Polyhedra®

Evaluating Polyhedra

Enea Polyhedra Ltd

ENEA

Copyright notice

Disclaimer

Trademarks

Manual Details

Manual title	**Evaluating Polyhedra**
Document number	**src/release/evaluate**
Revision number	**9.0.0**
Revision Date	**2nd June 2015**

Software Version

This documentation corresponds to the following version of the Polyhedra software:

Version	**9.0**

Preface

This document describes the Polyhedra release kits. It discusses how to run Polyhedra components and presents a series of demos that can be used to familiarise the user with the Polyhedra system.

CONTENTS

1. Introduction

Welcome. You are about to start using Polyhedra, a real-time, cross-platform, database management system. This document works through a series of simple demos, each of which introduces and illustrates a feature of the Polyhedra product set. It assumes that you have already successfully installed Polyhedra, in accordance with the instructions in a companion document, 'Installing Polyhedra' (install.pdf).

This document contains the necessary information required to run the Demo Suite. It is also recommended that users refer to the supplied readme.pdf file supplied with this release for current changes and additional information not yet included in the Demo Suite manuals.

This document is divided into two parts. The first part describes the various demos provided as part of the installation kit. It is strongly advised that those new to Polyhedra work through these demos sequentially in order to gain familiarity with the product.

The final section continues the familiarisation with Polyhedra, covering topics such as writing and compiling your own client applications, additional details about the fault tolerance mechanisms, and an introduction to the Polyhedra manual set.

This document assumes some familiarity with the concepts of relational databases and object oriented systems. It is not intended as a training course in the use of Polyhedra. However, it should allow you to gain some familiarity with the basic facilities of Polyhedra and enable you to progress towards setting up your own databases and writing your own client applications. There is a separate set of reference manuals (covered in section 3) giving details of various aspects of Polyhedra, and which should be treated as the authoritative sources - this evaluation guide is a simplified introduction.

NOTE:	**This document is for use with all editions of the Polyhedra product. However, some of the demos it covers use features that are not supported by some editions of Polyhedra - where this applies, the description of the demo will make this clear. To find out what features are supported by the edition of Polyhedra you are using, please refer to the feature.txt file in the Polyhedra release kit.**

2. Running the Demo Suite

2.1 Introduction

The Polyhedra demo suite is designed to illustrate the various important features of Polyhedra, the working methods employed by Polyhedra, and the terms used when referring to the various components and configurations.

The demo suite is set up to introduce users to the Polyhedra database. This is supplemented by a series of demos, which, if followed in order, will provide the user with a foundation in the principles of Polyhedra and its capabilities. The section called "Further Directions" shows how the user can progress towards constructing their own database applications.

2.2 About Polyhedra, and the Demonstration Suite

Polyhedra is a real-time, relational database possessing features that are highly applicable to the vertical market world, finding itself strategically placed as an integral part of many products around the world. It is an object-relational, client-server database application for real-time embedded systems application development. Based on open standards, Polyhedra's advanced database technology is targeted at an increasing number of areas that require a level of performance, flexibility and availability that conventional databases cannot provide. It presents a unique combination of database technology, including Active Queries and object method encapsulation, with relational database standards for high speed, event-driven SQL database execution.

The standard Polyhedra database executes entirely in main memory, using standard and optimised memory allocation, to produce performance that is many times faster than conventional disk-based databases, even if they are entirely cached in memory. Data identified as persistent, including SQL transactions, is written to non-volatile magnetic media. Polyhedra's sophisticated data persistence mechanism includes a full fault tolerant configuration providing continuous database server availability with fault tolerant transactions and clients. The standard Polyhedra database is usually only limited by available memory.

A Polyhedra Flash DBMS database is stored in a flash file and its size is therefore not limited by the available memory (RAM).

Application programming interfaces enable real-time database application developers to produce heterogeneous client-server and web-database server applications on 32- and 64-bit platforms for major UNIX, Linux, RTOS and Windows computing platforms.

Polyhedra is a "transaction based" system that operates as a user process on an operating system. The standard database runs entirely in random access memory and at a minimum requires sufficient memory to hold the entire database it will manage with an additional 2-3M of disk space depending on the operating system and the functionality included.

In this demonstration suite the following features are shown:

The database	A basic introduction to it and SQL.
Data Persistence	The survival of nominated data across a restart of the database. This uses the journaling feature and so is not applicable to all editions of Polyhedra.
Fault Tolerance	A hot standby facility allowing Polyhedra to be used where high availability is a requirement. This uses the fault-tolerance feature and so is not applicable to all editions of Polyhedra.
CL	A full database programming language, allowing code to be written that runs inside the database performing various functions. These range from simple range checking, to full applications enforcing application level integrity and performing useful and safe denormalization of data for improved performance.

