*: Open named data section. • *FileHandle*: A pointer to a variable in which to return the handle of the open file.
Return value	The **RfcOpenFile** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors:	
	RFE_DONTKNOW	Invalid options or invalid file name.
	RFE_INVDPTR	Invalid 'D' pointer.
	RFE_NOACCESS	Insufficient access rights.
	RFE_NOACCOUNT	No current account.
	RFE_NOFILE	No file found.
Remarks	The file handle returned must be used for all subsequent references to the file. By default, a file is opened with item overwrite enabled. To prevent overwriting of items, use **RfcSetFileOptions** to set the **RFC_OPT_NO_OVERWRITE** option before opening the file.	

See also **RfcClose, RfcSetFileOptions.**

4.18 RfcRead

Purpose	Returns the contents of an item from a Reality file.
Synopsis	int **RfcRead**(*FileHandle, ItemId, ItemIdlen, Item, ItemMaxLen, ItemLen*) **RFC_FILE** *FileHandle*;

	char * *ItemId*; **int** *ItemIdLen*; **char *** *Item*; **int** *ItemMaxLen*; **int *** *ItemLen*;	
Parameters	*FileHandle*	The handle of the required Reality file, returned by **RfcOpenFile**.
	ItemId	A pointer to a buffer containing the item-id.
	ItemIdLen	The length of the item-id in *ItemId*.
	Item	A pointer to a buffer containing the data to be stored in the inserted item.
	ItemMaxLen	The length of the *Item* buffer.
	ItemLen	A pointer to a variable in which the length of the item data will be returned. If the complete item was too long to fit into the buffer, this variable will be returned set to the total length of the item if known, or to zero.
Return value	The **RfcRead** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors:	
	RFE_DONTKNOW	An error occurred in the underlying operating system.
	RFE_IDEXCEED	Item-id too long.
	RFE_NOITEM	Item not found.
	RFE_READEXCEED	Item too long for buffer (see below).
Remarks	If the length of the item is greater than the length of the buffer, the data is truncated and the error **RFE_READEXCEED** is returned. The **RfcReadRest** function must then be called to read the remainder of the item. The header flags and Reality date for the item can be obtained by calling **RfcGetHeader**. Note, however, that this must be done before any other file operation is performed.	

See also **RfcLockRead**, **RfcReadAttr**, **RfcReadRest**, **RfcGetHeader**.

4.19 RfcReadAttr

Purpose	Returns the contents of a specified attribute from a Reality file item.
Synopsis	int **RfcReadAttr**(*FileHandle, ItemId, ItemIdlen, AttrNum, Attr, AttrMaxLen, AttrLen*) **RFC_FILE** *FileHandle*; **char *** *ItemId*; **int** *ItemIdLen*; **int** *AttrNum*; **char *** *Attr*;

	int *AttrMaxLen*; **int *** *AttrLen*;	
Parameters	*FileHandle*	The handle of the required Reality file, returned by **RfcOpenFile**.
	ItemId	A pointer to a buffer containing the item-id of the item to be inserted.
	ItemIdLen	The length of the item-id in *ItemId*.
	AttrNum	The number of the required attribute.
	Attr	A pointer to a buffer in which the contents of the attribute will be returned.
	AttrMaxLen	The length of the *Attr* buffer.
	AttrLen	A pointer to a variable in which the length of the attribute data will be returned. If the complete attribute was too long to fit into the buffer, the value returned in this variable will be undefined.
Return value	The **RfcReadAttr** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors:	
	RFE_DONTKNOW	An error occurred in the underlying operating system.
	RFE_IDEXCEED	Item-id too long.
	RFE_NOITEM	Item not found.
	RFE_READEXCEED	Item too long for buffer (see below).
Remarks	If the length of the attribute is greater than the length of the buffer, the data is truncated and the error **RFE_READEXCEED** is returned. Note that the only way to read the remainder of the attribute is to try again with a larger buffer – the **RfcReadRest** function cannot be used.	

See also **RfcRead**, **RfcLockReadAttr**.

4.20 RfcReadRest

Purpose	Retrieves successive blocks of data as a continuation of the data returned from a previous function call.	
Synopsis	int **RfcReadRest**(*FileHandle, Item, ItemMaxLen, DataLen*) **RFC_FILE** *FileHandle*; **char *** *Item*; **int** *ItemMaxLen*; **int *** *DataLen*;	
Parameters	*FileHandle*	The handle of the required Reality file, returned by **RfcOpenFile**.
	Item	A pointer to a buffer in which the item data will be returned.

	ItemMaxLen	The length of the Item buffer.
	DataLen	A pointer to a variable in which the length of the item data will be returned. If the complete item was too long to fit into the buffer, this variable will be returned set to the total length of the item if known, or to zero.
Return value	The **RfcReadRest** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors:	
	RFE_DONTKNOW	An error occurred in the underlying operating system.
	RFE_NOREAD	Call not preceded by an RFE_READEXCEED error.
	RFE_READEXCEED	Item too long for buffer (see below).
Remarks	If a file item-reading function (**RfcLockRead** or **RfcRead**) completes with the code **RFE_READEXCEED**, this indicates that the supplied buffer was not large enough to hold the item. **RfcReadRestshould** be used as many times as is necessary to fetch the rest of the item. If, on completion, there is still more data to come, **RfcReadRest** will return the code **RFE_READEXCEED**. The end of the data is indicated by the completion code SUCCESS (0). If there is no more data to come (that is, the last **RfcReadRest** call returned **SUCCESS**), the error **RFE_NOREAD** will be returned.	

See also **RfcLockRead**, **RfcRead**.

4.21 RfcRenameFile

Purpose	Renames a file or part of a file.	
Synopsis	int **RfcRenameFile**(*OldName, NewName*) **char *** *OldName*; **char *** *NewName*;	
Parameters	*OldName*	A pointer to a string containing the file name.
	NewName	A pointer to a string containing the new file name.
Return value	The **RfcRenameFile** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors:	
	RFE_NOFILE	The file *OldName* does not exist.
	RFE_IEXISTS	The item *NewName* already exists in dictionary, either in MD or in *OldName* dictionary, as data section file.
	RFE_INVDPTR	Invalid **D** pointer.

	RFE_NOACCESS	Insufficient access rights.
	RFE_NOACCOUNT	No current account.
Remarks	The *OldName* and *NewName* parameters must have the same format, which must be one of the following:	
	filename	The dictionary and default data section are
	filename,dataname	The specified data section is renamed.

See also **RfcCreateFile**, **RfcSetFileOptions**.

4.22 RfcSetAccount

Purpose	Sets the current account handle.	
Synopsis	int **RfcSetAccount**(*AccountHandle*) **RFC_ACCOUNT** *AccountHandle*;	
Parameters	*AccountHandle*	The handle of the account that is to be made the current account.
Return value	The **RfcSetAccount** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors:	
	RFE_ACCTACTIVE	The current handle has not been saved.
Remarks	When you connect to a database, an account handle is assigned and stored internally. If you need concurrent connections to two or more databases (or to different accounts on the same database), you can fetch the account handle for the current connection by calling **RfcGetAccount** and store it for later use. You can then use **RfcConnect** to connect to another database (or another account on the same database), without losing your connection to the first. If you subsequently need to access the first database, you can re-establish the connection by calling **RfcSetAccount**, specifying the saved account handle. If you save the account handle for each connection you make, you can switch between connections as necessary. You should always close all connections (with **RfcDisconnect**) before terminating your program. You should always save a connection's account handle before making another connection. If you do not, you will be unable to return to it to disconnect. If you attempt to switch to an old account without first saving the current account, the error code **RFE_ACCTACTIVE** is returned.	

	At the end of a program that has established connections to two or more databases, **RfcSetAccount** is used to restore the dummy outer connection so that this outer connection can be closed (see Appendix B).

See also **RfcConnect**, **RfcGetAccount**.

4.23 RfcSetFileOptions

Purpose	The **RfcSetFileOptions** function sets the file options for the next call to **RfcOpenFile**.	
Synopsis	void **RfcSetFileOptions**(*Options*) **RFC_FILE_OPTS** *Options*;	
Parameters	*Options*	A combination of the following bit-significant options: • **RFC_OPT_DICT**: Open the dictionary section of the file. • **RFC_OPT_NO_OVERWRITE**: Inhibit overwriting of existing items. • **RFC_OPT_NOT_LOGGED**: Inhibit transaction logging on this file.
Remarks	With the default setting, the data section of the file is opened (unless specified otherwise), existing items can be overwritten and transaction logging is enabled. The file options are reset to the default values after each call to **RfcOpenFile**.	

See also **RfcOpenFile**, **RfcWrite**, **RfcWriteAppend**, **RfcWriteAttr**, **RfcWriteAttrUnlock**, **RfcWriteUnlock**.

4.24 RfcSetHeader

Purpose	Sets the item header flags for the next item to be written.	
Synopsis	void **RfcSetHeader**(*Flags*) **RFC_IFLAGS** *Flags*;	
Parameters	*Flags*	One of the following options: • **RFC_IFLAG_BINARY**: Binary item. • **RFC_IFLAG_DPTR**: D pointer.
Remarks	The flags are reset after the item is written; the next write will therefore use the default setting (normal text item) unless RfcSetHeader is called again. The write item flags are not affected by reading an item. To duplicate an item, do the following:	

	1. Read the item using **RfcRead** or **RfcLockRead**. 2. Then read the item's flags using **RfcGetHeader**. 3. Set the flags for the new item using **RfcSetHeader**. 4. Write the new item using **RfcInsert** or **RfcWrite**.

See also **RfcGetHeader**, **RfcInsert**, **RfcInsertUnlock**, **RfcWrite**, **RfcWriteUnlock**.

4.25 RfcSetLockMode

Purpose	Sets the lock control flag for calls to the lock and read functions (**RfcLockRead** and **RfcLockReadAttr**).	
Synopsis	void **RfcSetLockMode**(*Flags*) **RFC_LOCK_OPTS** *Flags*;	
Parameters	*Flags*	A combination of the following bit-significant options: • **RFC_OPT_NONE**: The lock and read functions wait for locked items to be released, and do not lock non-existent items. • **RFC_OPT_NO_WAIT**: If the item is locked, the lock and read functions return immediately with the error RFE_LOCKED. • **RFC_OPT_HOLD**: If the item does not exist, the lock and read functions set an item lock.

See also **RfcLockRead**, **RfcLockReadAttr**.

4.26 RfcSetRetUpdLocks

Purpose	Sets the retrieval and update locks for the next file to be created using **RfcCreateFile**.	
Synopsis	void **RfcSetRetUpdLocks**(*RetLocks, UpdLocks*) **char** * *RetLocks*; **char** * *UpdLocks*;	
Parameters	*RetLocks*	A pointer to a string containing the required retrieval locks. Multiple retrieval locks must be separated by commas.
	UpdLocks	A pointer to a string containing the required update locks. Multiple update locks must be separated by commas.

Remarks	The retrieval and update locks are stored as attributes 5 and 6 of the D-pointers created using **RfcCreateFile**. **RfcCreateFile** uses the default retrieval and update security codes for the current account, unless specifically changed with **RfcSetRetUpdLocks**. Note, however, that **RfcSetRetUpdLocks** affects only the next call to **RfcCreateFile** – subsequent calls revert to the default settings.

See also **RfcCreateFile**.

4.27 RfcUnlock

Purpose	Unlocks an item in a Reality file.	
Synopsis	void **RfcUnlock**(*FileHandle, ItemId, ItemIdLen*) **RFC_FILE** *FileHandle*; **char *** *ItemId*; **int** *ItemIdLen*;	
Parameters	*FileHandle*	The handle of the required Reality file, returned by **RfcOpenFile**.
	ItemId	A pointer to a buffer containing the item-id.
	ItemIdLen	The length of the item-id.
Return value	The **RfcUnlock** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A.	

See also **RfcLockRead**, **RfcLockReadAttr**, **RfcUnlockAll**.

4.28 RfcUnlockAll

Purpose	Unlocks all locked items in a Reality file.	
Synopsis	void **RfcUnlockAll**(*FileHandle*) **RFC_FILE** *FileHandle*;	
Parameters	*FileHandle*	The handle of the required Reality file, returned by **RfcOpenFile**.
Return value	The **RfcUnlockAll** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A.	

See also **RfcLockRead**, **RfcLockReadAttr**, **RfcUnlock**.

4.29 RfcWrite

Purpose	Writes data to an item in a Reality file.	
Synopsis	int **RfcWrite**(*FileHandle, ItemId, ItemIdLen, Item, ItemLen*) **RFC_FILE** *FileHandle*; **char *** *ItemId*; **int** *ItemIdLen*; **char *** *Item*; **int** *ItemLen*;	
Parameters	*FiletHandle*	The handle of the required Reality file, returned by **RfcOpenFile**.
	ItemId	A pointer to a buffer containing the item-id of the item to be written.
	ItemIdLen	The length of the item-id in *ItemId*.
	Item	A pointer to a buffer containing the data to be stored in the item.
	ItemLen	The length of the item data.
Return value	The **RfcWrite** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors:	
	RFE_DONTKNOW	An error occurred in the underlying operating system.
	RFE_IDEXCEED	Item-id too long.
	RFE_IEXISTS	Item already exists and overwrite flag not set.
Remarks	**RfcWrite** does not change the states of any item locks – if the item is locked when the function is called, on completion it will remain locked. Items are normally written as standard Reality textual items. Other types of items may be written by calling the **RfcSetHeader** function to set the appropriate header flags before calling **RfcWrite**. If required, overwriting can be disabled for the next write operation, by using **RfcSetFileOptions** to set the **RFC_OPT_NO_OVERWRITE** option. If this has been done, and the specified item already exists, **RfcWrite** will fail and return the error **RFE_IEXISTS**.	

See also **RfcInsert**, **RfcWriteAppend**, **RfcWriteAttr**, **RfcWriteUnlock**.

4.30 RfcWriteAppend

Purpose	Appends data to an existing Reality item.
Synopsis	int **RfcWriteAppend**(*FileHandle, ItemId, ItemIdLen, Item, ItemLen*)

	RFC_FILE *FileHandle*; **char *** *ItemId*; **int** *ItemIdLen*; **char *** *Item*; **int** *ItemLen*;	
Parameters	*FiletHandle*	The handle of the required Reality file, returned by **RfcOpenFile**.
	ItemId	A pointer to a buffer containing the item-id of the item to be written.
	ItemIdLen	The length of the item-id in *ItemId*.
	Item	A pointer to a buffer containing the data to be stored in the item.
	ItemLen	The length of the item data.
Return value	The **RfcWriteAppend** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors:	
	RFE_DONTKNOW	An error occurred in the underlying operating system.
	RFE_IDEXCEED	Item-id too long.
	RFE_IEXISTS	Item already exists and overwrite flag not set.
	RFE_NOITEM	Item not found.
Remarks	**RfcWriteAppend** does not change the states of any item locks – if the item is locked when **RfcWriteAppend** is called, on completion it will remain locked. If required, overwriting can be disabled for the next write operation, by using **RfcSetFileOptions** to set the **RFC_OPT_NO_OVERWRITE** option. If this has been done, and the specified item already exists, **RfcWriteAppend** will fail and return the error **RFE_IEXISTS**.	

See also **RfcInsert**, **RfcInsertUnlock**, **RfcWrite**, **RfcWriteAttr**, **RfcWriteAttrUnlock**, **RfcWriteUnlock**.

4.31 RfcWriteAttr

Purpose	Writes data to one attribute of an item in a Reality file.
Synopsis	int **RfcWriteAttr**(*FileHandle, ItemId, ItemIdLen, AttrNum, Attr, AttrLen*) **RFC_FILE** *FileHandle*; **char *** *ItemId*; **int** *ItemIdLen*; **int** *AttrNum*; **char *** *Attr*;

	int *AttrLen*;	
Parameters	*FiletHandle*	The handle of the required Reality file, returned by **RfcOpenFile**.
	ItemId	A pointer to a buffer containing the item-id of the item to be written.
	ItemIdLen	The length of the item-id in *ItemId*.
	AttrNum	The attribute number.
	Attr	A pointer to a buffer containing the data to be stored in the attribute.
	AttrLen	The length of the attribute data.
Return value	The **RfcWriteAttr** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors:	
	RFE_DONTKNOW	An error occurred in the underlying operating system.
	RFE_IDEXCEED	Item-id too long.
	RFE_IEXISTS	Item already exists and overwrite flag not set.
	RFE_NOITEM	Item does not exist.
Remarks	**RfcWrite** does not change the states of any item locks – if the item is locked when **RfcWrite** is called, on completion it will remain locked. If required, overwriting can be disabled for the next write operation, by using **RfcSetFileOptions** to set the **RFC_OPT_NO_OVERWRITE** option. If this has been done, and the specified item already exists, **RfcWriteAttr** will fail and return the error **RFE_IEXISTS**.	

See also **RfcInsert**, **RfcWrite**, **RfcWriteAppend**, **RfcWriteAttrUnlock**.

4.32 RfcWriteAttrUnlock

Purpose	Writes data to one attribute of an item in a Reality file. On completion, the item is unlocked (cf. **RfcWriteAttr**).
Synopsis	int **RfcWriteAttrUnlock**(*FileHandle, ItemId, ItemIdLen, AttrNum, Attr, AttrLen*) **RFC_FILE** *FileHandle*; **char** * *ItemId*; **int** *ItemIdLen*; **int** *AttrNum*; **char** * *Attr*; **int** *AttrLen*;

Parameters	*FiletHandle*	The handle of the required Reality file, returned by **RfcOpenFile**.
	ItemId	A pointer to a buffer containing the item-id of the item to be written.
	ItemIdLen	The length of the item-id in *ItemId*.
	AttrNum	The attribute number.
	Attr	A pointer to a buffer containing the data to be stored in the attribute.
	AttrLen	The length of the attribute data.
Return value	The **RfcWriteAttrUnlock** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors:	
	RFE_DONTKNOW	An error occurred in the underlying operating system.
	RFE_IDEXCEED	Item-id too long.
	RFE_IEXISTS	Item already exists and overwrite flag not set.
	RFE_NOITEM	Item does not exist.
Remarks	If required, overwriting can be disabled for the next write operation, by using **RfcSetFileOptions** to set the **RFC_OPT_NO_OVERWRITE** option. If this has been done, and the specified item already exists, **RfcWriteAttrUnlock** will fail and return the error **RFE_IEXISTS**.	

See also **RfcInsertUnlock, RfcWriteAttr, RfcWriteUnlock.**

4.33 RfcWriteUnlock

Purpose	Writes data to one attribute of an item in a Reality file. On completion, the item is unlocked (cf. **RfcWrite**).	
Synopsis	int **RfcWriteUnlock**(*FileHandle, ItemId, ItemIdLen, Item, ItemLen)* **RFC_FILE** *FileHandle*; **char *** *ItemId*; **int** *ItemIdLen*; **char *** *Item*; **int** *ItemLen*;	
Parameters	*FiletHandle*	The handle of the required Reality file, returned by **RfcOpenFile**.
	ItemId	A pointer to a buffer containing the item-id of the item to be written.
	ItemIdLen	The length of the item-id in *ItemId*.
	Item	A pointer to a buffer containing the data to be stored in the item.

	ItemLen	The length of the item data.
Return value	The **RfcWriteUnlock** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors:	
	RFE_DONTKNOW	An error occurred in the underlying operating system.
	RFE_IEXISTS	Item already exists and overwrite flag not set.
	RFE_IDEXCEED	Item-id too long.
Remarks	Items are normally written as standard Reality textual items. Other types of item may be written by calling the **RfcSetHeader** function to set the appropriate header flags before calling **RfcWriteUnlock**. If required, overwriting can be disabled for the next write operation, by using **RfcSetFileOptions** to set the **RFC_OPT_NO_OVERWRITE** option. If this has been done, and the specified item already exists, **RfcWriteUnlock** will fail and return the error **RFE_IEXISTS**.	

See also **RfcInsertUnlock, RfcWrite, RfcWriteAttrUnlock.**

Section 5: Reality General Service Interface

The Reality General Services Interface (Rgc) functions enable a C program to manipulate the elements of a Reality database: items, attributes, values and subvalues.

5.1 Reality General Services Interface Functions

The Reality General Services Interface functions provide a means of manipulating the elements of a Reality database, that is, items, attributes, values and subvalues.

Caution

None of these functions operate directly on a database. They operate on local data that has typically been read from a database with the Rfc functions (RfcRead, and so on....).

There are also functions for starting up and shutting down the Interactive File Access (IFA) services, for reporting errors and for retrieving the time and date in Reality format.

The Rgc functions are listed below.

5.1.1 Services

Services	Definition
RgcStartUpServices	Initializes the Interactive File Access services.
RgcShutDownServices	Shuts down all active Interactive File Access services.
RgcErrMsg	Retrieve the error message that corresponds to a return code.
RgcPerror	Displays an error message.

5.1.2 String manipulation

Services	Definition
RgcDeleteAttr	Deletes an attribute.
RgcDeleteSubValue	Deletes a subvalue.
RgcDeleteValue	Deletes a value.
RgcFindAttr	Finds the location of an attribute within an item.
RgcFindValue	Finds the location of a value within an attribute.
RgcFindSubValue	Finds the location of a subvalue within a value.
RgcGetAttr	Extracts an attribute from an item.
RgcGetNumAttr	Converts an attribute to a numeric value.

Services	Definition
RgcGetSubValue	Extracts a subvalue from an item.
RgcGetValue	Extracts a value from an item.
RgcInsertAttr	Inserts an attribute into an item.
RgcInsertNumAttr	Converts a numeric value to a string and inserts the result into an item as an attribute.
RgcInsertNumSubValue	Converts a numeric value to a string and inserts the result into an item as a subvalue.
RgcInsertNumValue	Converts a numeric value to a string and inserts the result into an item as a value.
RgcInsertSubValue	Inserts a subvalue into an item.
RgcInsertValue	Inserts a value into an item.
RgcSetAttr	Sets the contents of an attribute.
RgcSetNumAttr	Sets an attribute to a numeric value.
RgcSetNumSubValue	Sets a subvalue to a numeric value.
RgcSetNumValue	Sets a value to a numeric value.
RgcSetSubValue	Sets the contents of a subvalue.
RgcSetValue	Sets the contents of a value.

5.1.3 Time and data

Services	Definition
RgcGetTimeDate	Gets the time and date in internal Reality format.

5.2 RgcDeleteAttr

Purpose	**`RgcDeleteAttr`** deletes an attribute from a file item.
Synopsis	int **`RgcDeleteAttr`**(*Item , ItemLen, AttrNo, NewItemLen)* **char *** *Item*; **int** *ItemLen*; **int** *AttrNo*; **int *** *NewItemLen*;

Parameters	*Item*	A pointer to a buffer containing the item from which the attribute is to be deleted.
	ItemLen	The length of the item in the *Item* buffer.
	AttrNo	The number of the attribute to delete.
	NewItemLen	A pointer to a variable in which the length of the modified item will be returned.
Return value	The **RgcDeleteAttr** function always returns **SUCCESS**. It is not considered an error if the specified attribute could not be found.	

See **also RgcDeleteSubValue, RgcDeleteValue**.

5.3 RgcDeleteSubValue

Purpose	**RgcDeleteSubValue** deletes a subvalue from a specified attribute and value in a file item.	
Synopsis	int **RgcDeleteSubValue**(*Item , ItemLen, AttrNo, ValueNo, SubValueNo, NewItemLen)* **char *** *Item*; **int** *ItemLen*; **int** *AttrNo*; **int** *ValueNo*; **int** *SubValueNo*; **int *** *NewItemLen*;	
Parameters	*Item*	A pointer to a buffer containing the item from which the attribute is to be deleted.
	ItemLen	The length of the item in the Item buffer.
	AttrNo	The number of the attribute containing the subvalue to delete.
	ValueNo	The number of the value containing the subvalue to delete.
	SubValueNo	The number of the subvalue to delete.
	NewItemLen	A pointer to a variable in which the length of the modified item will be returned.
Return value	The **RgcDeleteSubValue** function always returns **SUCCESS**. It is not considered an error if the specified subvalue could not be found.	

See also **RgcDeleteAttr, RgcDeleteValue.**

5.4 RgcDeleteValue

Purpose	**RgcDeleteSubValue** deletes a value from a specified attribute and value in a file item.

Synopsis	int **RgcDeleteValue**(*Item , ItemLen, AttrNo, ValueNo, NewItemLen)* **char *** *Item*; **int** *ItemLen*; **int** *AttrNo*; **int** *ValueNo*; **int *** *NewItemLen*;	
Parameters	*Item*	A pointer to a buffer containing the item from which the attribute is to be deleted.
	ItemLen	The length of the item in the *Item* buffer.
	AttrNo	The number of the attribute containing the value to delete.
	ValueNo	The number of the value to delete.
	NewItemLen	A pointer to a variable in which the length of the modified item will be returned.
Return value	The **RgcDeleteValue** function always returns **SUCCESS**. It is not considered an error if the specified value could not be found.	

See also **RgcDeleteAttr**, **RgcDeleteSubValue**.

5.5 RgcErrMsg

Purpose	Retrieve the error description that corresponds to a return code.	
Synopsis	char * **RgcErrMsg**(*ErrorCode)* **int** *ErrorCode*;	
Parameters	*ErrorCode*	A status code returned by a function.
Return value	The **RgcErrMsg** function returns a pointer to a buffer holding the corresponding error description. The description is null terminated.	
Remarks	Subsequent calls to this function will use the same buffer.	

See also **RgcPerror**.

5.6 RgcFindAttr

Purpose	**RgcFindAttr** finds the location of a specified attribute within an item.
Synopsis	char * **RgcFindAttr**(*Item, ItemLen, AttrNo, Length)* **char *** *Item*; **int** *ItemLen*; **int** *AttrNo*;

	int * *Length*;	
Parameters	*Item*	A pointer to a buffer containing the item.
	ItemLen	The length of the item in the *Item* buffer.
	AttrNo	The number of the required attribute.
	Length	A pointer to a variable in which the length of the specified attribute will be returned.
Return value	The **RgcFindAttr** function returns a pointer to start of the specified attribute. If the specified attribute is not found, the function returns a null pointer and the *Length* parameter is set to zero.	
Remarks	Subsequent calls to this function will use the same buffer.	

See also **RgcFindSubValue, RgcFindValue.**

5.7 RgcFindSubValue

Purpose	**RgcFindSubValue** finds the location of a specified subvalue within an item.	
Synopsis	char * **RgcFindSubValue**(*Item, ItemLen, AttrNo, ValueNo, SubValueNo, Length)* **char *** *Item*; **int** *ItemLen*; **int** *AttrNo*; **int** *ValueNo*; **int** *SubValueNo*; **int *** *Length*;	
Parameters	*Item*	A pointer to a buffer containing the item.
	ItemLen	The length of the item in the *Item* buffer.
	AttrNo	The number of the attribute containing the required subvalue.
	ValueNo	The number of the value containing the required subvalue.
	SubValueNo	The number of the required subvalue.
	Length	A pointer to a variable in which the length of the specified attribute will be returned.
Return value	The **RgcFindSubValue** function returns a pointer to start of the specified subvalue. If the specified subvalue is not found, the function returns a null pointer and the *Length* parameter is set to zero.	
Remarks	Subsequent calls to this function will use the same buffer.	

See also **RgcFindAttr, RgcFindValue.**

5.8 RgcFindValue

Purpose	**RgcFindValue** finds the location of a specified value within an item.	
Synopsis	char * **RgcFindValue**(*Item, ItemLen, AttrNo, ValueNo, Length)* **char *** *Item*; **int** *ItemLen*; **int** *AttrNo*; **int** *ValueNo*; **int *** *Length*;	
Parameters	*Item*	A pointer to a buffer containing the item.
	ItemLen	The length of the item in the *Item* buffer.
	AttrNo	The number of the attribute containing the required value.
	ValueNo	The number the required value.
	Length	A pointer to a variable in which the length of the specified attribute will be returned.
Return value	The **RgcFindValue** function returns a pointer to start of the specified value. If the specified value is not found, the function returns a null pointer and the *Length* parameter is set to zero.	

See also **RgcFindAttr, RgcFindSubValue.**

5.9 RgcGetAttr

Purpose	**RgcGetAttr** extracts an attribute from an item.	
Synopsis	char * **RgcGetAttr**(*Item, ItemLen, AttrNo, Data, DataLen)* **char *** *Item*; **int** *ItemLen*; **int** *AttrNo*; **char *** *Data*; **int *** *DataLen*;	
Parameters	*Item*	A pointer to a buffer containing the item.
	ItemLen	The length of the item in the *Item* buffer.
	AttrNo	The number of the required attribute.
	Data	A pointer to a buffer in which the contents of the attribute will be returned. If this pointer is null, the function allocates a buffer using the malloc() function.

	DataLen	A pointer to a variable in which the length of the specified attribute will be returned.
Return value	The **RgcGetAttr** function returns a pointer to the buffer containing the required attribute. If the specified attribute is not found, the function returns a null pointer and the *DataLen* parameter is set to zero.	
Remarks	The user is responsible for freeing any buffers allocated by this function. If you supply a buffer in which to return the data, you must ensure that it is large enough. You can do this by first calling **RgcFindAttr** to obtain the length of the data.	

See also **RgcGetNumAttr**, **RgcGetSubValue**, **RgcGetValue**.

5.10 RgcGetNumAttr

Purpose	**RgcGetNumAttr** converts an attribute to a numeric value.	
Synopsis	int **RgcGetNumAttr**(*ItemPtr, ItemLen, AttrNo, Number)* **char** * *ItemPtr*; **int** *ItemLen*; **int** *AttrNo*; **long** * *Number*;	
Parameters	*ItemPtr*	A pointer to a buffer containing the item.
	ItemLen	The length of the item in the *Item* buffer.
	AttrNo	The number of the required attribute.
	Number	A pointer to a variable in which the value of the attribute will be returned.
Return value	The **RgcGetNumAttr** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following is the most likely error: • **RGE_NOTNUM**: The attribute did not contain a valid number.	
Remarks	For an attribute to be recognised as a number it must contain only the characters '+', '-' and '0' to '9'. Leading white space is permitted but ignored. There must be no space between the sign (if any) and the first digit.	

See also **RgcGetAttr**, **RgcGetNumValue**, **RgcGetNumSubValue**.

5.11 RgcGetSubValue

Purpose	**RgcGetSubValue** extracts a subvalue from an item.

Synopsis	char * **RgcGetSubValue**(*Item, ItemLen, AttrNo, ValueNo, SubValueNo, Data, DataLen)* **char *** *Item*; **int** *ItemLen*; **int** *AttrNo*; **int** *ValueNo*; **int** *SubValueNo*; **char *** *Data*; **int *** *DataLen*;	
Parameters	*Item*	A pointer to a buffer containing the item.
	ItemLen	The length of the item in the *Item* buffer.
	AttrNo	The number of the attribute containing the required subvalue.
	ValueNo	The number of the value containing the required subvalue.
	SubValueNo	The number of the required subvalue.
	Data	A pointer to a buffer in which the contents of the subvalue will be returned. If this pointer is null, the function allocates a buffer using the malloc() function.
	DataLen	A pointer to a variable in which the length of the specified subvalue will be returned.
Return value	The **RgcGetSubValue** function returns a pointer to the buffer containing the required subvalue. If the specified subvalue is not found, the function returns a null pointer and the *DataLen* parameter is set to zero.	
Remarks	The user is responsible for freeing any buffers allocated by this function. If you supply a buffer in which to return the data, you must ensure that it is large enough. You can do this by first calling **RgcFindSubValue** to obtain the length of the data.	

See also **RgcGetAttr**, **RgcGetNumAttr**, **RgcGetValue**.

5.12 RgcGetTimeDate

Purpose	Gets the time and date in internal Reality format.	
Synopsis	void **RgcGetTimeDate**(*Time, Date)* **long *** *Time*; **long *** *Date*;	
Parameters	*Item*	A pointer to a buffer containing the item.

	Time	A pointer to a variable in which the time will be returned. The value returned is the number of milliseconds since midnight.
	Date	A pointer to a variable in which the date will be returned. The value returned is the number of days since 31st December 1967.

5.13 RgcGetValue

Purpose	**`RgcGetValue`** extracts a value from an item.	
Synopsis	char * **`RgcGetValue`**(*Item, ItemLen, AttrNo, ValueNo, Data, DataLen)* **char *** *Item*; **int** *ItemLen*; **int** *AttrNo*; **int** *ValueNo*; **char *** *Data*; **int *** *DataLen*;	
Parameters	*Item*	A pointer to a buffer containing the item.
	ItemLen	The length of the item in the *Item* buffer.
	AttrNo	The number of the attribute containing the required subvalue.
	ValueNo	The number of the value containing the required value.
	Data	A pointer to a buffer in which the contents of the value will be returned. If this pointer is null, the function allocates a buffer using the malloc() function.
	DataLen	A pointer to a variable in which the length of the specified value will be returned.
Return value	The **`RgcGetValue`** function returns a pointer to the buffer containing the required value. If the specified value is not found, the function returns a null pointer and the *DataLen* parameter is set to zero.	
Remarks	The user is responsible for freeing any buffers allocated by this function. If you supply a buffer in which to return the data, you must ensure that it is large enough. You can do this by first calling **`RgcFindValue`** to obtain the length of the data.	

See also **`RgcGetAttr`**, **`RgcGetNumAttr`**, **`RgcGetSubValue`**.

5.14 RgcInsertAttr

Purpose	**`RgcInsertAttr`** inserts an attribute into an item.

Synopsis	int **RgcInsertAttr**(*Item, ItemLen, AttrNo, Data, DataLen, ItemMaxLen, NewItemlen)* **char *** *Item*; **int** *ItemLen*; **int** *AttrNo*; **char *** *Data*; **int** *DataLen*; **int** *ItemMaxLen*; **int *** *NewItemLen*;	
Parameters	*Item*	A pointer to a buffer containing the item into which the data is to be inserted.
	ItemLen	The length of the item in the *Item* buffer.
	AttrNo	The number of the attribute before which the new attribute will be inserted.
	Data	A pointer to the data to be inserted.
	DataLen	The length of the data to be inserted.
	ItemMaxLen	The maximum length of the Item buffer.
	NewItemLen	A pointer to a variable in which the length of the modified item will be returned.
Return value	The **RgcInsertAttr** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following is the most likely error: • **RGE_NOSPACE**: The data was too long to fit in the *Item* buffer.	
Remarks	If the attribute number specified is greater than the number of existing attributes, a new attribute is appended to the item. If necessary, the item is extended with null attributes, so that the new attribute will have the specified attribute number. If you specify an attribute number of -1, the new attribute will be appended to the end of the item.	

See also **RgcInsertNumAttr**, **RgcInsertValue**, **RgcInsertSubValue**, **RgcSetAttr**.

5.15 RgcInsertNumAttr

Purpose	**RgcInsertNumAttr** converts a numeric value to a string and inserts the result into an item as an attribute.
Synopsis	int **RgcInsertNumAttr**(*Item, ItemLen, AttrNo, Number, DataLen, ItemMaxLen, NewItemlen)* **char *** *Item*; **int** *ItemLen*; **int** *AttrNo*;

	long *Number*; **int** *ItemMaxLen*; **int *** *NewItemLen*;	
Parameters	*Item*	A pointer to a buffer containing the item into which the data is to be inserted.
	ItemLen	The length of the item in the *Item* buffer.
	AttrNo	The number of the attribute before which the new attribute will be inserted.
	Number	The numeric value to be inserted.
	ItemMaxLen	The maximum length of the *Item* buffer.
	NewItemLen	A pointer to a variable in which the length of the modified item will be returned.
Return value	The **RgcInsertNumAttr** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following is the most likely error: • **RGE_NOSPACE**: The data was too long to fit in the *Item* buffer.	
Remarks	If the attribute number specified is greater than the number of existing attributes, a new attribute is appended to the item. If necessary, the item is extended with null attributes, so that the new attribute will have the specified attribute number. If you specify an attribute number of -1, the new attribute will be appended to the end of the item.	

See also **RgcInsertAttr**, **RgcInsertNumSubValue**, **RgcInsertNumValue**, **RgcSetNumAttr**.

5.16 RgcInsertNumSubValue

Purpose	**RgcInsertNumSubValue** converts a numeric value to a string and inserts the result into an item as a subvalue.	
Synopsis	int **RgcInsertNumSubValue**(*Item, ItemLen, AttrNo, ValueNo, SubValueNo, Number, ItemMaxLen, NewItemlen)* **char *** *Item*; **int** *ItemLen*; **int** *AttrNo*; **int** *ValueNo*; **int** *SubValueNo*; **long** *Number*; **int** *ItemMaxLen*; **int *** *NewItemLen*;	
Parameters	*Item*	A pointer to a buffer containing the item into which the data is to be inserted.
	ItemLen	The length of the item in the *Item* buffer.

	AttrNo	The number of the attribute in which the new subvalue will be inserted.
	ValueNo	The number, within the specified attribute, of the value in which the new subvalue will be inserted.
	SubValueNo	The number, within the specified value, of the subvalue before which the new subvalue will be inserted.
	Number	The numeric value to be inserted.
	ItemMaxLen	The maximum length of the *Item* buffer.
	NewItemLen	A pointer to a variable in which the length of the modified item will be returned.
Return value	The **`RgcInsertNumSubValue`** function returns **`SUCCESS`** for successful completion, or one of the return codes listed in Appendix A. The following is the most likely error: • **RGE_NOSPACE**: The data was too long to fit in the *Item* buffer.	
Remarks	If the attribute number specified in the *AttrNo* parameter is greater than the number of existing attributes, a new attribute is appended to the item. If necessary, the item is extended with null attributes, so that the new attribute will have the specified attribute number. If you specify an attribute number of -1, the new attribute will be appended to the end of the item. The above also applies to the *ValueNo* and *SubValueNo* parameters.	

See also **`RgcInsertNumAttr`**, **`RgcInsertNumValue`**, **`RgcInsertSubValue`**, **`RgcSetNumSubValue`**.

5.17 RgcInsertNumValue

Purpose	**`RgcInsertNumValue`** converts a numeric value to a string and inserts the result into an item as a subvalue.	
Synopsis	int **`RgcInsertNumSubValue`**(*Item, ItemLen, AttrNo, ValueNo, Number, ItemMaxLen, NewItemlen)* **char *** *Item*; **int** *ItemLen*; **int** *AttrNo*; **int** *ValueNo*; **int** *SubValueNo*; **long** *Number*; **int** *ItemMaxLen*; **int *** *NewItemLen*;	
Parameters	*Item*	A pointer to a buffer containing the item into which the data is to be inserted.

	ItemLen	The length of the item in the *Item* buffer.
	AttrNo	The number of the attribute in which the new value will be inserted.
	ValueNo	The number, within the specified attribute, of the value in which the new value will be inserted.
	Number	The numeric value to be inserted.
	ItemMaxLen	The maximum length of the *Item* buffer.
	NewItemLen	A pointer to a variable in which the length of the modified item will be returned.
Return value	The **RgcInsertNumValue** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following is the most likely error: • **RGE_NOSPACE**: The data was too long to fit in the *Item* buffer.	
Remarks	If the attribute number specified in the *AttrNo* parameter is greater than the number of existing attributes, a new attribute is appended to the item. If necessary, the item is extended with null attributes, so that the new attribute will have the specified attribute number. If you specify an attribute number of -1, the new attribute will be appended to the end of the item. The above also applies to the *ValueNo* parameter.	

See also **RgcInsertNumAttr, RgcInsertNumSubValue, RgcInsertValue, RgcSetNumValue.**

5.18 RgcInsertSubValue

Purpose	**RgcInsertSubValue** inserts data into an item as a subvalue.	
Synopsis	int **RgcInsertSubValue**(*Item, ItemLen, AttrNo, ValueNo, SubValueNo, Data, DataLen, ItemMaxLen, NewItemlen)* **char *** *Item*; **int** *ItemLen*; **int** *AttrNo*; **int** *ValueNo*; **int** *SubValueNo*; **char *** *Data*; **int** *DataLen*; **int** *ItemMaxLen*; **int *** *NewItemLen*;	
Parameters	*Item*	A pointer to a buffer containing the item into which the data is to be inserted.