As discussed in section 2.10, there are number of other features of Polyhedra that are not covered by this version of the demonstration suite. The demo suite and this document are not intended to be a complete tutorial in the use of Polyhedra, but aim to provide familiarity with the most commonly-used features enabling users to begin constructing their own individual applications.

Each demonstration set contains a database that is started first, followed by a series of exercises, each of which demonstrate one or more specific features of Polyhedra. The demonstration sets share a common theme, a table showing the conversion rates between various currencies. The demonstration sets are numbered 1 to 5, the different exercises are called 1a, 1b, 2a, 2b, 2c etc., and the presentation that follows assumes that you work through these in order.

2.3 Demo suite layout

Each demo in the demonstration suite is based on the same central theme of having a table containing the details for various currencies, including a conversion rate against the US Dollar. Thus, in most of the demos sets a single table called "currency" is defined, and its initial values are:

```
+-------+---------------+----------+----------+
| code  | country       | name     | usdollar |
+-------+---------------+----------+----------+
| 'GBP' | 'UK'          | 'Pound'  |     0.67 |
| 'EUR' | 'Eurozone'    | 'Euro'   |     0.81 |
| 'CHF' | 'Switzerland' | 'Franc'  |     1.67 |
| 'CAD' | 'Canada'      | 'Dollar' |     1.49 |
| 'AUD' | 'Australia'   | 'Dollar' |     1.72 |
| 'JPY' | 'Japan'       | 'Yen'    |    109.5 |
+-------+---------------+----------+----------+
```

NOTE:	**The values illustrated are for example purposes only and should not be taken as actual or current exchange rate values**

The table was created using the following sql:

```
create table currency
( persistent
, code     large varchar primary key
, country  large varchar
, name     large varchar
, usdollar real
);
```

(the significance of the 'persistent' keyword is discussed in demo 3, covered in section 2.7.). Some of the demos use additional tables or a modified definition of the above; for those demos, the schema in use is documented with the description of the demo.

2.3.1 Demonstration Suite databases

The demo sets are in directories with names of the form poly<version>\examples\demo_<name>. The demo sets are summarised below, and covered in more detail in individual subsections as indicated.

2.3.1.1 Demo_1

This is a simple, single table database containing the currency table above. The exercises will show how to start a Polyhedra database and an SQL session to that database. They will go on to demonstrate some simple clients that perform queries and make changes to the data. There is a demonstration here of the Active Query mechanism, one of the unique features of Polyhedra. See section 2.5 for more details.

2.3.1.2 Demo_2

This is a demonstration of table inheritance. Polyhedra is an object-oriented database that allows tables to be derived from one another. In this demonstration, a table called "currency_limits" is derived from the basic currency table, adding columns to provide set limits outside of which the currencies should not fluctuate. A client using the active query mechanism is used to show when the currencies transgress the limits. See section 2.6 for more details.

2.3.1.3 Demo_3

This is a demonstration of data persistence. Polyhedra has a data persistence mechanism that allows nominated data in the database to be persisted to a non-volatile store, such as a hard disk. A client is supplied that controls the data persistence mechanism, and the various types of data persistence are discussed. See section 2.7 for more details.

This demonstration uses the journaling feature and so is not applicable to all editions of Polyhedra.

2.3.1.4 Demo_4

This is a demonstration of fault tolerance. Polyhedra can be configured in a fault tolerant manner, to include a master and a "hot" standby database. In this demonstration, clients are used to show how a client can access the fault-tolerant pair of

databases. It also illustrates how the operation of the database is almost seamlessly transferred from the master database to the standby database when a failure occurs. See section 2.8 for more details.

This demonstration uses the fault tolerance feature and so is not applicable to all editions of Polyhedra.

2.3.1.5 Demo_5

This is a demonstration of CL code. Polyhedra has a Control Language (CL), that runs inside the database, triggered by changes to data. This demonstration shows this language being used to monitor the values of the currencies as they fluctuate, informing the user when the values transgress the limits placed on them by the currency_limits table. See section 2.9 for more details.

2.3.1.6 Other demonstrations

In addition to the worked examples described above, additional demonstrations are available, some of which are provided as separate subdirectories in the examples directory. Each such directory will contain a text file describing the demo, and for complex examples there may also be a PDF file. Note that these additional examples assume the user is familiar with the general features of Polyhedra, and in particular that the user has experimented with the worked examples described in this evaluation guide - and thus knows how to run the rtrdb, the sqlc command-line interface, and the precompiled clients supplied in the release kit.

Note that not all of the additional demonstrations are applicable to all editions of Polyhedra; where applicable, the description file will list the features used, and you can determine from the features.txt file in the release kit if your edition of Polyhedra should be able to run the demo.

2.3.2 Demo Directory Layout

This section details the layout of the directories that make up the Polyhedra demonstration suite.

2.3.2.1 poly<version>\examples\demo_<n>

There are 5 demonstration database configurations contained in directories called Demo_1 to Demo_5. These are referred to here as the "main demo directories". For each of the demonstrations the user will start a command prompt, change directory to the main demo directory for the required demonstration and start the database(s). The appropriate clients will then need to be started, which will then show the demonstration.

The main demo directory for each demo will contain at least the following files:

- poly.cfg — the configuration file. This contains the parameters that set up how the database will start up. These parameters are called “resources”.
- test.dat — the database load_file. The file that contains the initial schema and data for the database, and which the database starts from. In the fault tolerance demonstration, demo_4, the test.dat file is kept in the logdir1 and logdir2 subdirectories.

There will also be a file called <n>_demo_name - for example:

```
01_Simple_Database
```

This file provides an indication of which demonstration directory the user has selected. The file also contains version information.

Each demo contains a subdirectory called “init”. This contains the necessary files used to restore the database back to its original state - this is detailed in section 2.4.4 of this document.

2.3.2.2 poly<version>\<platform>\<target>\bin

This contains the executables for the various Polyhedra components, and the various clients that are used throughout the demonstration.

2.3.2.3 poly<version>\examples\clients

There is a directory called "clients" at the same level as the demo_<n> directories. This contains all the necessary source code used to build the various sample client applications used throughout the demonstration suite. There is a directory for each of the clients, and two sub directories called "callback" and "odbc". These subdirectories contain the required source code to build clients using either of the two native API’s. Instructions on how to compile these are given in the "Further Directions" section.

2.3.2.4 poly<version>\examples\template

This directory contains a template of files and a suitable poly.cfg for building your own databases. This is discussed in the "Further Directions" section.

2.3.3 Other information

All demonstrations are started from a command prompt. You will need to be in the directory where the demo resides. Instructions throughout this manual follow this format:

Instructions entered by the user are in Courier font and bold e.g.:

```
% rtrdb db
```

Responses from the computer are in Courier:

```
List of currencies in database
```

When the user is using the interactive SQL client (SQLC) then the line will be prefixed by SQL>. This is shown, but should not be entered by the user e.g.:

```
SQL> select * from currency;
```

NOTE:	**This demo suite does not intend to be a tutorial on SQL or relational databases, and as such assumes a basic familiarity with the concepts. However, for your convenience samples of many types of SQL statements are provided in the 'Further Directions' part of this document - see Section 3**

The following manuals will be useful to you in your evaluation of Polyhedra:

- userguid.pdf
- sql.pdf
- sqlc.pdf
- rtrdb.pdf
- clref.pdf

2.4 Components of the demo suite

The Polyhedra demo suite consists of the main Polyhedra executables, various precompiled example client applications, and some sample databases. Each individual demonstration will require you to start the correct database as described in the documentation for that particular demonstration, and then perform various exercises (such as running some of the example client applications) to illustrate particular features of the Polyhedra system.

2.4.1 Polyhedra components

The main Polyhedra Components are:

- rtrdb
- sqlc
- clc

The Polyhedra components are best started from the command (or shell) prompt. The rtrdb component is a console application, and so may require a new console for each instance of the application (see section 2.4.3 for more details). Each of these components will also take an argument, which is an entry point into the configuration file “poly.cfg” in the directory in which the component is running. Consequently, when starting these components, you must be in the main demo directory of the demo you wish to run.

2.4.2 Clients

The example clients are also best started from a command prompt. They are console applications, and most of the clients print output on a continual basis until they are deliberately stopped - these are better started in their own windows. These are

- active
- animate
- monitor
- persist
- query
- transact
- websocket

The function and use of these clients is described later, as they are first made use of in the demonstrations. They do not have to be started from the demonstration directory, but can be started from anywhere. The instruction given in the demonstration for running a client will be shown (for example) as:

```
% active
```

In addition to any other parameters they may need, each of the clients take an optional parameter, which is always the last parameter specified. This is the data service port that the client will try to connect to. By default this is 8001, which is the port that most of the databases in this demonstration suite use (the fault tolerance demo being the only exception). The instructions do not show the optional parameter, as it is not needed when using the release kit as supplied - except in the fault tolerant examples, where the data service port must be specified. If the client is run on a different machine to that running the server to which you want to connect, or if you have modified any of the demonstrations to use a different data service, then the format for the parameter is **"<ip address>:<port number>"**as in:

```
% active 10.1.2.3:8001
```

2.4.2.1 The Animator client

One special client that we have supplied is called the animator. It is designed purely to change the data in a demo database in a sensible way without the user having to continually update the data by hand. It is useful for showing the data changing in the database, and is used in various places for this purpose.