	ItemLen	The length of the item in the *Item* buffer.
	AttrNo	The number of the attribute in which the new subvalue will be inserted.
	ValueNo	The number, within the specified attribute, of the value into which the new subvalue will be inserted.
	SubValueNo	The number, within the specified value, of the subvalue before which the new subvalue will be inserted.
	Data	A pointer to the data to be inserted.
	DataLen	The length of the data to be inserted.
	ItemMaxLen	The maximum length of the *Item* buffer.
	NewItemLen	A pointer to a variable in which the length of the modified item will be returned.
Return value	The **`RgcInsertSubValue`** function returns **`SUCCESS`** for successful completion, or one of the return codes listed in Appendix A. The following is the most likely error: • **RGE_NOSPACE**: The data was too long to fit in the *Item* buffer.	
Remarks	If the attribute number specified in the *AttrNo* parameter is greater than the number of existing attributes, a new attribute is appended to the item. If necessary, the item is extended with null attributes, so that the new attribute will have the specified attribute number. If you specify an attribute number of -1, the new attribute will be appended to the end of the item. The above also applies to the *ValueNo* and *SubValueNo* parameters.	

See also **`RgcInsertAttr`**, **`RgcInsertNumSubValue`**, **`RgcInsertValue`**, **`RgcSetValue`**.

5.19 RgcInsertValue

Purpose	**`RgcInsertSubValue`** inserts data into an item as a value.
Synopsis	int **`RgcInsertValue`**(*Item, ItemLen, AttrNo, ValueNo, Data, DataLen, ItemMaxLen, NewItemlen)* **char *** *Item*; **int** *ItemLen*; **int** *AttrNo*; **int** *ValueNo*; **char *** *Data*; **int** *DataLen*; **int** *ItemMaxLen*; **int *** *NewItemLen*;

Parameters	*Item*	A pointer to a buffer containing the item into which the data is to be inserted.
	ItemLen	The length of the item in the *Item* buffer.
	AttrNo	The number of the attribute in which the new value will be inserted.
	ValueNo	The number within the specified attribute of the value into which the new value will be inserted.
	Data	A pointer to the data to be inserted.
	DataLen	The length of the data to be inserted.
	ItemMaxLen	The maximum length of the *Item* buffer.
	NewItemLen	A pointer to a variable in which the length of the modified item will be returned.
Return value	The **`RgcInsertValue`** function returns **`SUCCESS`** for successful completion, or one of the return codes listed in Appendix A. The following is the most likely error: • **RGE_NOSPACE**: The data was too long to fit in the *Item* buffer.	
Remarks	If the attribute number specified in the *AttrNo* parameter is greater than the number of existing attributes, a new attribute is appended to the item. If necessary, the item is extended with null attributes, so that the new attribute will have the specified attribute number. If you specify an attribute number of -1, the new attribute will be appended to the end of the item. The above also applies to the *ValueNo* parameter.	

See also **`RgcInsertAttr`**, **`RgcInsertNumValue`**, **`RgcInsertSubValue`**, **`RgcSetValue`**.

5.20 RgcPerror

Purpose	Displays the error description that corresponds to a specified return code, prefixed with the name of a function.	
Synopsis	void **`RgcPerror`**(*FuncName, ErrorCode)* **char *** *FuncName*; **int** *ErrorCode*;	
Parameters	*Item*	A pointer to a buffer containing the item into which the data is to be inserted.
	ItemLen	The length of the item in the *Item* buffer.

See also **`RgcErrMsg`**.

5.21 RgcSetAttr

Purpose	**RgcSetAttr** writes data to an item as an attribute.	
Synopsis	int **RgcSetAttr**(*Item, ItemLen, AttrNo, Data, DataLen, ItemMaxLen, NewItemlen)* **char *** *Item*; **int** *ItemLen*; **int** *AttrNo*; **char *** *Data*; **int** *DataLen*; **int** *ItemMaxLen*; **int *** *NewItemLen*;	
Parameters	*Item*	A pointer to a buffer containing the item into which the data is to be inserted.
	ItemLen	The length of the item in the *Item* buffer.
	AttrNo	The number of the attribute to be written.
	Data	A pointer to the data to be written.
	DataLen	The length of the data to be written.
	ItemMaxLen	The maximum length of the *Item* buffer.
	NewItemLen	A pointer to a variable in which the length of the modified item will be returned.
Return value	The **RgcSetAttr** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following is the most likely error: • **RGE_NOSPACE**: The data was too long to fit in the *Item* buffer.	
Remarks	If the attribute number specified is greater than the number of existing attributes, a new attribute is appended to the item. If necessary, the item is extended with null attributes, so that the new attribute will have the specified attribute number. If you specify an attribute number of -1, the new attribute will be appended to the end of the item.	

See only **RgcInsertAttr, RgcInsertSubValue, RgcInsertValue.**

5.22 RgcSetNumAttr

Purpose	**RgcSetAttr** writes data to an item as an attribute.
Synopsis	int **RgcSetNumAttr**(*Item, ItemLen, AttrNo, Number, ItemMaxLen, NewItemlen)*

	char * *Item*; **int** *ItemLen*; **int** *AttrNo*; **long** *Number*; **int** *ItemMaxLen*; **int *** *NewItemLen*;	
Parameters	*Item*	A pointer to a buffer containing the item into which the attribute is to be written.
	ItemLen	The length of the item in the *Item* buffer.
	AttrNo	The number of the attribute to be written.
	Number	The numeric value to be written.
	ItemMaxLen	The maximum length of the Item buffer.
	NewItemLen	A pointer to a variable in which the length of the modified item will be returned.
Return value	The **RgcSetNumAttr** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following is the most likely error: • **RGE_NOSPACE**: The data was too long to fit in the *Item* buffer. • **RGE_MALLOC**: Cannot allocate memory.	
Remarks	If the attribute number specified in the *AttrNo* parameter is greater than the number of existing attributes, a new attribute is appended to the item. If necessary, the item is extended with null attributes, so that the new attribute will have the specified attribute number. If you specify an attribute number of -1, the new attribute will be appended to the end of the item.	

See also **RgcInsertAttr, RgcInsertNumAttr, RgcSetNumSubValue, RgcSetNumValue.**

5.23 RgcSetNumSubValue

Purpose	**RgcSetNumSubValue** converts a numeric value to a string and writes the result to an item as a subvalue.
Synopsis	int **RgcSetNumSubValue** (*Item, ItemLen, AttrNo, ValueNo, SubValueNo, Number, ItemMaxLen, NewItemlen)* **char *** *Item*; **int** *ItemLen*; **int** *AttrNo*; **int** *ValueNo*; **int** *SubValueNo*; **long** *Number*; **int** *ItemMaxLen*; **int *** *NewItemLen*;

Parameters	*Item*	A pointer to a buffer containing the item into which the attribute is to be written.
	ItemLen	The length of the item in the *Item* buffer.
	AttrNo	The number of the attribute to be written.
	ValueNo	The number of the value to be written.
	SubValueNo	The number of the subvalue to be written.
	Number	The numeric value to be written.
	ItemMaxLen	Maximum length of buffer available.
	NewItemLen	Pointer to an integer, returned with the new length of the item.
Return value	The **RgcSetNumSubValue** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following is the most likely error: • **RGE_NOSPACE**: The data was too long to fit in the *Item* buffer. • **RGE_MALLOC**: Cannot allocate memory.	
Remarks	If the attribute number specified in the *AttrNo* parameter is greater than the number of existing attributes, a new attribute is appended to the item. If necessary, the item is extended with null attributes, so that the new attribute will have the specified attribute number. If you specify an attribute number of -1, the new attribute will be appended to the end of the item. The above also applies to the *ValueNo* and *SubValueNo* parameters.	

See also **RgcInsertNumSubValue**, **RgcInsertSubValue**, **RgcSetNumAttr**, **RgcSetNumValue**, **RgcSetSubValue**.

5.24 RgcSetNumValue

Purpose	**RgcSetNumValue** converts a numeric value to a string and writes the result to an item as a value.
Synopsis	int **RgcSetNumValue** (*Item, ItemLen, AttrNo, ValueNo, SubValueNo, Number, ItemMaxLen, NewItemlen)* **char *** *Item*; **int** *ItemLen*; **int** *AttrNo*; **int** *ValueNo*; **long** *Number*; **int** *ItemMaxLen*; **int *** *NewItemLen*;

Parameters	*Item*	A pointer to a buffer containing the item into which the attribute is to be written.
	ItemLen	The length of the item in the *Item* buffer.
	AttrNo	The number of the attribute to be written.
	ValueNo	The number of the value to be written.
	Number	The numeric value to be written.
	ItemMaxLen	Maximum length of buffer available.
	NewItemLen	Pointer to an integer, returned with the new length of the item.
Return value	The **RgcSetNumValue** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following is the most likely error: • **RGE_NOSPACE**: The data was too long to fit in the *Item* buffer. • **RGE_MALLOC**: Cannot allocate memory.	
Remarks	If the attribute number specified in the *AttrNo* parameter is greater than the number of existing attributes, a new attribute is appended to the item. If necessary, the item is extended with null attributes, so that the new attribute will have the specified attribute number. If you specify an attribute number of -1, the new attribute will be appended to the end of the item. The above also applies to the *ValueNo* parameter.	

See also **RgcInsertNumValue**, **RgcSetNumAttr**, **RgcSetNumSubValue**, **RgcSetValue**.

5.25 RgcSetSubValue

Purpose	**RgcSetSubValue** writes data to an item as a subvalue.	
Synopsis	int **RgcSetSubValue** (*Item, ItemLen, AttrNo, ValueNo, SubValueNo, Data, DataLen, ItemMaxLen, NewItemlen)* **char *** *Item*; **int** *ItemLen*; **int** *AttrNo*; **int** *ValueNo*; **int** *SubValueNo*; **char *** *Data*; **int** *DataLen*; **int** *ItemMaxLen*; **int *** *NewItemLen*;	
Parameters	*Item*	A pointer to a buffer containing the item into which the attribute is to be written.

	ItemLen	The length of the item in the *Item* buffer.
	AttrNo	The number of the attribute in which the new subvalue will be written.
	ValueNo	The number within the specified attribute of the value in which the new subvalue will be written.
	SubValueNo	The number within the specified value of the subvalue to be written.
	Data	A pointer to the data to be written.
	DataLen	The length of the data to be written.
	ItemMaxLen	Maximum length of buffer available.
	NewItemLen	Pointer to an integer, returned with the new length of the item.
Return value	The **RgcSetSubValue** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following is the most likely error: • **RGE_NOSPACE**: The data was too long to fit in the *Item* buffer.	
Remarks	If the attribute number specified in the *AttrNo* parameter is greater than the number of existing attributes, a new attribute is appended to the item. If necessary, the item is extended with null attributes, so that the new attribute will have the specified attribute number. If you specify an attribute number of -1, the new attribute will be appended to the end of the item. The above also applies to the *ValueNo* and *SubValueNo* parameters.	

See only **RgcInsertSubValue**, **RgcSetAttr**, **RgcSetNumSubValue**, **RgcSetValue**.

5.26 RgcSetValue

Purpose	**RgcSetValue** writes data to an item as a value.
Synopsis	int **RgcSetValue** (*Item, ItemLen, AttrNo, ValueNo, Data, DataLen, ItemMaxLen, NewItemlen)* **char *** *Item*; **int** *ItemLen*; **int** *AttrNo*; **int** *ValueNo*; **char *** *Data*; **int** *DataLen*; **int** *ItemMaxLen*; **int *** *NewItemLen*;

Parameters	*Item*	A pointer to a buffer containing the item into which the attribute is to be written.
	ItemLen	The length of the item in the *Item* buffer.
	AttrNo	The number of the attribute in which the new value will be written.
	ValueNo	The number within the specified attribute of the value to be written.
	Data	A pointer to the data to be written.
	DataLen	The length of the data to be written.
	ItemMaxLen	Maximum length of the *Item* buffer.
	NewItemLen	A pointer to a variable in which the length of the modified item will be returned.
Return value	The **RgcSetValue** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following is the most likely error: • **RGE_NOSPACE**: The data was too long to fit in the *Item* buffer.	
Remarks	If the attribute number specified in the *AttrNo* parameter is greater than the number of existing attributes, a new attribute is appended to the item. If necessary, the item is extended with null attributes, so that the new attribute will have the specified attribute number. If you specify an attribute number of -1, the new attribute will be appended to the end of the item. The above also applies to the *ValueNo* parameter.	

See also **RgcInsertNumValue**, **RgcInsertValue**, **RgcSetAttr**, **RgcSetSubValue**.

5.27 RgcShutDownServices

Purpose	Shuts down all active Interactive File Access (IFA) services (Rgc, Rfc and Rlc).
Synopsis	void **RgcShutDownServices**()

See also **RgcStartUpServices**.

5.28 RgcStartUpServices

Purpose	Initializes the Interactive File Access (IFA) services (Rfc, Rgc and Rlc). Note that **RgcStartUpServices** is a macro, rather than a function.
Synopsis	void **RgcStartUpServices**(*Result*)

	int *Result*;	
Parameters	*Result*	A variable in which the result of the macro will be returned. ***Note*** This parameter must be a variable, not a pointer to a variable.
Remarks	**`RgcStartUpServices`** must be called from the main() function, before any other Interactive File Access service functions are called. The macro initializes only those services that are being used; that is, those for which header files have been included in the main module. ***Note*** The rgc.h header file must always be included.	

See also **`RgcShutDownServices`**.

Section 6: Reality Index Sequential Services Interface

The Reality Index Sequential Services Interface (Risc) provides an alternative to the Reality Filing Interface for connecting to a local database and accessing Reality files.

6.1 Introduction

This interface provides an alternative method of connecting to a local Reality database and accessing Reality files. The main aim of this interface is to hide the special nature of the Reality item-id from the programmer. It works with records and index keys and introduces the concept of a *current* record (see below).

A *Record* is made up from the Reality item-id and item data, separated by an attribute mark (0xFE). Each record is made up from several variable length fields separated by attribute marks. The Reality item-id appears as the first field in each record.

Although this interface makes the Reality item-id appear as part of the record data, it still has special significance to the underlying Reality File System. It is still the identifier of the record and as such must have a different value in every record. A Reality file cannot contain two different *records* with the same value in the first field (item-id).

Note
The Reality Index Sequential Services Interface can only be used to access local Reality databases.

6.1.1 Index key

An Index Key is a Reality Key Value. In the simplest case where the file is indexed on a single field with no special conversions, the Key is just the appropriate field value. In an Index defined on several fields (again with no special conversions), the Key comprises the appropriate field values separated by Attribute Marks (0xFE).

With complex Indexes that include English conversions, the relationship between the Record and the Key value is less obvious.

6.1.2 Record locking

At most one record (the current record) may be locked at a time on a single **`RISC_FILE`** handle. Any read operation that changes the current position causes the previous record to be unlocked.

Record locks are implemented using Reality Item locks. Records are identified by their item-ids for the purpose of these locks and this imposes certain limitations on the use of locks by this interface.

6.1.3 Accessing a Reality file

To access a Reality file using the C-ISAM Indexed Access Layer, you will need to do the following:

1. Call **`RgcStartUpServices`** to start the Interactive File Access (IFA) services.
2. Connect to the required Reality database using **`RiscConnect`**.
3. Open the required Reality file using **`RiscOpen`**.
4. Either use **`RiscSelect`** to associate an existing index with this file, or **`RiscCreateIndex`** to create a new index and associate it with this file.

5. Carry out the required processing on the file.
6. Close the Reality file with **RiscClose**.
7. Disconnect from the database with **RiscDisconnect**.
8. Use **RgcShutDownServices** to shut down any active Interactive File Access services.

   For example,

```
RISC_FILE FileHandle;
int StartUpResult;
RgcStartUpServices(StartUpResult);
if (StartUpResult == SUCCESS) {
if (RiscConnect("SPI-22",
"SYSPROG",
"KEY1",
"ODESSA",
"OMEGA") == SUCCESS) {
if (RiscOpen("TESTFILE", &FileHandle) == SUCCESS) {
if (RiscSelect(FileHandle, "TESTINDEX") == SUCCESS) {
/* Process the index. */
}
else {
/* handle the select error. */
}
RiscClose(FileHandle);
}
else {
/* handle the file open error. */
}
RiscDisconnect();
}
else {
/* handle the connect error. */
}
RgcShutDownServices();
}
else {
/* handle the startup error. */
}
```

   In this example, connection is made to the **ODESSA** account on the database **SPI-22**, using the user-id **SYSPROG**. The password for the **SYSPROG** user is **KEY1** and that for the **ODESSA** account is **OMEGA**. If the connection is successful, the file **TESTFILE** is opened and, if this file is opened successfully, the index **TESTINDEX** is opened for processing.

> ***Note***
> The Reality Index Sequential Services Interface can only be used to access local Reality databases.

6.1.4 The current record

The C-ISAM Indexed Access Layer uses the concept of the current record to determine which record in the index is currently accessible. At any given time, only one record is the current record.

6.1.4.1 Moving to the next or previous record

Calling the **RiscRead** function with its Direction parameter set to **RISC_NEXT** makes the next record in the index current. Generally, this is used to step through the records in an index to extract data on a record-by-record basis.

If the current record is either the first or last record of the index, any attempt to move further towards the beginning or end will return **RIXE_EOL**. You will then no longer have a valid current record.

For example,

```
char KeyBuf[KEYBUFLEN + 1];
char RecBuf[RECBUFLEN + 1];
int RecLen;
int KeyLen;
int Result;
.
.
.
while ((Result = RiscRead(FileHandle,
RISC_NEXT,
RISC_LOCK_NOWAIT,
KeyBuf, KEYBUFLEN, &KeyLen,
RecBuf, RECBUFLEN, &RecLen))
== SUCCESS) {
/* Code to work with the current record... */
}
if (Result != RIXE_EOL) {
/* handle any error */
}
```

At the end of the loop, the current record pointer is invalid.

Note

The example above assumes that none of the records is longer than the record buffer. The example on page 6-6 shows a method of handling records that are too long for the buffer.

To move to the previous record, call the **RiscRead** function with its Direction parameter set to **RISC_PREV**. Note, however, that you cannot move to the previous record if physical sequential order has been selected.

6.1.4.2 Moving to the next or previous record

To move to the beginning or end of the index, call the **RiscPosition** function with its Position parameter set to **RISS_BEG** or **RISS_END** respectively. For example,

```
RiscPosition(FileHandle, RISS_BEG, KeyBuf, KeyLen);
```

moves to the beginning of the index.

Note, however, that using **RiscPosition** in this way does not *select* the first or last record. Rather, the position is set to just before the beginning, or just after the end of the index, and using **RiscRead** to read the current record will fail. To read the first or last record, call **RiscRead** to read the next or previous record respectively. For example,

```
char KeyBuf[KEYBUFLEN + 1];
int KeyLen;
char RecBuf[RECBUFLEN + 1];
```

```
int RecLen;
.
.
.
RiscPosition(FileHandle, RISS_END, KeyBuf, KEYBUFLEN);
RiscRead(FileHandle,
RISC_PREV,
RISC_LOCK_NOWAIT,
KeyBuf, KEYBUFLEN, &KeyLen,
RecBuf, RECBUFLEN, &RecLen);
```

selects the last record.

6.1.4.3 Other ways of moving through an index

In addition to the methods described above, you can move to a specific record by using the **RiscReadByKey** function or the **RiscPosition** function with its Direction parameter set to **RISS_EQ** or **RISS_GE**.

6.1.5 Reading records

There are two ways of reading the contents of a record. They are:

- The RiscRead function allows you to read the current record, or the next or previous record in the index as described above.
- The RiscReadByKey function allows you to read a specified record. You must specify the value of the index key for the required record. For example,

```
char RecBuf[BUFLEN + 1];
int RecLen;
.
.
.
if (RiscReadByKey(FileHandle,
RISC_LOCK_NOWAIT,
"UM70006812", 10,
RecBuff, RECBUFLEN, &RecLen) == SUCCESS) {
RecBuff[RecLen] = '\0';
/* do something with the record. */
}
else {
/ handle the error.
}
```

One reason the **RiscRead** and **RiscReadByKey** functions might fail is if the record buffer supplied is too short. Under these circumstances, the first part of the record is returned in the buffer and the function returns the error **RFE_READEXCEED**. The remainder of the record can then be retrieved by calling the **RiscReadRest** function as many times as necessary. For example,

```
char KeyBuf[KEYBUFLEN + 1];
int KeyLen;
char RecBuf[RECBUFLEN + 1];
int RecLen;
char *DataBuf;
int DataBufSize;
int Result;
.
.
.
/* Attempt to read the record. */
```

```
Result = RiscRead(FileHandle,
RISC_PREV,
RISC_LOCK_NOWAIT,
KeyBuf, KEYBUFLEN, &KeyLen,
RecBuf, RECBUFLEN, &RecLen);
/* Allocate a buffer for the record. */
DataBufSize = RECBUFLEN + 1;
DataBuf = (char *)calloc(DataBufSize, sizeof(char));
/* Null terminate the data. */
if (Result == SUCCESS)
RecBuf[RecLen] = '\0';
else
RecBuf[RECBUFLEN] = '\0';
/* Copy the record data into the data buffer. */
strcpy(DataBuf, RecBuf);
/* While there is more data to come... */
while (Result == RFE_READEXCEED) {
/* Get more data. */
Result = RiscReadRest(FileHandle,
RecBuf, RECBUFLEN, &RecLen)
/* Calculate the size of the data received so far. */
DataBufSize += RecLen;
/* Make the record buffer bigger. */
DataBuf = (char *)realloc(DataBuf, DataBufSize);
/* Null terminate the record buffer. */
RecBuf[RecLen] = '\0';
/* Append the new data to the old. */
strcat(DataBuf, RecBuf);
}
/* Do something with the record. */
/* Free up the buffer memory. */
free(DataBuf);
```

6.1.6 Writing records

There are three functions you can use to write records to a Reality file:

Item-id	Definition
RiscInsert	Inserts an item into a Reality file. If an item with the specified item-id already exists, the function will fail.
RiscUpdate	Updates the current record. ***Caution*** Any part of the record, including the item-id, can be changed. If the resulting record has the same item-id as another record, that record will be overwritten.
RiscWrite	Writes data to an item in a Reality file. If an item with the specified item-id already exists, it will be overwritten; otherwise, a new item will be created.

In all cases, you must supply a file handle, a buffer containing the record data and the length of the data. For example:

```
char RecBuf[RECBUFLEN + 1];
```

```
int RecLen;
strcpy(RecBuf, "221816\0xFEWebster\0xFEMartin")
RiscWrite(FileHandle, RecBuf, strlen(RecBuf));
```

Note that the first attribute in the record is always the Reality item-id.

6.1.7 Indexes

6.1.7.1 Selecting an index

Before the items in a Reality file can be accessed using the C-ISAM Indexed Access Layer, an index must be opened. This is done using the **RiscSelect** function, specifying the name of the index required. For example,

```
if (RiscSelect(FileHandle, "TESTINDEX") == SUCCESS) {
/* Process the index. */
}
else {
/* handle any select error. */
}
```

Note that a file can also be accessed in physical sequential order. To do this, call **RiscSelect** as above but pass a null pointer instead of an index name. For example,

```
RiscSelect(FileHandle, (char *) 0)
```

6.1.7.2 Creating a new index

A new index can be created with the **RiscCreateIndex** function. The index is defined by creating an array of Index Description structures (see for details). For example,

```
RISC_DESC IndexDesc[3];
.
.
.
/* Define the index. */
/* Ascending numeric index on the third field. */
IndexDesc[0].Field = 2;
IndexDesc[0].Type = RISC_NUM;
IndexDesc[0].UpDown = RISC_UP;
IndexDesc[0].Op = 0;
/* Ascending string index on second subfield in field 6. */
IndexDesc[1].Field = 6;
IndexDesc[1].Type = RISC_STR;
IndexDesc[1].UpDown = RISC_UP;
IndexDesc[1].Op = RISC_GRP;
IndexDesc[1].Arg1 = '*';
IndexDesc[1].Arg2 = 2;
/* Descending string index on field 8. */
IndexDesc[2].Field = 8;
IndexDesc[2].Type = RISC_STR;
IndexDesc[2].UpDown = RISC_DOWN;
IndexDesc[2].Op = 0;
/* Create the index. */
if (RiscCreateIndex("TESTFILE",
"TESTINDEX",
3, IndexDesc) == SUCCESS) {
/* Do something with the new index. */
}
else {
/* Handle any error. */
```

```
}
```

6.1.7.3 Using an existing index

As an alternative to defining a new index from scratch, you can copy or modify an existing index. Use the **RiscDescribeIndex** function to fetch the details of the existing index. Then modify the Index Description structures as required and create the new index with **RiscCreateIndex.**

Note

1. If you are modifying the existing index, rather than creating a new one, you must delete the original (with **RiscDeleteIndex**) before calling **RiscCreateIndex** (see below).
2. When using **RiscDescribeIndex**, the file containing the index must have been opened using **RiscOpen**. The example below assumes that this has been done.

```
RISC_DESC IndexDesc[8];
int NumParts;
.
.
.
if (RiscDescribeIndex(FileHandle,
"TESTINDEX",
MaxParts, &NumParts, IndexDesc) == SUCCESS) {
/* Modify the index. */
/* Make the index in element 0 descending. */
IndexDesc[0].UpDown = RISC_DOWN;
/* Make the index in element 1 use the third multivalue. */
IndexDesc[1].Arg2 = 3;
/* Delete the original index */
RiscDeleteIndex("TESTFILE", "TESTINDEX")
/* Create the new index. */
if (RiscCreateIndex("TESTFILE",
"TESTINDEX",
3, IndexDesc) == SUCCESS) {
/* Do something with the new index. */
}
else {
/* Handle any error. */
}
}
else {
/* Handle any error. */
}
```

6.2 Index description structure

This structure is used in the **RiscCreateIndex** and **RiscDescribeIndex** functions to describe a simple index key or part of a complex key. A typical index description consists of an array of **RISC_DESC** structures.

```
typedef struct RiscDesc RISC_DESC;
struct RiscDesc
{
int Field;
RISC_FTYPE Type;
RISC_SDIR UpDown;
RISC_OP Op;
int Arg1;
int Arg2;
};
```

Index items	Definition
Field	Field number (Field 0 = Item Id).
Type	Field type – one of the following: • **RISC_STR**: string; • **RISC_NUM**: numeric.
UpDown	Sort direction – one of the following: • **RISC_UP**: ascending; • **RISC_DOWN**: descending.
Op	Operation code (see below).
Arg1	First argument to the operation code.
Arg2	Second argument to the operation code.

The *Op*, *Arg1* and *Arg2* members define an optional operation to perform on the basic value of the field identified by *Field*.

The *Op* member may have the following values:

Values	Definition
0	No additional operation is performed. Arg1 and Arg2 are ignored.
RISC_SUB	Extract substring: • Arg1: start column; • Arg2: length.
RISC_GRP	Extract substring: • Arg1: delimiter; • Arg2: field number (1 based).

These additional operations are equivalent to English correlatives T (Text Extraction) and G (Group Extraction).

6.3 RiscClear

Purpose	Deletes all the items in an open Reality file.	
Synopsis	int **`RiscClear`** (*FileHandle)* **RISC_FILE** *FileHandle*;	
Parameters	*FileHandle*	The handle of the required Reality file, returned by **`RiscOpen`**.
Return value	The **`RiscClear`** function returns **`SUCCESS`** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors: • **RFE_NOACCESS**: Insufficient access rights.	

Remarks	The file to be cleared must be open. If the file handle references a dictionary, all items are deleted except for D pointers and self-referencing Q pointers. If the file handle references a data section, only the data section concerned will be cleared. Note that item locks are not checked nor released.

See also **`RiscDelCurr`**, **`RiscDelete`**.

6.4 RiscClose

Purpose	Closes a previously opened Reality file.	
Synopsis	int **`RiscClose`** (*FileHandle)* **RISC_FILE** *FileHandle*;	
Parameters	*FileHandle*	The handle of the required Reality file, returned by **`RiscOpen`**.
Return value	The **`RiscClose`** function returns **`SUCCESS`** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors: • **RFE_NOACCESS**: Insufficient access rights.	
Remarks	If necessary, **`RiscClose`** will also close a previously opened index or list. Any item locks held by the file server will be released.	

See also **`RiscOpen`**.

6.5 RiscConnect

Purpose	The **`RiscConnect`** function establishes a connection to a database and logs on under the specified user-id to the named account.	
Synopsis	int **`RiscConnect`** (*DatabaseName, User, UserPasswd, Account, AcctPasswd)* **char*** *DatabaseName*; **char*** *User*; **char*** *UserPasswd*; **char*** *Account*; **char*** *AcctPasswd*;	
Parameters	*DatabaseName*	A pointer to a string containing the name of the database. This must be the name of a

		RealityX entry in the ROUTE-FILE or the full UNIX pathname of the database.
	User	A pointer to a string containing the user-id or the user-id and password, in the form: *UserId*[,*Password*] If this parameter is a null string, the UNIX user-id from which the program is being run is used. For remote connections this user-id is used to access the USERS-FILE and obtain the user-id to be used when logging on to the remote database.
	UserPasswd	A pointer to a string containing the password for the user-id specified in the User parameter. This parameter must be a null pointer, or point to a null string if: • The password is specified in the User parameter; • The User parameter is null; • The specified user-id does not have a password.
	Account	A pointer to a string containing the account name or the account name and password, in the form: *Account* [,*Password*] If this parameter is a null string, the default account for the specified user-id will be used.
	AcctPasswd	A pointer to a string containing the password for the account specified in the Account parameter. This parameter must be a null pointer, or point to a null string if: The password is specified in the Account parameter: • The Account parameter is null; • The specified account does not have a password.
Return value	The **`RiscConnect`** function returns **`SUCCESS`** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors: • **RFE_ACCTACTIVE**: There is already a current connection. • **RFE_INVACCPASS**: Invalid account or password. • **RFE_INVALID**: Invalid database name. • **RFE_INVDBASEDIR**: Invalid database. • **RFE_NOACCESS**: Insufficient access rights. • **RFE_REMOTE**: Cannot connect to a remote database.	
Remarks	This function must be called to establish a connection to a database and account before calling **`RiscOpen`** or any other Risc functions.	

	It is not possible to connect to more than one database (or more than one account on the same database) at any one time using this interface. It is an error to attempt to connect to a new database (or account) without first disconnecting from the current database. Where a program is going to connect to multiple databases in turn, a **dummy** outer connection must be made before the first **`RiscConnect`** and this outer connection must not be closed until after the final **`RiscDisconnect`**. The **dummy** connection must be made using **`RfcConnect`** so that **`RfcGetAccount`** and **`RfcSetAccount`** can be used to store and retrieve the account handle for the connection. **`RfcConnect`**, **`RfcGetAccount`** and **`RfcSetAccount`** are described in detail in Chapter 4. General rules for connecting to multiple databases are provided in Appendix B.

See also **`RiscDisconnect`**.

6.6 RiscCreateFile

Purpose	Creates a Reality file in the current account.	
Synopsis	int **`RiscCreateFile`** (*FileName, RecSize, NumRecs)* **char*** *FileName*; **int** *RecSize*; **int** *NumRecs*;	

Parameters	*FileName*	Points to a string containing the file dictionary and/or data names.
		The following forms of filename may be used as required to create dictionary or data sections, or both:
		• *filename*: Create dictionary and/or default data section.
		  ○ If the file does not exist, it is created with a dictionary and default data section.
		  ○ If the specified file exists, but does not have a default data section, a default data section is created.
		  ○ If the default data section already exists, an error occurs.
		• *filename,dataname*: Creates the named data section, provided the dictionary section filename exists.
		• **DICT** *filename*: Creates a dictionary section only. The size information specified in the RecSize and NumRecs

		parameters is used to size the dictionary.
	RecSize	The expected average record size.
	NumRecs	The expected number of records in file.
Return value	The **RiscCreateFile** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors: • **RFE_NOACCESS**: Unable to create Reality file. • **RFE_NOACCOUNT**: No current connection. • **RFE_NOFILE**: Dictionary file does not exist. • **RFE_SECTEXISTS**: Dictionary or data section already exists.	
Remarks	The size of the file created depends on the values of the *NumRecs* and *RecSize* parameters.	

See also **RiscDeleteFile**.

6.7 RiscCreateIndex

Purpose	Creates a new index.	
Synopsis	int **RiscCreateIndex** (*FileName, IndexName, NumParts, IndexDesc)* **char*** *FileName*; **char*** *IndexName*; **int** *NumParts*; **RISC_DESC*** *IndexDesc*;	
Parameters	*FileName*	The name of the data file. *FileName* may take any of the following forms: • *dictname*: Creates an index on the default data section. • *dictname,dataname*: creates index on specified section. ***Note*** The **DICT** *filename* form is not valid, because only data sections can be indexed.
	IndexName	The name for the new index.
	NumParts	The number of **RISC_DESC** structures in *IndexDesc*.
	IndexDesc	The address of an array of **RISC_DESC** structures describing the index.
Return value	The **RiscCreateIndex** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors:	

	• **RFE_NOFILE**: File does not exist. • **RFE_SECTEXISTS**: Data section does not exist.
Remarks	Creates a new index as defined by the structures pointed to by *IndexDesc*.

See also **RiscDeleteIndex**.

6.8 RiscDelCurr

Purpose	Deletes the current record from the specified file.	
Synopsis	int **RiscDelCurr** (*FileHandle)* **RISC_FILE** *FileHandle*;	
Parameters	*FileHandle*	The handle of the required Reality file, returned by **RiscOpen**.
Return value	The **RiscDelCurr** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors: • **RFE_NOLOCK**: Current record not locked. • **RFE_NOREAD**: No current record.	
Remarks	The current record must previously have been locked. On completion, the lock is released.	

See also **RiscDelete**.

6.9 RiscDelete

Purpose	Deletes the current record from the specified file.	
Synopsis	int **RiscDelete**(*FileHandle, KeyVal, KeyLen)* **RISC_FILE** *FileHandle*; **char*** *KeyVal*; **int** *KeyLen*;	
Parameters	*FileHandle*	The handle of the required Reality file, returned by **RiscOpen**.
	KeyVal	A pointer to a key value.
	KeyLen	The length of the key value.
Return value	The **RiscDelete** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors: • **RIXE_NOT_FOUND**: No such key.	

Remarks	**RiscDelete** deletes the first record with key equal to *KeyVal*, from the open file specified by *FileHandle*.

See also **RiscDelCurr**.

6.10 RiscDeleteFile

Purpose	Deletes all or part of a Reality file.	
Synopsis	int **RiscDeleteFile**(*FileName)* **char*** *FileName*;	
Parameters	*FileHandle*	Points to a string containing the file dictionary and/or data names. The following forms of filename may be used as required to delete dictionary or data sections, or both: • *filename*: Delete all data sections including the default. • **DICT** *filename*: Delete dictionary (fails if there are any data sections). • *filename,dataname*: Delete specified data section.
Return value	The **RiscDeleteFile** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors: • **RFE_DATA_EXISTS**: Attempt to delete dictionary while data sections still present. • **RFE_INVDPTR**: Invalid D pointer. • **RFE_NOACCESS**: Insufficient access rights. • **RFE_NOACCOUNT**: No current account. • **RFE_NOFILE**: No file found.	

See also **RiscCreateFile**.

6.11 RiscDeleteIndex

Purpose	Deletes the named index.	
Synopsis	int **RiscDeleteIndex**(*Filename, IndexName)* **char*** *FileName*; **char*** *IndexName*;	
Parameters	*FileName*	Points to a string containing the file dictionary and/or data names.

		The following forms of filename may be used as required to delete dictionary or data sections, or both: • *filename*: Delete all data sections including the default. • *filename,dataname*: Delete specified data section. **Note** The **DICT** *filename* form is not valid, because only data sections can be indexed.
	IndexName	The name of the index to be deleted.
Return value	The **RiscDeleteIndex** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors: • **RFE_NOFILE**: File does not exist. • **RFE_NOSECT**: Data section does not exist. • **RIXE_NOT_FOUND**: Index specified does not exist.	

See also **RiscCreateIndex**, **RiscDescribeIndex**.

6.12 RiscDescribeIndex

Purpose	Reads the description of an index.	
Synopsis	int **RiscDescribeIndex**(*FileHandle, IndexName, MaxParts, NumParts, IndexDesc)* **RISC_FILE** *FileHandle*; **char*** *IndexName*; **int** *MaxParts*; **int*** *NumParts*; **RISC_DESC*** *IndexDesc*;	
Parameters	*FileHandle*	The handle of the required Reality file, returned by **RiscOpen**.
	IndexName	The name of the index.
	MaxParts	The number of **RISC_DESC** structures available in **IndexDesc**.
	NumParts	A pointer to a variable in which the number of **RISC_DESC** structures returned in **IndexDesc** will be returned.
	IndexDesc	A pointer to an array of **RISC_DESC** structures to receive the index description.

Return value	The **RiscDeleteIndex** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors: • **RIXE_NOT_FOUND**: Index specified does not exist.

See also **RiscCreateIndex**.

6.13 RiscDisconnect

Purpose	**RiscDisconnect** closes any open files and terminates the current connection.
Synopsis	int **RiscDisconnect**()
Return value	The **RiscDisconnect** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors: • **RFE_NODATABASE**: No current connection.
Remarks	You should always close all connections before terminating your program. At the end of a program that has connected to several databases in turn, the **dummy** outer connection must be closed using **RfcDisconnect** following the final **RiscDisconnect**. **RfcSetAccount** is used to retrieve the account handle for this outer connection. **RfcDisconnect**, **RfcGetAccount** and **RfcSetAccount** are described in detail in Section 4. General rules for connecting to multiple databases are provided in Appendix B.

See also **RiscConnect**.

6.14 RiscGetMultiValues

Purpose	Gets the value and subvalue numbers for the current key.	
Synopsis	void **RiscGetMultiValues**(*FileHandle, ValNum, SubValNum)* **RISC_FILE** *FileHandle*, **Int*** *ValNum*, **Int*** *SubValNum*	
Parameters	*FileHandle*	The handle of the required Reality file, returned by **RiscOpen**.
	ValNum	A pointer to a variable in which the current value number will be returned.
	SubValNum	A pointer to a variable in which the current subvalue number will be returned.
Remarks	**RiscGetMultiValues** gets the numbers of the value and subvalue associated with the current key. This function should normally be used	

	when the key is from an exploding index. If used with a non-exploding index, both *ValNum* and *SubValNum* are returned set to 0.

6.15 RiscInsert

Purpose	Inserts an item into a Reality file.	
Synopsis	int **RiscInsert**(*FileHandle, RecBuff, RecLen)* **RISC_FILE** *FileHandle*; **char*** *RecBuff*; **int** *RecLen*;	
Parameters	*FileHandle*	The handle of the required Reality file, returned by **RiscOpen**.
	RecBuff	A pointer to a buffer containing the record to be inserted. The record must have the following format: *ItemId* 0xFE *ItemData*
	RecLen	The length of the record in the buffer.
Return value	**RiscDeleteIndex** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors: • **RFE_IDEXCEED**: Item-id too long. • **RFE_IEXISTS**: Item already exists.	
Remarks	If an item with the same item-id already exists in the file, the function will fail with error **RFE_IEXISTS**. The current index position is not changed by this function.	

See also **RiscDelCurr**, **RiscDelete**, **RiscUpdate**, **RiscWrite**.

6.16 RiscOpen

Purpose	Opens a Reality file in the current account and returns a file handle.	
Synopsis	int **RiscOpen**(*Filename*, *FileHandle)* **char *** *FileName*; **RISC_FILE *** *FileHandle*;	
Parameters	*FileHandle*	A pointer to a string containing the file dictionary and/or data names. The following forms of filename may be used as required to open dictionary or data sections:

		• **DICT** *filename*: Open dictionary. • *filename*: Open default data section. • *filename,dataname*: Open named data section. • *FileHandle*: A pointer to a variable in which to return the handle of the open file.
Return value	**`RiscOpen`** function returns **`SUCCESS`** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors: • **RFE_INVDPTR**: Invalid D pointer. • **RFE_NOACCESS**: Insufficient access rights. • **RFE_NOACCOUNT**: No current connection. • **RFE_NODATABASE**: No current connection. • **RFE_NOFILE**: No file found.	
Remarks	The file handle returned must be used for all subsequent references to the file.	

See also **`RiscClose`**.

6.17 RiscPosition

Purpose	Sets the current position to the beginning or end of the index or to a specified key value.	
Synopsis	int **`RiscPosition`**(*FileHandle, Position, KeyVal, KeyLen)* **RISC_FILE** *FileHandle*; **RISC_POS** *Position*; **char*** *KeyVal*; **int** *KeyLen*;	
Parameters	*FileHandle*	The handle of the required Reality file, returned by **`RiscOpen`**.
	Position	One of the following: • **RISS_BEG**: The current position is set before the first record in index order, so that reading the next item will read the record with the earliest key value. • **RISS_END**: The current position is set after the last record in index order, so that reading the previous item will read the record with the highest key value. • **RISS_EQ**: The current position is set immediately before the earliest record with key value equal to or greater than *KeyVal*. If the specified

		key does not exist return **RIXE_NOT_FOUND**. • **RISS_GE**: The current position is set immediately before the earliest record with key value equal to or greater than *KeyVal*. If the specified key does not exist, return 0 (success).
	KeyVal	A pointer to the key value. Ignored unless Position is **RISS_EQ** or **RISS_GE**.
	KeyLen	The length of the key value pointed to by *KeyVal*.
Return value	**`RiscPosition`** function returns **`SUCCESS`** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors: • **RFE_INVPARAM**: Invalid Position parameter or request invalid in physical sequential mode. • **RIXE_NOT_FOUND**: **`RISC_EQ`** was specified and no such key exists.	
Remarks	If physical sequential order has been selected, positioning by key value is invalid.	

See also **`RiscDelCur`**, **`RiscRead`**.

6.18 RiscRead

Purpose	Reads an item and its index key value from a Reality file.	
Synopsis	int **`RiscRead`**(*FileHandle, Direction, LockOpts, KeyBuff, MaxKeyLen KeyLen, RecBuff, MaxRecLen, RecLen)* **RISC_FILE** *FileHandle*; **RISC_DIR** *Direction*; **RISC_OPT** *LockOpts*; **char *** *KeyBuff*; **int** *MaxKeyLen*; **int *** *KeyLen*; **char *** *RecBuff*; **int** *MaxRecLen*; **int *** *RecLen*;	
Parameters	*FileHandle*	The handle of the required Reality file, returned by **`RiscOpen`**.
	Direction	One of the following: • **RISC_NEXT**: Read the next record in index order from the current position. • **RISC_PREV**: Read the previous record in Index order from the current position.

		• **RISC_CURR**: Re-read the current record.
	LockOpts	One of the following: • **RISC_LOCK_NONE**: Do not lock the record. • **RISC_LOCK_WAIT**: Lock the record. Wait if currently locked by another process. • **RISC_LOCK_NOWAIT**: Lock the record if available or return **RFE_LOCKED** if the record is currently locked by another process.
	KeyBuff	The address of a buffer in which the index key value will be returned.
	MaxKeyLen	The length of *KeyBuff*.
	KeyLen	A pointer to a variable in which the length of the key value will be returned.
	RecBuff	The address of a buffer in which the record will be returned.
	MaxRecLen	The size of the *RecBuff* buffer.
	RecLen	A pointer to a variable in which the length of the record will be returned.
Return value	**RiscRead** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors: • **RFE_INVPARAM**: Invalid *Direction* or *LockOpts* parameter or request invalid in physical sequential mode. • **RFE_LOCKED**: Record is locked by another process. • **RFE_NOREAD**: No current record. • **RFE_READEXCEED**: Record too long for *RecBuff*. • **RIXE_EOL**: At beginning or end of index. • **RIXE_KEY_TOO_BIG**: Key too long for *KeyBuff*. • **RIXE_NOT_FOUND**: No such key exists.	
Remarks	This function reads a record and its associated key value into the specified buffers. The record to be read is specified by the *Direction* parameter – **RISS_NEXT** and **RISS_PREV** respectively specify the next and previous record in index order relative to the current position, and RISS_CURR re-reads the current record. In all cases, the record read becomes the **current** record. Note that **RISS_PREV** is not valid if physical sequential order has been selected. If *FileHandle* points to a file that has been associated (using RiscSelect) with an exploding index, **RiscRead** will repeatedly return the same item ID for each multi- and/or sub-value. If the length of the record to be read is greater than the size of the buffer supplied, the data is truncated and the error **RFE_READEXCEED** is returned. If the total size of the item is known then *RecLen* will be set to this size; otherwise, *RecLen* will be set to zero. **RiscReadRest** can then be called to read the rest of the item.	

	If the length of the key to be returned is greater than the size of the key buffer the error **RFE_IDEXCEED** will be returned and the actual key size will be returned in *KeyLen*. The read may be repeated by supplying a larger buffer and specifying a *Direction* of **RISS_CURR**.

See also **`RiscReadByKey`**, **`RiscReadRest`**.

6.19 RiscReadByKey

Purpose	Read a specified item from a Reality file.	
Synopsis	int **`RiscReadByKey`**(*FileHandle, LockOpts, KeyVal, KeyLen, RecBuff, MaxRecLen, RecLen)* **RISC_FILE** *FileHandle*; **RISC_OPT** *LockOpts*; **char*** *KeyVal*; **int** *KeyLen*; **char*** *RecBuff*; **int** *MaxRecLen*; **int*** *RecLen*;	
Parameters	*FileHandle*	The handle of the required Reality file, returned by **`RiscOpen`**.
	LockOpts	One of the following: • **RISC_LOCK_NONE**: Do not lock the record. • **RISC_LOCK_WAIT**: Lock the record. Wait if currently locked by another process. • **RISC_LOCK_NOWAIT**: Lock the record if available or return **RFE_LOCKED** if the record is currently locked by another process.
	LockOpts	One of the following: • **RISC_LOCK_NONE**: Do not lock the record. • **RISC_LOCK_WAIT**: Lock the record. Wait if currently locked by another process. • **RISC_LOCK_NOWAIT**: Lock the record if available or return **RFE_LOCKED** if the record is currently locked by another process.
	KeyVal	A pointer to a key value. The length of the key value must not exceed 104 characters.
	KeyLen	The length of the key value.

	RecBuff	The address of a buffer in which the record will be returned.
	MaxRecLen	The size of the *RecBuff* buffer.
	RecLen	A pointer to a variable in which the length of the record will be returned.
Return value	**`RiscReadByKey`** function returns **`SUCCESS`** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors: • **RFE_INVPARAM**: Invalid *LockOpts* parameter or request invalid in physical sequential mode. • **RFE_LOCKED**: Record is locked by another process. • **RFE_NOREAD**: No current record. • **RFE_READEXCEED**: Record too long for *RecBuff*. • **RIXE_EOL**: At end of index. • **RIXE_KEY_TOO_BIG**: Key too long. • **RIXE_NOT_FOUND**: No such key.	
Remarks	This function reads the specified record and its associated key value into the specified buffers. It reads the first record with key equal to that specified in *KeyVal*. This record becomes the **current** record. The current position is always updated even if the requested record does not exist (**RIXE_NOT_FOUND**). In this case a subsequent call to **`RiscRead`** specifying **RISC_NEXT** will return the first record with key value greater than the key specified in **`RiscReadByKey`**. If the length of the record to be read is greater than the size of the buffer supplied, the data is truncated and the error **RFE_READEXCEED** is returned. If the total size of the item is known then *RecLen* will be set to this size; otherwise, *RecLen* will be set to zero. **`RiscReadRest`** can then be called to read the rest of the item. If physical sequential order has been selected this function can be used to read by item-id. However, in this case the record read does not affect the **current position** as seen by **`RiscRead`**.	

See also **`RiscRead`**, **`RiscReadRest`**.

6.20 RiscReadRest

Purpose	Read the remainder of a partially read record.	
Synopsis	int **`RiscReadRest`**(*FileHandle, DataBuff, MaxDataLen, DataLen)* **RISC_FILE** *FileHandle*; **char*** *DataBuff*; **int** *MaxDataLen*; **int*** *DataLen*;	
Parameters	*FileHandle*	The handle of the required Reality file, returned by **`RiscOpen`**.

	DataBuff	The address of a buffer in which the remaining data will be returned.
	MaxDataLen	The size of the *DataBuff* buffer.
	DataLen	A pointer to a variable in which the length of the data placed in *DataBuff* will be returned.
Return value	**`RiscReadRest`** function returns **`SUCCESS`** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors: • **RFE_NOREAD**: Call not preceded by an **RFE_READEXCEED** error. • **RFE_READEXCEED**: Data too long for *DataBuff*.	
Remarks	This function is used after an attempt to read a record returned the error **RFE_READEXCEED**, to read the rest of the record. **`RiscReadRest`** can only be used immediately after the failed **`RiscRead`**, **`RiscReadByKey`** or **`RiscReadRest`** call. If any other items are read from or written to the file, **`RiscReadRest`** will return the error **RFE_NOREAD**. Note that **`RiscReadRest`** can also return the error **RFE_READEXCEED** and may therefore be called again to get more of the item.	

See also **`RiscRead`**, **`RiscReadByKey`**.

6.21 RiscSelect

Purpose	Either associate a specified index with an open file and initialise it for sequential access in index order or initialise the file for access in physical sequential (group) order.	
Synopsis	int **`RiscSelect`**(*FileHandle, DataBuff, MaxDataLen, DataLen)* **RISC_FILE** *FileHandle*; **char*** *IndexName*;	
Parameters	*FileHandle*	The handle of the required Reality file, returned by **`RiscOpen`**.
	IndexName	A pointer to a string containing the name of the index. If *IndexName* is a null pointer, the file is initialised for physical sequential access.
Return value	**`RiscSelect`** function returns **`SUCCESS`** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors: • **RIXE_NO_INDEX**: Specified index does not exist.	
Remarks	Any previously selected index is closed. On successful completion, the current position is set before the first record in index order, so that reading the next item will read the record with the earliest key value.	

See also **RiscCreateIndex, RiscRead.**

6.22 RiscUnlock

Purpose	Unlocks the current record in the specified open file.	
Synopsis	int **RiscUnlock**(*FileHandle)* **RISC_FILE** *FileHandle*;	
Parameters	*FileHandle*	The handle of the required Reality file, returned by **RiscOpen**.
Return value	**RiscSelect** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A.	
Remarks	If the current record is not locked, this function does nothing.	

See also **RiscPosition, RiscRead, RiscReadByKey.**

6.23 RiscUpdate

Purpose	Updates the current record.	
Synopsis	int **RiscUpdate**(*FileHandle, RecBuff, RecLen)* **RISC_FILE** *FileHandle*; **char*** *RecBuff*; **int** *RecLen*;	
Parameters	*FileHandle*	The handle of the required Reality file, returned by **RiscOpen**.
	RecBuff	A pointer to a buffer containing the updated record.
	RecLen	The length of the record in *RecBuff*.
Return value	**RiscUpdate** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors: • **RFE_IDEXCEED**: Item-id too long. • **RFE_NOLOCK**: Current record not locked. • **RFE_NOREAD**: No current record.	
Remarks	Any part of the current record may be changed including the item-id. If the item-id is changed and another item exists with that Id it will be overwritten with the updated record and the old item deleted. The current record must be locked before calling this function. The lock will be released when this function completes.	

See also **RiscInsert, RiscWrite.**

6.24 RiscWrite

Purpose	Writes data to an item in a Reality file.	
Synopsis	int **RiscWrite**(*FileHandle, RecBuff, RecLen)* **RISC_FILE** *FileHandle*; **char*** *RecBuff*; **int** *RecLen*;	
Parameters	*FileHandle*	The handle of the required Reality file, returned by **RiscOpen**.
	RecBuff	A pointer to a buffer containing the updated record. The record must have the following format: *ItemId* 0xFE *ItemData*
	RecLen	The length of the record in *RecBuff*.
Return value	**RiscWrite** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors: • **RFE_IDEXCEED**: Item-id (field 1) too long.	
Remarks	If an item with the same item-id (the first attribute in *RecBuff*) already exists, it will be overwritten. The current index position is not altered by this function.	

See also **RiscInsert, RiscUpdate.**

Section 7: Reality List Services Interface

The Reality List Services Interface functions allow user written UNIX programs to create and manipulate Reality lists.

7.1 Rlc functions

The Reality List Services Interface (Rlc) is a library of C functions which allows a UNIX program to use Reality list handling features. Lists allow the sequential processing of files.

The **`RgcStartUpServices`** macro which is part of the Rgc services (see Chapter 5) must be called to initialise the Rlc services.

Reality lists are lists of item-ids created by list-generating English verbs. A list can be saved in a file item: this can be in POINTER-FILE or another specified file. Alternatively, a list can be dynamically created from the item-ids of an open file. For further details on lists, see English Reference Manual.

Rlc allows user-written C programs to manipulate lists in the Reality environment. Functions are provided to create lists, save and retrieve the created lists to/from files, and use the lists to access data from a specified file.

7.1.1 List handles

A list can be created from the item-ids of an open file with the **`RlcMakeList`** function. This returns a list **handle**. This list handle is used by all functions which perform operations on lists.

7.1.2 Rlc functions

The Rlc functions are listed as follows:

Functions	Definition
RlcCloseList	Closes an open list.
RlcDeleteList	Deletes a named list.
RlcGetList	Opens a previously saved list.
RlcLockReadNextItem	Obtains the next item-id from the specified list. It then locks the corresponding item in the specified file and returns the contents of that item.
RlcMakeList	Constructs a list of item-ids from an open file.
RlcNext	Reads the next item-id from an open list.
RlcReadNextItem	Obtains the next item-id from the specified list. It then reads the corresponding item in the specified file and returns the contents of that item.
RlcSaveList	Save an open list to a file item.
RlcSelect	Creates a list of item-ids selected from a Reality file.

7.2 RlcCloseList

Purpose	Closes an open list.	
Synopsis	int **`RlcCloseList`**(*ListHandle)* **RLC_LIST** *ListHandle*;	
Parameters	*ListHandle*	The handle of an open list, returned by **`RlcGetList`**, **`RlcMakeList`** or **`RlcSelect`**.
Return value	The **`RlcCloseList`** function always returns SUCCESS.	

See also **`RlcGetList`**, **`RlcMakeList`**, **`RlcSelect`**.

7.3 RlcDeleteList

Purpose	Deletes a named list.	
Synopsis	int **`RlcDeleteList`**(*FileName*, *ListName)* **char *** *FileName*; **char *** *ListName*;	
Parameters	*FileName*	A pointer to a string containing the name of the file that contains the list. If *Filename* is a null pointer, the list is deleted from the POINTER-FILE.
	ListName	A pointer to a string containing the name of the list.
Return value	The **`RlcDeleteList`** function returns **`SUCCESS`** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors: • **RFE_DONTKNOW**: Undefined error. • **RFE_IDEXCEED**: Item id length too long/buffer too small. • **RFE_INVDPTR**: Bad D/pointer. • **RFE_INVPARAM**: Invalid parameters to function. • **RFE_NOACCESS**: No access. • **RFE_NOACCOUNT**: Not logged on. • **RFE_NOFILE**: File does not exist.	

See also **`RlcMakeList`**, **`RlcSelect`**.

7.4 RlcGetList

Purpose	Opens a previously saved list.
Synopsis	int **`RlcGetList`**(*FileName*, *ListName, ListHandle)*

	char * *FileName*; **char *** *ListName*; **RLC_LIST *** *ListHandle*;	
Parameters	*FileName*	A pointer to a string containing the name of the file that contains the list. If *Filename* is a null pointer, the list is deleted from the POINTER-FILE.
	ListName	A pointer to a string containing the name of the list.
	ListHandle	A pointer to a variable in which to return the handle of the open list.
Return value	The **`RlcGetList`** function returns **`SUCCESS`** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors: • **RFE_DONTKNOW**: Undefined error. • **RFE_IDEXCEED**: Item id length too long/buffer too small. • **RFE_INVDPTR**: Bad D/pointer. • **RFE_INVPARAM**: Invalid parameters to function. • **RFE_NOACCESS**: No access. • **RFE_NOACCOUNT**: Not logged on. • **RFE_NOFILE**: File does not exist. • **RFE_NOSPACE:** Unable to allocate more memory.	

See also **`RlcMakeList`**, **`RlcSaveList`**, **`RlcSelect`**.

7.5 RlcGetMultiValues

Purpose	Gets the value and sub-value numbers for the current element.	
Synopsis	void **`RlcGetMultiValues`**(*ListHandle, ValNum, SubValNum)* **RLC_LIST** *ListHandle*, **Int*** *ValNum*, **Int*** *SubValNum*	
Parameters	*ListHandle*	A pointer to a variable in which the current sub-value number will be returned.
	ValNum	A pointer to a variable in which the current value number will be returned.
	SubValNum	A pointer to a variable in which the current sub-value number will be returned.
Remarks	**`RlcGetMultiValues`** gets the numbers of the value and sub-value associated with the current element. This function should normally be used when the element is from an exploding index. If used with a non-exploding index, both *ValNum* and *SubValNum* are returned set to 1.	

7.6 RlcLockReadNextItem

Purpose	Obtains the next item-id from the specified list. It then locks the corresponding item in the specified file and returns the contents of that item.	
Synopsis	int **`RlcLockReadnextItem`**(*ListHandle, FileHandle, ItemId, ItemIdLen, Item, ItemMaxLen, ItemLen)* **RLC_LIST** *ListHandle*; **RFC_FILE** *FileHandle*; **char *** *ItemId*; **int *** *ItemIdLen*; **char *** *Item*; **int** *ItemMaxLen*; **int *** *ItemLen*;	
Parameters	*ListHandle*	The handle of an open list, returned by **`RlcGetList`**, **`RlcMakeList`** or **`RlcSelect`**.
	FileHandle	The handle of the required Reality file, returned by **`RfcOpenFile`**.
	ItemId	A pointer to a buffer (at least **RFE_MAX_ID_SIZE** bytes in length) in which the item-id will be returned.
	ItemIdLen	A pointer to a variable in which the length of the item-id will be returned.
	Item	A pointer to a buffer in which the item data will be returned.
	ItemMaxLen	The length of the *Item* buffer.
	ItemLen	A pointer to a variable in which the length of the item data will be returned. If the complete item was too long to fit into the buffer, this variable will be returned set to the total length of the item if known, or to zero.
Return value	The **`RlcLockReadNextItem`** function returns **`SUCCESS`** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors: • **RFE_DONTKNOW**: Undefined error. • **RFE_LOCKED**: The item is locked. • **RFE_NOITEM**: Item does not exist. • **RFE_READEXCEED**: The item is longer than the *Item* buffer – the data has been truncated. Use the **`RfcReadRest`** function to read the remainder of the item. • **RLE_ENDOFLIST**: The end of the list has been reached –the list is no longer available. • **RFE_NOFILE**: Unable to open scratch file. • **RFE_NOSPACE**: Unable to allocate more memory. • **RLE_RESIZEBUFF**: Item-id buffer too small.	
Remarks	This function is identical to **`RlcReadNextItem`**, except that the item is locked first. The operation of **`RlcLockReadNextItem`** depends on the	

	flags set with the **RfcSetLockMode** function. • If the lock mode has not been set, or is set to **RFC_OPT_NONE**, **RlcLockReadNextItem** will wait for a locked item to be released and will not lock a non-existent item. • If the **RFC_OPT_NO_WAIT** option is set, if the item is locked, **RlcLockReadNextItem** will return immediately with the error **RFE_LOCKED**. • If the **RFC_OPT_HOLD** option is set and the item does not exist, **RlcLockReadNextItem** will set an item lock. If the length of the item-id is greater than **RFE_MAX_ID_SIZE**, the error **RLE_RESIZEBUFF** is returned.

See also **RlcNext, RlcReadNextItem.**

7.7 RlcMakeList

Purpose	Constructs a list of item-ids from an open file.	
Synopsis	int **RlcMakeList**(*FileHandle, ListHandle)* **RFC_FILE** *FileHandle*; **RLC_LIST *** *ListHandle*;	
Parameters	*FileHandle*	The handle of the required Reality file, returned by **RfcOpenFile.**
	ListHandle	A pointer to a variable in which to return the handle of the open list.
Return value	The **RlcMakeList** function returns **SUCCESS** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors: • **RFE_NOFILE**: Unable to open scratch file. • **RFE_NOSPACE**: Unable to allocate more memory.	
Remarks	The file must remain open while reading the list – closing the file before closing the list will cause subsequent list accesses to fail. If the length of the item-id is greater than **RFE_MAX_ID_SIZE**, the error **RLE_RESIZEBUFF** is returned.	

See also **RlcGetList, RlcSaveList, RlcSelect.**

7.8 RlcNext

Purpose	Reads the next item-id from an open list.
Synopsis	int **RlcNext**(*ListHandle, Element, ElementMaxLen, ElementLen)*

	RLC_LIST *ListHandle*; **char *** *Element*; **int** *ElementMaxLen*; **int *** *ElementLen*;	
Parameters	*ListHandle*	The handle of an open list, returned by **`RlcGetList`**, **`RlcMakeList`** or **`RlcSelect`**.
	Element	A pointer to a buffer in which the item-id will be returned.
	ElementMaxLen	The length of the *Element* buffer.
	ElementLen	A pointer to a variable in which the length of the item-id will be returned.
Return value	The **`RlcNext`** function returns **`SUCCESS`** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors: • **RLE_ENDOFLIST**: The end of the list has been reached – the list is no longer available. • **RLE_INVALIDOP**: Invalid operation on this list (list not opened in elemental mode).	
Remarks	If the item-id is too long to fit in the *Element* buffer, it will be truncated. If *ListHandle* points to a list that was generated from an exploding index, **`RlcNext`** will repeatedly return the same item ID for each multi- and/or sub-value.	

See also **`RlcLockReadNextItem`**, **`RlcReadNextItem`**, **`RiscGetMultiValues`**.

7.9 RlcReadNextItem

Purpose	Obtains the next item-id from the specified list. It then reads the corresponding item in the specified file and returns the contents of that item.	
Synopsis	int **`RlcReadNextItem`**(*ListHandle, FileHandle, ItemId, ItemIdLen, Item, ItemMaxLen, ItemLen)* **RLC_LIST** *ListHandle*; **RFC_FILE** *FileHandle*; **char *** *ItemId*; **int *** *ItemIdLen*; **char *** *Item*; **int** *ItemMaxLen*; **int *** *ItemLen*;	
Parameters	*ListHandle*	The handle of an open list, returned by **`RlcGetList`**, **`RlcMakeList`** or **`RlcSelect`**.

	FileHandle	The handle of the required Reality file, returned by **`RfcOpenFile`**.
	ItemId	A pointer to a buffer in which the item-id will be returned.
	ItemIdLen	A pointer to a variable in which the length of the item-id will be returned.
	Item	A pointer to a buffer in which the item data will be returned.
	ItemMaxLen	The length of the *Item* buffer.
	ItemLen	A pointer to a variable in which the length of the item data will be returned. If the complete item was too long to fit into the buffer, this variable will be returned set to the total length of the item if known, or to zero.
Return value	The **`RlcReadNextItem`** function returns **`SUCCESS`** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors: • **RFE_DONTKNOW**: Undefined error. • **RFE_NOITEM**: Item does not exist. • **RFE_READEXCEED**: The item is longer than the *Item* buffer – the data has been truncated. Use the **`RfcReadRest`** function to read the remainder of the item. • **RLE_ENDOFLIST**: The end of the list has been reached – the list is no longer available. • **RLE_NOFILE**: Unable to open scratch file. • **RLE_NOSPACE**: Unable to allocate more memory. • **RLE_RESIZEBUFF**: Item-id buffer too small.	
Remarks	If the length of the item-id is greater than **RFE_MAX_ID_SIZE**, the error **RLE_RESIZEBUFF** is returned. If *ListHandle* points to a list that was generated from an exploding index, **`RlcNext`** will repeatedly return the same item ID for each multi- and/or sub-value.	

See also **`RlcLockReadNextItem`**, **`RlcReadNextItem`**, **`RiscGetMultiValues`**.

7.10 RlcSaveList

Purpose	Save an open list to a file item.	
Synopsis	int **`RlcSaveList`**(*ListHandle, FileHandle, ListName)* **RLC_LIST** *ListHandle*; **char *** *FileName*; **char *** *ListName*;	
Parameters	*ListHandle*	The handle of an open list, returned by **`RlcGetList`**, **`RlcMakeList`** or **`RlcSelect`**.

	FileName	A pointer to a string containing the name of the file in which to save the list. If *Filename* is a null pointer, the list is saved in the POINTER-FILE.
	ListName	A pointer to a string containing the name of the item in which to save the list. If the item already exists, it is overwritten.
Return value	The **`RlcSaveList`** function returns **`SUCCESS`** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors: • **RFE_DONTKNOW**: Undefined error. • **RFE_IDEXCEED**: Item id length too long/buffer too small. • **RFE_INVDPTR**: Bad D/pointer. • **RFE_INVPARAM**: Invalid parameters to function. • **RFE_NOACCESS**: No access. • **RFE_NOACCOUNT**: Not logged on. • **RFE_NOFILE**: File not found. • **RLE_ENDOFLIST**: The end of the list has been reached –the list is no longer available. • **RLE_NOSPACE**: Unable to allocate more memory.	
Remarks	On completion, **`RlcSaveList`** closes the list. It can be reopened with the **`RlcGetList`** function. The list will be saved starting from the current item-id. If the list has been partially read, therefore, only the unread portion is saved.	

See also **`RlcDeleteList`**, **`RlcGetList`**, **`RlcLockReadNextItem`**, **`RlcMakeList`**, **`RlcNext`**, **`RlcReadNextItem`**, **`RlcSelect`**.

7.11 RlcSelect

Purpose	Creates a list of item-ids selected from a Reality file.	
Synopsis	int **`RlcSelect`**(*QueryType*, *Filename, Criteria*, *ListHandle)* **RLC_QUERY_TYPE** *QueryType*; **char *** *FileName*; **char *** *Criteria*; **RLC_LIST *** *ListHandle*;	
Parameters	*QueryType*	The type of Query. This must be one of the following: • **RLC_QT_ENGLISH_SELECT**: Select only. • **RLC_QT_ENGLISH_SSELECT**: Select and sort.
	FileName	A pointer to a string containing the name of the file from which to make the selection.
	Criteria	Points to a string containing the select criteria. These must be in the

		same form as in an English command line; for example, `WITH AGE < "30"` will select all items in which the attribute AGE is less than 30. Refer to the *English Programming Reference Manual* for full details. If there are no select criteria, Criteria must point to a null string.
	ListHandle	A pointer to a variable in which to return the handle of the new list.
Return value	The **`RlcSelect`** function returns **`SUCCESS`** for successful completion, or one of the return codes listed in Appendix A. The following are likely errors: • **RFE_DONTKNOW**: Undefined error. • **RFE_INVDPTR**: Bad D/pointer. • **RFE_INVPARAM**: Invalid parameters to function. • **RFE_NOACCESS**: No access. • **RFE_NOFILE**: File not found. • **RLE_NOSPACE**: Unable to allocate more memory.	

See also **`RlcMakeList`**, **`RlcGetList`**, **`RlcSaveList`**.

Section 8: Appendix A – Error Return Codes

This appendix lists the values returned by the Reality C API functions. The constants defined in the header files are listed in alphabetical order.

8.1 Introduction

Most Reality Interface functions return an integer that is numeric return code. This return code will have a value of zero if the function call is successful. If the function call is unsuccessful, the return code will have a non-zero value.

Return code definitions are #defined in the header files,

- ros/rfc.h
- ros/rgc.h
- ros/rlc.h
- ros/rcc.h

These can be included as needed in user-written C programs which use the Reality Interface functions.

Note

To allow these files to be included, the following should be added to the complier's include path:
Windows systems: `%REALROOT%\include`
UNIX systems: `/usr/include`.

If a function call results in a non-zero return code, an associated textual error message may be displayed by the program.

Textual messages associated with Interactive File Access and Interprocess Communication return codes can be displayed using the **`RgcErrMsg`** function. The **`RgcErrMsg`** function is passed a return code, which it uses as an index to the error message file, and a pointer to a buffer. **`RgcErrMsg`** extracts the textual error message and places it in the buffer.

8.1.1 Example

In the example below the if clause is executed if **`RetCode`** does not equal **`SUCCESS`**. In these circumstances, **`RgcErrMsg`** is called to read the associated error message into the supplied buffer, ErrorString. The **`printf`** function displays the contents of the buffer.

```
if ((RetCode = RfcOpenFile(FileName,&FileHandle) != SUCCESS))
{
ErrorString = RgcErrMsg(RetCode);
(void) printf("%s\n", ErrorString);
exit (2);
}
```

Textual messages associated with InterProcess Communication function return codes can be displayed using the **`RccError`** function (if you are using Interactive File Access as well, however, you must use **`RgcErrMsg`**).

8.1.2 List of error definitions

The following list is of Reality errors:

Definition	Meaning
EACCOUNT	Invalid Account/Password on Remote System
ECONREF	Connection Refused by Remote System (unspecified reason)
ELENGTH	Qualifier/Data Length too long on Remote Systems
ENOCHAN	No Channels Available on Remote System
ENOPRC	No Process Available on Remote System
ENOVP	No Virtual Ports Available on Remote System
EPROTOCOL	Protocol Error
EPROTSUP	Protocol Not Supported on Remote System
ESERVER	Invalid Server Name on Remote System
ETIMEOUT	Timeout Error on Remote System

The following is a list of Reality Communications Interface errors:

Definition	Meaning
RCE_ACCESS	System error: Access
RCE_ACI_ENTRY	The ROUTE-FILE entry is for ACI connections only
RCE_ADDR_FORMAT	Invalid Address Format in ROUTE-FILE
RCE_CHARMODE_NOT_SUPP	System Error: Character Mode Circuit Not Supported
RCE_CIRCUIT_ABORT	Remote: Circuit Aborted
RCE_CLIENT_NOT_TO	Client Request has not Timed Out
RCE_COMMAND	Remote: Illegal Command
RCE_CONEXCEED	Exceeded maximum number of connections
RCE_CONNECTION_REFUSED	Connection refused by remote system
RCE_CONNECTION_REJ	Connection rejected by remote server
RCE_CREATE_IPCQ	Create IPC Message Queue Failure

Definition	Meaning
RCE_CWD_NOT_FOUND	Cannot Find Current Working Directory
RCE_DDA_ACI_REPLUG	Remote: DDA to ACI Replug Not Supported
RCE_DELETE_IPCQ	Delete IPC Message Queue Failure
RCE_DETACH_FAIL	System Error: Session Manager Detach Fail
RCE_ENCPLID_LENGTH	PLId environment variable too long
RCE_ENVIRON	System error: Get Environment
RCE_ERRMSG_FILE	Cannot Open ERRMSG-FILE
RCE_ERRMSG_LOCATE	Cannot locate error message in ERRMSG-FILE
RCE_ERRMSG_READ	Cannot Read Error Message From ERRMSGFILE
RCE_ERRNUM_READ	Cannot Read Error Number From ERRMSG-FILE
RCE_EVENT_LOG_OPEN	Failed to Open Session Manager Event Log File
RCE_EXEC	System Error: Exec
RCE_EXPEDITED	Expedited Data Received
RCE_FCNTL	System Error: File Control
RCE_FORK	System Error: Fork
RCE_FREE_SYSCON	System Connection Release Error
RCE_FSTAT	System Error: File Stats
RCE_ILLSREF	Illegal Session Reference
RCE_INCOMPAT_IPC_MSG	Incompatible rcs library and session manager
RCE_INSUFFMEM	System Error: Insufficient Memory
RCE_INTERRUPT	Interrupted System Call
RCE_INV_SMANAGERQ	Invalid SMANAGERQ Environment Variable
RCE_INVALID_DBASE	Invalid Database Name

Definition	Meaning
RCE_INVALID_FORMAT	Invalid Session Message Format
RCE_INVALID_INT_ACTION	Invalid Interrupt Action
RCE_INVALID_PROTOCOL	Invalid Session Connect Protocol
RCE_INVALID_SCONNECT	Invalid Session Connect String
RCE_INVALID_TIMEOUT	Invalid Timeout
RCE_INVARG	Invalid argument
RCE_INVDBUF	Invalid Data Buffer
RCE_IOCTL	System Error: IO Control
RCE_KILL	System Error: Kill
RCE_LENGTH_OVFL	Qualifier or Data Length Overflow
RCE_MEMCPY	System Error: Memory Copy
RCE_MOREDATA	More Data Available
RCE_NO_NET	Network Option not Installed
RCE_NO_PLID	No PLId saved and not requested to generate a PLId
RCE_NO_PROCESS	Remote: No Process Available On Remote System
RCE_NO_RCS_MANAGER	Session Manager Not Running
RCE_NO_ROUTEFILE	Cannot open ROUTE-FILE
RCE_NO_SYSCON	System Connections not licenced
RCE_NO_USERSFILE	Cannot Open USERS-FILE
RCE_NODATA	Data Not Yet Available
RCE_PATHEXCEED	Path too long for supplied buffer
RCE_PLID	Invalid Physical Location Identifier
RCE_PLID_LENGTH	PLId too long for supplied buffer
RCE_PLID_NULL	PLId is null string

Definition	Meaning
RCE_POLL	System Error: Poll
RCE_PROTOCOL	Remote: Protocol Violation
RCE_PROTOCOL_NOTSUPP	Remote: Protocol Not Supported
RCE_PSW	Invalid Password
RCE_QUAL_DATA_OVFL	Remote: Qualifier or Data Overflow
RCE_QUALOVFL	Qualifier Overflow
RCE_QUALTRUNC	Qualifier Truncation
RCE_QUALTRUNC_EXPEDITED	Qualifier Truncation and Expedited Data Received
RCE_QUALTRUNC_MOREDATA	Qualifier Truncation and More Data Available
RCE_RCV_IPC_MSG	Receive IPC Message Failure
RCE_Reality_SERVER	Start Reality Server Failure
RCE_REALPLID	Cannot set REALPLId environment variable
RCE_REMOTE	Entry for remote system
RCE_ROUTEFILE	Invalid ROUTE_FILE format
RCE_SERVER	Invalid Server Name
RCE_SERVER_NOT_EXEC	Invalid server name: not EXECUTABLE.
RCE_SERVER_NOT_TO	Server Request has not Timed Out
RCE_SESSION_LOG_OPEN	Failed to Open Session Manager Session Log File
RCE_SIGNAL	System Error: Signal
RCE_SND_IPC_MSG	Send IPC Message Failure
RCE_SYSCON_EXCEED	System Connection Limit Exceeded
RCE_SYSTEM	Invalid System Name
RCE_TACCEPT	Transport: Accept Connection Failure

Definition	Meaning
RCE_TALLOC	Transport: Allocation Failure
RCE_TBIND	Transport: Address Bind Failure
RCE_TCLOSE	Transport: Close Device Failure
RCE_TCONNECT	Transport: Connection Refused
RCE_TDEQUEUE	Service Request Timer Dequeue Failure
RCE_TENDPT	Transport: Exceeded maximum listening endpoints
RCE_TEVENT	Transport: Unexpected Event Received on Listening Endpoint
RCE_TEVENT_CANCEL	Transport: connection event cancelled by disconnection event
RCE_THOSTDISC	Transport: Circuit Disconnected
RCE_TIMEOUT	Operation Timed Out
RCE_TLISTEN	Transport: Listen for Connection Failure
RCE_TLOOK	Transport: State Enquiry (Look) Failure
RCE_TO_COMPLETION	Service Request Completion Routine Failure
RCE_TOPENDEV	Transport: Open Device Failure
RCE_TPEXPOVFL	Expedited Data Overflow
RCE_TRACE_LOG_OPEN	Failed to Open Session Manager Trace Log File
RCE_TRCV	Transport: Receive Failure
RCE_TRCVDIS	Transport: Receive Disconnect Failure
RCE_TRCVREL	Transport: Receive Orderly Release Failure
RCE_TSND	Transport: Send Failure
RCE_TSNDDIS	Transport: Send Disconnect Failure
RCE_TSNDREL	Transport: Send Orderly Release Failure
RCE_TSYNC	Transport: Process Synchronisation Failure
RCE_UNKNOWN_MSGTYPE	Unknown IPC Message Type Received
RCE_USERID	Invalid Userid/Password
RCE_USERSFILE	Invalid USERS-FILE Format
RCE_WAITEVENT	System Error: Wait for Event

Definition	Meaning
RCSE_GETUCB	System error: insufficient memory.

The following is a list of Reality Filing Interface errors:

Definition	Meaning
RFE_ACCTACTIVE	Account handle has not been saved
RFE_DELETED	Deleted item — BS private
RFE_DISKFULL	File system full
RFE_DONTKNOW	Undefined error
RFE_EOF	End of file reached
RFE_EXDBASE	Not allowed across databases
RFE_FILMAX	Max number of files already open
RFE_FLSOPEN	File still open
RFE_GFE	Group Format Error
RFE_IDEXCEED	Item id length too long/buffer too small
RFE_IEXISTS	Item already exists, no overwrite allowed
RFE_INVACCPASS	Invalid logon attempt
RFE_INVALID	Invalid database name
RFE_INVDBASEDIR	Bad directory for database
RFE_INVDPTR	Bad D/pointer
RFE_INVEVENT	Bad call to event handler
RFE_INVLEVEL	Invalid file level
RFE_INVNAME	Bad item name
RFE_INVOFFSET	Invalid offset
RFE_INVPARAM	Invalid parameters to function
RFE_INVUPDATE	Invalid D/pointer update
RFE_LOCKCLEARED	Lock found and cleared
RFE_LOCKED	Lock is taken
RFE_NOACCESS	No access

Definition	Meaning
RFE_NOACCOUNT	Not logged on
RFE_NOATTR	Attribute does not exist
RFE_NODATABASE	Not connected to a database
RFE_NODEL	Delete failed
RFE_NODICT	No DICT for DATA
RFE_NOFILE	File does not exist
RFE_NOHANDLE	Handle not valid
RFE_NOILOCKS	Item lock table full
RFE_NOITEM	Item does not exist
RFE_NOLOCK	Lock does not exist
RFE_NONAME	Name not supplied
RFE_NONUNIQUE	Non-unique name
RFE_NOREAD	No outstanding read
RFE_NOSECT	File section does not exist
RFE_NOSEQACCESS	No RfsSetupSeqAccess called
RFE_NOSPACE	Unable to allocate more memory
RFE_NOSUPPORT	Operation not supported
RFE_NOTOPEN	File not open on this reference
RFE_OPENMODE	Inconsistent with file open mode
RFE_PRIV	Insufficient privilege level
RFE_READEXCEED	Read too big for buffer
RFE_READONLY	File is read-only
RFE_REMOTE	Remote database
RFE_RETPRO	Retrieval lock set
RFE_REUSE	Handle being reused
RFE_SECTEXISTS	File section already exists
RFE_TOOBIG	File or item is too big
RFE_UPDPRO	File is update protected

The following is a list of Reality General Services Interface errors:

Definition	Meaning
RGE_ABORT	ABORT
RGE_BAD_CB	Invalid control block
RGE_BAD_MODULE	Bad module number
RGE_BAD_MSG	Bad message received
RGE_BUFFER_TOO_SMALL	Buffer too small for operation
RGE_DUMPED	Core Dumped
RGE_DUMPSUPP	Core Dump Suppressed by REALDUMP=0
RGE_ENDMSG	Located mark is a segment mark
RGE_ENOEXTRACT	Could not extract the attribute from the string
RGE_LAYER_OVFL	Vector table overflow
RGE_MALLOC	Cannot allocate memory
RGE_MIDMSG	Located mark isn't segment mark
RGE_NO_MSG_BUF	No buffer for received message
RGE_NOATTR	Attribute does not exist
RGE_NODELETE	Mark being deleted does not exist
RGE_NODUMP	Core Dump Failed
RGE_NOHANDLE	Invalid Database Handle
RGE_NOMARK	Mark does not exist
RGE_NOPRESENTRY	No Process Resource Table Entry
RGE_NOSHMHANDLE	Invalid shared memory address
RGE_NOSPACE	Allocated buffer too small
RGE_NOT_SUPPORTED	Operation not supported
RGE_NOTNUM	String does not convert to a number
RGE_NOVALUE	Value does not exist
RGE_RUNNING	Running
RGE_SERVICE_TABLE_FULL	Notification service table full

The following is a list of C-ISAM Indexed Access Layer errors:

Definition	Meaning
RIXE_EOL	At beginning or end of index
RIXE_KEY_TOO_BIG	Key too long for *KeyBuff*
RIXE_NO_INDEX	Index specified does not exist
RIXE_NOT_FOUND	No such key

The following is a list of Reality List Services Interface errors:

Definition	Meaning
RLE_ENDOFLIST	Reached end of list
RLE_FAILURE	Unknown error occurred
RLE_INVALIDOP	Invalid operation on this list
RLE_NO_CB	No allocation of control block
RLE_NOCLOSE	Could not close list file
RLE_NOFILE	Scratch file open failed
RLE_NOLIST	List does not exist
RLE_NOPOINTERFL	No pointer file
RLE_NOSPACE	Unable to allocate more memory
RLE_OSERROR	Unexpected error in underlying OS
RLE_READEXHAUST	Reached end of list buffer
RLE_RESIZEBUFF	Buffer too small
RLE_SELECT_CRI	Selection criteria error

The following is a list of other completion codes:

Definition	Meaning
SUCCESS	Function completed successfully

Section 9: Appendix B – Connecting to multiple databases

This appendix describes how to make connections to multiple Reality databases using the Rfc and Risc interfaces.

9.1 Overview

The Reality Filing Services Interface (Rfc), described in Section 4, enables a C program to connect to a Reality database to create, delete, read from and write to Reality files. The Reality Indexed Access Interface, described in Section 6, provides the same facilities, but works with records and keys, handling the Reality item-id as part of the record data.

When using either interface, before connecting to a database, the program must first call the **RgcStartUpServices** macro to perform one-time initialisation operations.

If the program is then to make multiple connections to Reality databases, the first connection must be a dummy outer connection, made via **RfcConnect**. This connection must be kept open until all subsequent connections have been closed, that is the number of open connections must always be at least one.

The second connection to a database is a real connection: the **RfcConnect** or **RiscConnect** statement is followed by **RfcOpenFile** or **RiscOpen** and the program can then manipulate data in the specified database file.

When the final real connection has been closed using **RfcDisconnect** or **RiscDisconnect**, the outer dummy connection can be closed using **RfcDisconnect**. The program must then call the **RgcShutDownServices** macro to close all active services.

9.2 Example

1. A C program that is to make connections to Reality databases must first call **RgcStartUpServices**:

```
#define MAX_NAME_LEN 30
#define MAX_PASSWORD_LEN 10
char DatabaseName[MAX_NAME_LEN+1] = "dbase1";
char UserName[MAX_NAME_LEN+1] = "user1";
char UserPassword[MAX_PASSWORD_LEN+1] = "upswd1";
char AccountName[MAX_NAME_LEN+1] = "account1";
char AccountPassword[MAX_PASSWORD_LEN+1] = "apswd1";
char DatabaseFilename[MAX_NAME_LEN+1] = "file1";
int nResult = 0;
RFC_ACCOUNT ExtraAccountHandle = NULL;
RISC_FILE FileHandle = NULL;
RgcStartUpServices(nResult);
```

2. The program must now make the outer, dummy database connection. **RfcGetAccount** is used to store the account handle for this session:

```
nResult = RfcConnect(DatabaseName,
UserName,
UserPassword,
AccountName,
AccountPassword);
nResult = RfcGetAccount(&ExtraAccountHandle);
```

3. The program can now make a real database connection:

```
nResult = RiscConnect(DatabaseName,
UserName,
UserPassword,
AccountName,
AccountPassword);
```

4. The required database file is opened:

```
nResult = RiscOpen(DatabaseFileName, &FileHandle);
```

5. When the program has completed the required operations on the data in the specified file, the file can be closed:

```
nResult = RiscClose(FileHandle);
```

6. When work on the files in this database is complete, the connection to the database is closed:

```
nResult = RiscDisconnect();
```

   Steps 3 to 6 can now be repeated as many times as is necessary, carrying out work on any number of files in any number of Reality databases. Where connection is via the Rfc Interface, **`RfcGetAccount`** and **`RfcSetAccount`** can be used to maintain concurrent connections to two or more databases.

7. When step 6 has been completed for the final time – that is, when all the real database connections are closed - the outer, dummy database connection is closed:

```
nResult = RfcSetAccount(ExtraAccountHandle);
nResult = RfcDisconnect();
```

8. Finally, the program must call **`RgcShutDownServices`** to close all active services:

```
RgcShutDownServices();
```

Section 10: Appendix C – Example programs

This appendix contains four example C programs. The first uses the Rfc functions to access a Reality file, while the second and third are a client and server program using the Rcc functions to communicate with each other. The fourth illustrates the use of the Risc interface in a multi-threaded environment.

10.1 File access

The following is an example C program which uses the Rfc functions.

This program is delivered with the UNIX-Connect product. It is held in the file **/usr/RCS/examples**, and can be run as follows:

```
$ cd /usr/RCS/examples
$ make
$ ifa_eg
```

The program reads data from a specified item within a specified file on a Reality system.

```
/* This program reads an item of unknown length from a specified file in
a Reality account */
#include <stdio.h>
#include <ros/rfc.h>
#include <ros/rgc.h>
#define BUFSIZE 100
char DatabaseName[51]; /* name of the reality database */
char User[51]; /* user id on the database */
char UserPasswd[] = ""; /* user password */
char Account[51]; /* account name on the database */
char AcctPasswd[] = ""; /* account password */
char FileName[51]; /* name of file containing item to read */
char ItemId[99]; /* name of the item to be read */
int ItemIdLen; /* length of above item-id */
main()
{
RFC_FILE FileHandle; /* contains file handle to opened file */
char Item [256]; /* buffer used to store the item */
int ItemMaxLen = 256; /* length of above buffer */
char *ErrorString; /* pointer to error message text */
int i, RetCode, ItemLen, DataLen, Size;
/* start up services */
RgcStartUpServices (RetCode);
/* connect to the database and log on under the specified user id
to the named account */
/* request the database (system) to connect to */
printf("\n\n");
printf("Type in database to connect to ? ");
fgets(DatabaseName, BUFSIZE, stdin);
/* discard <CR> from last character of string */
Size = strlen(DatabaseName);
DatabaseName[Size-1] = NULL;
/* request the user-id to connect to */
printf("\n\n");
printf("Type in Userid to connect to ? ");
fgets(User, BUFSIZE, stdin);
/* discard <CR> from last character of string */
Size = strlen(User);
User[Size-1] = NULL;
/* request the account to connect to */
printf("\n\n");
```

```
printf("Type in account to connect to ? ");
fgets(Account, BUFSIZE, stdin);
/* discard <CR> from last character of string */
Size = strlen(Account);
Account[Size-1] = NULL;
/* request name of file to be read */
printf("\n\n");
printf("Type in name of file to be read ? ");
fgets(FileName, BUFSIZE, stdin);
/* discard <CR> from last character of string */
Size = strlen(FileName);
FileName[Size-1] = NULL;
/* request item-id to be read */
printf("\n\n");
printf("Type in Itemid to be read ? ");
fgets(ItemId, BUFSIZE, stdin);
/* discard <CR> from last character of string */
Size = strlen(ItemId);
ItemId[Size-1] = NULL;
ItemIdLen = strlen(ItemId);
/* connect to server */
printf("\n\nConnecting... \n");
if ((RetCode = RfcConnect(DatabaseName,
User,
UserPasswd,
Account,
AcctPasswd)) != SUCCESS)
{
/* if connect fails print error message and return */
ErrorString = RgcErrMsg (RetCode);
printf("%s\n", ErrorString);
exit(1);
}
/* open the file containing the item to be read */
if ((RetCode = RfcOpenFile(FileName, &FileHandle)) != SUCCESS)
{
/* if open fails print error message and return */
ErrorString = RgcErrMsg (RetCode);
printf("%s\n", ErrorString);
exit(2);
}
/* read the item */
printf("\nReading ...\n\n");
RetCode =
RfcRead(FileHandle,ItemId,ItemIdLen,Item,ItemMaxLen,&ItemLen);
if (RetCode != SUCCESS && RetCode != RFE_READEXCEED)
{
ErrorString = RgcErrMsg (RetCode);
printf("%s\n", ErrorString);
exit(3);
}
else if (RetCode == RFE_READEXCEED)
{
/* if the buffer used to store the item is full and there is still
more data to read print out the data in the buffer */
for (i=0; i<ItemMaxLen ; i++)
{
if (isascii(Item[i]))
printf("%c", Item[i]);
else
printf("\n"); /* Assume an attribute mark */
}
/* read the next batch of data until all has been read */
while (RetCode == RFE_READEXCEED)
{
RetCode = RfcReadRest(FileHandle, Item, ItemMaxLen, &DataLen);
```

```
if (RetCode != SUCCESS && RetCode != RFE_READEXCEED)
{
/* if read rest fails print error message and return */
ErrorString = RgcErrMsg(RetCode);
printf("%s\n", ErrorString);
exit(4);
}
else
{
for (i=0; i<DataLen ; i++)
{
if (isascii(Item[i]))
printf("%c", Item[i]);
else
printf("\n"); /* Assume an attribute mark */
}
printf("\n");
}
}
}
else
{
/* if buffer not full and no more data to be read */
for (i=0; i<ItemLen ; i++)
{
if (isascii(Item[i]))
printf("%c", Item[i]);
else
printf("\n"); /* Assume an attribute mark */
}
}
/* close the file containing the item */
if ((RetCode = RfcClose(FileHandle)) != SUCCESS)
{
/* if close fails print error message and return */
ErrorString = RgcErrMsg(RetCode);
printf("%s\n", ErrorString);
exit(5);
}
/* disconnect from the database and log off */
printf("\nDisconnecting ...\n\n");
if ((RetCode = RfcDisconnect()) != SUCCESS)
{
/* if disconnect fails print error message and return */
ErrorString = RgcErrMsg(RetCode);
printf("%s\n", ErrorString);
exit(6);
}
RgcShutDownServices();
exit(0);
}
```

10.2 Client and server

Two example programs are delivered with the UNIX-Connect product. They are held in the **usr/RCS/examples** file on the UNIX environment.

The programs are a client and server which run back-to-back on the UNIX environment: the client program prompts the user for an environment to connect to and having made the connection, reads in a command typed on the user's keyboard and sends it to the server. The server program executes the UNIX command and transmits the response back to the client.

To run the client program, enter the following commands:

```
$ cd /usr/RCS/examples
$ make
$ client
```

The following prompts are displayed:

```
Type in system to connect to ?
Type in userid to connect to ?
Type in account to connect to ?
Type in server to connect to ?
Timeout ?
```

Terms	Definition
system	Enter the system name of the listening entry with a network type of local loopback in the ROUTE-FILE (see *UNIX-Connect System Administration Guide*).
user-id	Enter a user-id which is valid on the local UNIX system. If the user-id has a password this must be entered, separated from the user-id by a comma.
account	Press *Return*.
server	Enter the name of the server program, which for this example program is **/usr/RCS/examples/server**.
timeout	Enter a value between 1 and 255 which indicates (in minutes) the time within which the connection must be made.

Once the connection is established the prompt `Type in your command` is displayed. Any UNIX command can be entered. To terminate the programs, enter **exit**.

10.2.1 Client

The client program is displayed as follows:

```
/*
* Example UNIX-Connect client program
*
* Uses the Rcc API
*
* It can make a connect to a Reality DataBasic server program
* or a UNIX-Connect server program.
*
* It sends the supplied command to the server and displays the output.
* It deliberately receives the returned data in two chunks.
*
*/
#include <stdio.h>
#include <ros/rcc.h>
#define BUFSIZE 80
char ExitStr[] = "exit";
main()
{
```

```
int RetCode, Size;
RCS_SREF Reference;
RCS_MCB Msg;
unsigned char SndBuf [BUFSIZE];
unsigned char QualBuf [BUFSIZE];
unsigned char RcvBuf [20];
unsigned char *NewRcvBuf = NULL;
int LengthLeft;
int Timeout;
char SystemName [51];
char Userid [51];
char Account [51];
char Server [100];
char Tout [20];
char ErrorStr [100];
/* Request the system to connect to */
printf("\n\n");
printf("Type in system to connect to ? ");
fgets (SystemName, BUFSIZE, stdin);
/* discard <CR> from last character of string */
Size = strlen (SystemName);
SystemName [Size - 1] = NULL;
/* Request the user-id to connect to */
printf("\n\n");
printf("Type in Userid to connect to ? ");
fgets (Userid, BUFSIZE, stdin);
/* Discard <CR> from last character of string */
Size = strlen (Userid);
Userid [Size - 1] = NULL;
/* Request the account to be connected to */
printf("\n\n");
printf("Type in account to connect to ? ");
fgets (Account, BUFSIZE, stdin);
/* Discard <CR> from last character of string */
Size = strlen (Account);
Account [Size - 1] = NULL;
/* Request the server to be connected to */
printf("\n\n");
printf("Type in server to connect to ? ");
fgets (Server, BUFSIZE, stdin);
/* discard <CR> from last character of string */
Size = strlen (Server);
Server [Size - 1] = NULL;
/* Request the Timeout */
printf("\n\n");
printf("Connect Timeout in minutes ? ");
fgets (Tout, BUFSIZE, stdin);
if ((Timeout = atoi(Tout)) != 0)
{
printf("Setting the timeout to %d minutes\n", Timeout);
RccSetConnectOptions (0, Timeout);
}
/* Connect to server */
printf ("Connecting .... \n");
if ((RetCode = RccConnect (&Reference, SystemName, Userid, Account,
Server)) != SUCCESS)
{
RccError (RetCode, ErrorStr);
printf("RccConnect Error : %s\n", ErrorStr);
exit();
}
/* Initialise message structure */
Msg.Function = 0;
Msg.Reference = 0;
Msg.QualLength = 0;
Msg.DataLength = 0; /* initially */
```

```
Msg.QualBuffer = QualBuf;
Msg.DataBuffer = SndBuf;
Msg.MaxQualLength = sizeof (QualBuf);
Msg.MaxDataLength = sizeof (SndBuf);
fflush (stdout);
while (fgets (SndBuf, BUFSIZE, stdin))
{
printf("\n");
/* check for termination condition */
if (!strncmp (SndBuf, ExitStr, 4))
break;
/* discard <CR> from last character of string */
Size = strlen (SndBuf);
SndBuf [Size - 1] = NULL;
/* send off typed in message */
printf ("Sending message with data : %s.\n", SndBuf);
Msg.DataLength = strlen (SndBuf);
Msg.DataBuffer = SndBuf;
if (RetCode = RccSendMsg (Reference, &Msg)) != SUCCESS)
{
RccError (RetCode, ErrorStr);
printf("RccSendMsg Error : %s\n", ErrorStr);
exit();
}
/* receive response */
Msg.DataBuffer = RcvBuf; /* set up larger buffer */
Msg.DataLength = 0; /* initially */
/* prime MaxDataLength with maximum size of buffer*/
Msg.MaxDataLength = sizeof(RcvBuf)-1;
printf ("Recwaiting message.\n");
/* must be able to cater for messages received which are
larger than the Message DataBuffer */
do
{
RetCode = RccRecWaitMsg (Reference, &Msg);
if (RetCode != SUCCESS)
if (RetCode != RCE_MOREDATA)
{
RccError (RetCode, ErrorStr);
printf ("RccRecWaitMsg Error : %s\n", ErrorStr);
exit();
}
/* print out results */
printf("The length of the data was %d.\n", Msg.DataLength);
Msg.DataBuffer[Msg.DataLength] = '\0';
printf("The data received was\n%s.\n", Msg.DataBuffer);
/* If we have more data then use NewRcvBuf */
if (RetCode == RCE_MOREDATA)
{
/* free off NewRcvBuf if necessary */
if (NewRcvBuf)
{
free (NewRcvBuf);
NewRcvBuf = NULL;
}
printf ("Length of data = %d.\n", Msg.MaxDataLength);
LengthLeft = Msg.MaxDataLength - Msg.DataLength;
printf ("The length left to read is %d.\n",
LengthLeft);
if ( !(NewRcvBuf = (unsigned char *) malloc (LengthLeft
+ 1)) )
{
printf ("Malloc() Failure\n");
exit();
}
/* Receive into new buffer */
```

```
Msg.DataBuffer = NewRcvBuf;
Msg.MaxDataLength = LengthLeft;
}
} while (RetCode == RCE_MOREDATA);
printf ("Type in your command : ");
fflush (stdout);
memset (SndBuf, '\0', sizeof (SndBuf));
}
/* disconnect the circuit */
printf ("Disconnecting ...\n\n");
if ((RetCode = RccDisconnect (Reference)) != SUCCESS)
{
RccError (RetCode, ErrorStr);
printf("RccDisconnect Error : %s\n", ErrorStr);
exit();
}
}
```

10.2.2 Server

The example server program is displayed as follows:

```
/*
* Example UNIX-Connect server program
*
* Uses the Rcc API
*
* It can receive connects from a Reality DataBasic client program.
* or a UNIX-Connect client program.
*
* It runs the supplied command and returns any output to the client.
*
* It demonstrates a server that performs an RccAccept until
* the RccAccept fails with a timeout. This is a useful design pattern
* for server programs, to avoid the delays associated with starting
* a program but prevent them hanging around forever. The timeout is
* set by the RccSetAcceptOptions().
*
* The first time this runs can be in response to an incoming client
connect.
*/
#include <stdio.h>
#include <fcntl.h>
#include <ros/rcc.h>
#define DATASIZE 0x100000
#define BUFSIZE 100
#define TRUE 1
#define FALSE 0
#define ZERO 0
char Server[] = "server";
main()
{
int RetCode, TraceLevel;
int Fd, Fd2;
RCS_SREF Reference;
RCS_MCB Msg;
unsigned char QualBuf[BUFSIZE];
unsigned char DataBuf[DATASIZE];
int AsynchMode;
char Buf [BUFSIZE];
int MaxBuf = BUFSIZE;
char Reply [10];
char Cmd [BUFSIZE];
FILE *Ptr, *popen();
char ErrorStr [100];
```

```
char ClientId [50], Plid [50];
/* initialise message structure */
Msg.Function = ZERO;
Msg.Reference = ZERO;
Msg.QualLength = ZERO;
Msg.DataLength = ZERO; /* initially */
Msg.QualBuffer = QualBuf;
Msg.DataBuffer = DataBuf;
Msg.MaxQualLength = ZERO;
Msg.MaxDataLength = DATASIZE;
printf ("\n\nDo you want to operate in Asynchronous Receive Mode ?
");
fgets (Reply, BUFSIZE, stdin);
if ((Reply[0] == 'y') || (Reply[0] == 'Y'))
AsynchMode = TRUE;
else
AsynchMode = FALSE;
/* Accept connection from client */
RccSetAcceptOptions (RCS_SECONDS, 40); /* Timeout = 40 seconds */
printf ("Accepting ....\n");
while ((RetCode = RccAccept (&Reference, "", Server, ClientId,
Plid)) == SUCCESS)
{
printf ("Connected to ClientId : %s from PLId %s\n",
ClientId,Plid);
while (1)
{
/* receive the command:
* Two ways are given for receiving data:-
* Asynchronously where the program has other work to do
* if no data has arrived. In this example we just sleep.
*
* Synchronous receive waits until data has arrrived or the
* circuit has disconnected
*/
if (AsynchMode)
{
while ((RetCode = RccReceiveMsg(Reference, &Msg)) ==
RCE_NODATA)
{
printf ("sleeping before polling for a message\n");
sleep (2);
}
if (RetCode != SUCCESS)
{
/* disconnect the circuit */
RccError (RetCode, ErrorStr);
printf("Receive failed: %s\nDisconnecting ...\n\n",
ErrorStr);
if ((RetCode = RccDisconnect (Reference))
!= SUCCESS)
{
RccError (RetCode, ErrorStr);
printf("Disconnect failed: %s\n", ErrorStr);
}
break;
}
/* Have some data */
}
else
{
/* Normal synchronous recwait() */
printf ("Recwaiting message.\n");
if ((RetCode = RccRecWaitMsg (Reference, &Msg))
!= SUCCESS)
{
```

```
RccError (RetCode, ErrorStr);
printf("Receive failed: %s\nDisconnecting ...
\n\n", ErrorStr);
if ((RetCode = RccDisconnect (Reference))
!= SUCCESS)
{
RccError (RetCode, ErrorStr);
printf("Disconnect failed: %s\n", ErrorStr);
}
break;
}
/* Have some data */
}
/* Data received, process it as a command */
strncpy (Cmd, Msg.DataBuffer, Msg.DataLength);
Cmd [Msg.DataLength] = NULL;
strcpy (Msg.DataBuffer, "");
printf("The Data Rcvd was : %s\n", Cmd);
/* Exec the command capturing data and send reply to client
*/
if ((Ptr = popen (Cmd, "r")) != NULL)
{
int ResponseLength;
for (ResponseLength=0;
((fgets(Buf, MaxBuf, Ptr)!=NULL)
&& (ResponseLength < DATASIZE));
ResponseLength += strlen (Buf))
{
strcat (Msg.DataBuffer, Buf);
}
Msg.DataLength = strlen (Msg.DataBuffer);
if ((RetCode = RccSendMsg (Reference, &Msg))
!= SUCCESS)
{
RccError (RetCode, ErrorStr);
printf("Send failed: %s\n", ErrorStr);
continue;
}
}
pclose (Ptr);
}
printf ("Accepting ....\n");
}
/* RccAccept failed. A timeout is acceptable */
if (RetCode != RCE_TIMEOUT)
{
RccError (RetCode, ErrorStr);
printf("RccAccept failed: %s\n", ErrorStr);
exit (1);
}
}
```

10.3 Using the Risc interface in multi-threaded applications

In multithreaded Windows NT/2000 applications, each thread must be treated as if it is a total independent connection to the Reality database. When a thread that uses the Risc Interface starts, it must call RgcStartUpServices(), which performs initialisation operations and starts up an Asynchronous thread to handle messaging, followed by RiscConnect() to connect to the database and RiscOpen() to open the required file. The data in the file can then be manipulated using the functions provided in the Risc and General Services interfaces.

When the thread terminates it must call RiscClose() to close the file, followed by

RiscDisconnect() to disconnect from the database and finally RgcShutDownServices().

The example which follows starts just one user thread, but this then calls most of the Risc functions to demonstrate the sequence in which an application might make Risc function calls.

10.3.1 Creating a Reality data file and an index file

The test program requires a database containing a file called TEST, based on the error message file from a standard Reality database. In addition, an index called BY-A1 must be defined for the TEST file, indexing using attribute 1. To create such a data file and index, proceed as follows:

1. Log on to the Sysman account by entering:

   ```
   LOGTO SYSMAN
   ```

2. Then create the TEST file by entering:

   ```
   CREATE-FILE TEST_FILE 1 1001
   ```

3. Copy the contents of the standard error message file to the newly created TEST file by entering:

   ```
   COPY errmsg *
   ```

4. Define an index on the TEST file by entering:

   ```
   DEFINE-INDEX TEST BY A1
   ```

   Where A1 is a reference to the second attribute in each data item, namely attribute number 1.

5. At the TO: prompt type:

   ```
   BY-A1
   ```

6. Finally, build the index by issuing the command:

   ```
   CREATE-INDEX TEST BY-A1
   ```

   To view the new file, enter:

   ```
   SORT TEST BY 1 1 2 3 4.
   ```

10.3.2 Amending the example code

The source code for the test program must know the names of the database and user so it must be changed as follows:

1. Change the #define statement for DATABASE_NAME to the name of your database.
2. Change the #define statement for USER_NAME to the user logon name to be used by the test program.

10.3.3 Example code

```
#include <process.h>
#include <risc.h>
```

```
#include <stdio.h>
#include <rfc.h>
#include <rgc.h>
#include <rlc.h>
#ifndef _INC_WINDOWS
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#endif
/****************** CONSTANT DEFINITIONS ****************************/
#define DATABASE_NAME "dbrog" // put your database name here (in quotes)
#define USER_NAME "rog" // put your logon name here (in quotes)
#define MAX_KEY_LENGTH 300
#define MAX_RECORD_LENGTH 500
#define MAX_PARTS 30
#define MAX_KEY_VAL_LENGTH 20
#define MAX_FILENAME_LENGTH 30
#define MAX_SMALL_RECORD_LENGTH 10
#define MAX_USERS 25
#define KEY_VALUE_ONE "400" // attribute 0 value that does NOT exist
#define KEY_VALUE_TWO "405" // atrribute 0 value that does exist
#define KEY_VALUE_THREE "L(1)" // attribute 1 value that does exist
#undef main
/****************** FUNCTION DECLARATIONS ****************************/
static int MainProgram1(UINT32 user_number);
/****************** MACRO DEFINITIONS
***********************************/
#define USER_PROMPT { fprintf(stdout,"\n\nPress \"Enter\" to
continue...."); \
getchar(); }
/****************************************************************************
*/
void main()
{
// arrays specified to allow code to be modified to
// support mulitple concurrent users
HANDLE NewThreadHandle[MAX_USERS] = {'\0'};
DWORD NewThreadID [MAX_USERS] = {'\0'};
DWORD TreadExitCode [MAX_USERS] = {'\0'};
int NumberOfUsers = 1;
UINT32 user_number = 1;
fprintf(stdout,"\n RISC Interface Test Program\n\nAbout to start a new
thread for user %d.",user_number);
// start a new synchronous thread, in suspended mode, for the new user
if (!(NewThreadHandle[user_number-1] = (HANDLE)_beginthreadex(0,0,
(LPTHREAD_START_ROUTINE)MainProgram1,(LPVOID)(user_number),
CREATE_SUSPENDED,&NewThreadID[user_number-1])))
{
fprintf(stdout,"\n\nERROR #1 in primary thread -- Unable to create new
thread for user %d",
user_number);
goto Exit;
}
// start progress on the new thread running MainProgram1 to completion
ResumeThread(NewThreadHandle[user_number-1]);
// primary thread waits here until all user threads terminates
fprintf(stdout,
"\nThe primary thread will now wait for user thread to complete....");
WaitForMultipleObjects(NumberOfUsers,NewThreadHandle,TRUE,INFINITE);
// retreive the exit code for the terminated user thread
```

```
if (GetExitCodeThread(NewThreadHandle[user_number-
1],&TreadExitCode[user_number-1]) != TRUE)
{
fprintf(stdout,"\n\nERROR #2 in primary thread -- Bad call to
GetExitCodeThread for user %d",
user_number);
}
// check if the synchorous thread reported an error
if (TreadExitCode[user_number-1] != 0)
{
fprintf(stdout,"\n\nERROR #3 in primary thread -- \nSync thread for user %d
returned error_code %d ",
user_number,TreadExitCode[user_number-1]);
}
// dispose of the user thread handle
if (CloseHandle(NewThreadHandle[user_number-1]) != TRUE)
{
fprintf(stdout,"\n\nERROR #4 in primary thread -- Bad call to CloseHandle
for
user %d",user_number);
}
Exit:
fprintf(stdout,"\n\nThe primary thread is about to end\n");
USER_PROMPT
}
int MainProgram1(UINT32 user_number)
{
const char *this_name = "MainProgram1";
char DatabaseName[] = DATABASE_NAME;
char User[] = USER_NAME;
char UserPasswd[] = "";
char Account[] = "SYSMAN";
char AcctPasswd[] = "";
char FileName[] = "TEST"; // Test file based on error messages
char IndexName[] = "BY-A1"; // Index on Test using attribute 1
char KeyVal [MAX_KEY_VAL_LENGTH+1] = {'\0'};
char KeyBuff [MAX_KEY_LENGTH+1] = {'\0'};
char RecBuff [MAX_RECORD_LENGTH+1] = {'\0'};
char NewFileName [MAX_FILENAME_LENGTH+1] = {'\0'};
char NewIndexName [MAX_FILENAME_LENGTH+1] = {'\0'};
int NewRecSize = 50;
int NewNumRecs = 100;
int KeyValLen = 0; // must be set to strlen(KeyVal)
int KeyBuffLen = 0;
int Result = 0;
int RecLen = 0;
int NumParts = 10;
int StartUpResult = 0;
int CodeLevel = 0;
int i = 0;
BOOL IndexSelected = FALSE; // influences choice of value to search for in
database file
BOOL MoreToRead = FALSE; // used when reading with a very small buffer
RISC_FILE FileHandle = NULL;
RISC_FILE NewFileHandle = NULL;
RISC_POS Position = RISC_GE; // RISC_BEG, RISC_EQ, RISC_GE, RISC_END
RISC_DESC IndexDesc[MAX_PARTS] = {0}; // must be initialised before calling
RiscCreateIndex()
RISC_DIR Direction = RISC_CURR;
```

```
RISC_OPT LockOpts = RISC_LOCK_NONE;// RISC_LOCK_NONE, RISC_LOCK_WAIT,
RISC_LOCK_NOWAIT, RISC_LOCK_HOLD
/*********************************************************************/
fprintf(stdout,"\nNew thread started for user %d to execute
%s",user_number,this_name);
USER_PROMPT
CodeLevel = 1;
RgcStartUpServices(StartUpResult);
if (StartUpResult != 0)
goto Exit;
else
fprintf(stdout,"\n%2d Good call made to \"RgcStartUpServices\"",CodeLevel);
CodeLevel = 2;
if (Result = RiscConnect(DatabaseName,User,UserPasswd,Account,AcctPasswd))
goto Exit;
else
fprintf(stdout,"\n%2d Good call made to \"RiscConnect\"",CodeLevel);
CodeLevel = 3;
if (Result = RiscOpen(FileName,&FileHandle))
goto Exit;
else
fprintf(stdout,"\n%2d Good call made to \"RiscOpen\"",CodeLevel);
CodeLevel = 4;
if (Result = RiscSelect(FileHandle,IndexName)) // selects an index table to
use for record accessing
goto Exit;
else
{
IndexSelected = TRUE;
fprintf(stdout,"\n%2d Good call made to \"RiscSelect\"",CodeLevel);
}
CodeLevel = 5;
_snprintf(KeyVal,sizeof(KeyVal)-1,"%s",KEY_VALUE_ONE);
// set search value (exact does not have to exist in file)
KeyValLen = strlen(KeyVal);
fprintf(stdout,"\n%2d KeyValue has been set to
\"%s\"",CodeLevel,KEY_VALUE_ONE);
CodeLevel = 6;
if (Result = RiscPosition(FileHandle,Position,KeyVal,KeyValLen))
goto Exit;
else
fprintf(stdout,"\n%2d Good call made to \"RiscPosition\"",CodeLevel);
Direction = RISC_PREV; // read item previous to current position in indexed
table
LockOpts = RISC_LOCK_WAIT; // LOCK THE RECORD WHEN READ
CodeLevel = 7;
if (Result = RiscRead(FileHandle,Direction,LockOpts,KeyBuff,MAX_KEY_LENGTH,
&KeyBuffLen,RecBuff,MAX_RECORD_LENGTH,&RecLen))
goto Exit;
else
fprintf(stdout,"\n%2d Good call made to \"RiscRead\"",CodeLevel);
RecBuff[RecLen] = '\0'; // NULL terminate the Record Buffer
KeyBuff[KeyBuffLen] = '\0'; // NULL terminate the Key Buffer
CodeLevel = 10;
if (Result = RiscUpdate(FileHandle,RecBuff,RecLen))
goto Exit;
else
fprintf(stdout,"\n%2d Good call made to \"RiscUpdate\"",CodeLevel);
CodeLevel = 11;
```

```
if (IndexSelected == TRUE)
// using index BY-A1. Set search value to an attribute 1 value
_snprintf(KeyVal,sizeof(KeyVal)-1,"%s",KEY_VALUE_THREE);
else
// set search value to an attribute 0 value
_snprintf(KeyVal,sizeof(KeyVal)-1,"%s",KEY_VALUE_TWO);
KeyValLen = strlen(KeyVal);
LockOpts = RISC_LOCK_WAIT; // LOCK THE RECORD WHEN READ
fprintf(stdout,"\n%2d KeyValue has been set to \"%s\"",CodeLevel,KeyVal);
CodeLevel = 12;
if (Result =
RiscReadByKey(FileHandle,LockOpts,KeyVal,KeyValLen,RecBuff,MAX_RECORD_LENGT
H,&RecL
en))
goto Exit;
else
fprintf(stdout,"\n%2d Good call made to \"RiscReadByKey\"",CodeLevel);
RecBuff[RecLen] = '\0'; // NULL terminate the Record Buffer
KeyBuff[KeyBuffLen] = '\0'; // NULL terminate the Key Buffer
USER_PROMPT
// READ A RECORD FROM DB WHERE ID IS REQUIRED VALUE, USING A VERY SMALL
BUFFER
CodeLevel = 13;
for (i=0; i<MAX_RECORD_LENGTH; i++)
{
RecBuff[i] = '\0'; // clear out the buffer
}
RecLen = 0;
LockOpts = RISC_LOCK_WAIT; // LOCK THE RECORD WHEN READ
Result = RiscRead(FileHandle,
Direction,
LockOpts,
KeyBuff,
MAX_KEY_LENGTH,
&KeyBuffLen,
RecBuff,
MAX_SMALL_RECORD_LENGTH, // SMALL BUFFER USED
&RecLen);
if (Result == RFE_READEXCEED)
{
RecLen = MAX_SMALL_RECORD_LENGTH; // RecLen returns size of item NOT buffer
occupency
MoreToRead = TRUE;
}
else if (Result != 0)
{
fprintf(stdout,"\n%2d -- Bad call made to \"RiscRead\" Result=%d \n\"%s\"
\n",
CodeLevel,Result,RgcErrMsg(Result));
fprintf(stdout,"\nKeyBuffLen=%d KeyBuff = \"%s\"\n",KeyBuffLen,KeyBuff);
goto Exit;
}
if ((Result == 0) || (Result == RFE_READEXCEED))
{
RecBuff[RecLen] = '\0'; // NULL terminate the Record Buffer
KeyBuff[KeyBuffLen] = '\0'; // NULL terminate the Key Buffer
fprintf(stdout,"\n%2d Good call made to \"RiscRead\"\nRecLen=%d RecBuff
contents =\"%s\"",
CodeLevel,RecLen,RecBuff);
```

```
fprintf(stdout,"\nKeyBuffLen=%d KeyBuff = \"%s\"
MoreToRead=%d\n",KeyBuffLen,KeyBuff,MoreToRead);
}
// READ REMAINDER OF RECORD
CodeLevel = 14;
while (MoreToRead == TRUE)
{
Result = RiscReadRest(FileHandle,
RecBuff,
MAX_SMALL_RECORD_LENGTH,
&RecLen); // this function returns ammount of data read, NOT item
size
if (Result != 0)
{
if (Result == RFE_READEXCEED)
MoreToRead = TRUE;
else
{
fprintf(stdout,"\n%2d -- Bad call made to \"RiscReadRest\" Result=%d
\n\"%s\"\n",
CodeLevel,Result,RgcErrMsg(Result));
goto Exit;
}
}
else
MoreToRead = FALSE;
RecBuff[RecLen] = '\0'; // NULL terminate the Record Buffer
KeyBuff[KeyBuffLen] = '\0'; // NULL terminate the Key Buffer
fprintf(stdout,"\n%2d Good call made to \"RiscReadRest\"\nRecLen=%d RecBuff
contents =\"%s\"",
CodeLevel,RecLen,RecBuff);
fprintf(stdout,"\nKeyBuffLen=%d KeyBuff = \"%s\"
MoreToRead=%d\n",KeyBuffLen,KeyBuff,MoreToRead);
}
USER_PROMPT
CodeLevel = 15;
if (Result = RiscUnlock(FileHandle))
goto Exit;
else
fprintf(stdout,"\n%2d Good call made to \"RiscUnlock\"",CodeLevel);
_snprintf(RecBuff,sizeof(RecBuff)-1,"#00%d\376written by user %d during
MainProgram1",
user_number,user_number);
RecLen = strlen(RecBuff);
CodeLevel = 16;
if (Result = RiscWrite(FileHandle,RecBuff,RecLen))
goto Exit;
else
fprintf(stdout,"\n%2d Good call made to \"RiscWrite\"",CodeLevel);
_snprintf(NewFileName,sizeof(NewFileName)-1,"NEWFILE_%d",user_number);
CodeLevel = 17;
if (Result = RiscCreateFile(NewFileName,NewRecSize,NewNumRecs))
goto Exit;
else
fprintf(stdout,"\n%2d Good call made to \"RiscCreateFile\"",CodeLevel);
CodeLevel = 20;
if (Result = RiscOpen(NewFileName,&NewFileHandle))
goto Exit;
else
```

```
fprintf(stdout,"\n%2d Good call made to \"RiscOpen\"",CodeLevel);
_snprintf(RecBuff,sizeof(RecBuff)-1,"%03d\376item written by user
%d",user_number);
RecLen = strlen(RecBuff);
CodeLevel = 21;
if (Result = RiscInsert(NewFileHandle,RecBuff,RecLen))
goto Exit;
else
fprintf(stdout,"\n%2d Good call made to \"RiscInsert\"",CodeLevel);
_snprintf(NewIndexName,sizeof(NewIndexName)-1,"INDEX_%02d",user_number);
CodeLevel = 22;
if (Result =
RiscDescribeIndex(FileHandle,IndexName,MAX_PARTS,&NumParts,IndexDesc))
goto Exit;
else
fprintf(stdout,"\n%2d Good call made to \"RiscDescribeIndex\"",CodeLevel);
CodeLevel = 23;
if (Result = RiscCreateIndex(NewFileName,NewIndexName,NumParts,IndexDesc))
goto Exit;
else
fprintf(stdout,"\n%2d Good call made to \"RiscCreateIndex\"",CodeLevel);
CodeLevel = 24;
if (Result = RiscDeleteIndex(NewFileName,NewIndexName))
goto Exit;
else
fprintf(stdout,"\n%2d Good call made to \"RiscDeleteIndex\"",CodeLevel);
CodeLevel = 25;
if (Result = RiscClear(NewFileHandle))
goto Exit;
else
fprintf(stdout,"\n%2d Good call made to \"RiscClear\"",CodeLevel);
CodeLevel = 26;
if (Result = RiscClose(NewFileHandle))
goto Exit;
else
fprintf(stdout,"\n%2d Good call made to \"RiscClose\"",CodeLevel);
CodeLevel = 27;
if (Result = RiscDeleteFile(NewFileName))
goto Exit;
else
fprintf(stdout,"\n%2d Good call made to \"RiscDeleteFile\"",CodeLevel);
CodeLevel = 28;
if (Result = RiscDisconnect())
goto Exit;
else
fprintf(stdout,"\n%2d Good call made to \"RiscDisconnect\"",CodeLevel);
RgcShutDownServices();
Exit:
if (Result != 0)
{
fprintf(stdout,"\n\nError -- user_number %d code level %d return value=
%d\n\"%s\"\n",
user_number,CodeLevel,Result,RgcErrMsg(Result));
}
else
fprintf(stdout,"\n\nRISC Interface Test Program completed for \nuser %d
with
no errors detected\n",
user_number);
```

```
_endthreadex(Result);
return (Result);
}
```

Section 11: Glossary

Term or abbreviation	Definition
Client	A program that initiates a connection.
Client ID	Part of a DDA CONNECT message – identifies the calling system and user.
CONNECT	A DataBasic statement used to initiate a program-to-program connection.
DDA	Distributed Data Access – a simple communications architecture designed for the NEC family of products.
Handle	• Account: An account handle is used to reference a database. • File: A file handle is used to reference an open file. • List: A list handle is used to reference a saved list. • Session: See **Session Reference**.
IEEE 802.3	A Local Area Network standard.
IFA	Interactive File Access – a facility which allows C programs to access Reality files.
IPC	Inter-process Communication – a Reality facility which allows programs written by a user to communicate.
LAN	Local Area Network.
Library	A suite of related C functions providing a particular service.
Listening entry	In the UNIX ROUTE-FILE, an entry which describes the local system such that session manager can accept incoming calls. On Windows, the registry entry NEC/ Reality/ Listening has the same function.
Logging	A file logging option is available when a file is opened for systems equipped with transaction logging.
Looping	A loopback connection is one that connects back to itself rather than to a remote entity.
MCB	Message Control Block.
PLId	Physical Location Identifier – part of the DDA CONNECT message.
Rcc	Reality Communications Interface for the C language – a set of C library functions which

Term or abbreviation	Definition
	enable a C program to communicate with a DataBasic program on a Reality system.
Rcs	Reality Communications Service.
Rfc	Reality Filing Interface for C Programs.
Rgc	Reality General Services Interface for C Programs.
Rlc	Reality List Services Interface for C Programs.
ROUTE-FILE	A file, at system level on a Reality system and in the /etc directory on a UNIX system which contains routing information required by the system to make a connection to a remote system. On Windows, routing information is stored in the registry.
Server	A program that responds to a client.
Session manager	A process which establishes and monitors connections.
Session reference	A variable used by client and server programs, that is allocated a number when a connect/accept is successful. The variable is used to reference the established session in all further program to program communication.
S-LAN	A NEC LAN controller for Reality systems.
Timeout	Part of a DDA CONNECT message that specifies the amount of time within which a connection must be made.
USERS-FILE	A file on a UNIX system which contains local user-ids and maps them onto network user IDs.
WAN	Wide Area Network (such as X.25).