The animator client is called “animate”. Once every two seconds, it modifies the ‘usdollar’ attribute of each currency record, first stepping the values up until they reach 110% of their starting values, then back down to 90% of their starting values, and so on. It produces messages that illustrate these changes to the user.

2.4.2.2 The Websocket client

The websocket client provides the interface that allows a web browser to show an automatically updated view of the example currency database. The websocket client listens for a websocket connection (normally from a web browser) and when requested performs an active query on the currency table of the example database. The active query sends changes of data to the client that then forwards those changes on to the web browser via the websocket protocol. The client performs no polling. All data changes are event driven.

To use the client whilst exploring the various demonstrations, follow the instructions in section 2.5.3.

The client is pre-built for Linux and Windows platforms. For other platforms or to alter the supplied client, please see the source code in the **poly<version>\examples\demo_websocket** folder.

The websocket client implements a minimum version of the Websocket protocol and is compatible with the current Google Chrome and Mozilla Firefox web browsers. The Linux implementation will handle concurrent connections (i.e. more than one web page at a time) whereas the Windows implementation will handle one concurrent connection. There is a limitation of the example websocket.c code that a closed websocket connection will only be noticed when a change is forwarded on to the closed websocket.

2.4.3 Platform considerations – Starting and stopping

Different operating systems require different command syntax to start and stop the Polyhedra executables. Whichever platform is being used the Polyhedra components should be started where they have access to the files in the demonstration directory.

To start the simple database that forms part of the demo_1, the instruction to run the database is simply shown as:

```
% rtrdb db
```

It is assumed that you have set yourself to the right directory, and for clarity the prompt is just represented in the instructions by a % sign. Starting the database will display the output similar to the following:

```
Component rtrdb (V09.00.0000/WIN32-i386)
Copyright (C) 1994-2015 by Enea Software AB
Constructing Database db
Loading 'test.dat'
Ready
```

The first line of the output will confirm the revision level of Polyhedra that is being used. When the "Ready" message is displayed, then the database is ready for connections to be made to it.

2.4.3.1 Windows

Under Windows, you would start an MS-DOS prompt, probably from the “*Start Menu -> Programs -> MS-DOS Prompt*” menu item.

You would then change to the demo directory.

For example:

```
c:\> cd \poly<version>\examples\demo_1
```

and then start the database:

```
c:\poly<version>\examples\demo_1> start rtrdb db
```

The SQL client (SQLC) will start a new window for itself, within which input to and output from the component can be seen.

Some of the demonstrations call for a database to be killed, or a client to be stopped without shutting down the database to which it is attached. On Windows, the easiest way to achieve this is to type Control-C, or using the Windows Task Manager.

2.4.3.2 Unix

The procedure for starting the database on a Unix operating system from a shell is to change directory:

```
% cd ~/poly<version>/examples/demo_1
```

and then start the database (in the background):

```
% rtrdb db &
```

The SQLC should be started in the foreground.

To kill a database, or to stop a client without shutting down the database to which it is attached, the '`kill -9 <processid>`' command should be used. On some Unix systems, depending on the command shell you are running, you may be able to kill the database by typing:

```
% kill -9 %rtrdb
```

2.4.3.3 OSE

To start the database from an OSE telnet session or shell, first change directory:

```
rtose@walnut> cd /ram/poly<version>/examples/demo_1
```

and then start the database (in the background):

```
rtose@walnut> pgrun rtrdb.elf db &
```

The sqlc does not need to be run on the same machine as the database. For example, an instance of sqlc running on a windows platform can be configured to connect to a database listening on tcp port 8001 running on an embedded platform:

```
c:\poly<version>\examples\demo_1> sqlc -rdata_service=10.1.2.3:8001 sql
```

To kill a database, or to stop a client without shutting down the database to which it is attached, the `pm_kill` command (with `pm_pginfo` to determine the program id) should be used:

```
rtose@walnut> pm_kill <programid>
```

2.4.4 Restoring databases back to their original state

It is possible that whilst using the release kit, you will 'corrupt' a database by deleting all the rows in a table, or dropping the table. (The database is not 'corrupt' as such, as these actions are perfectly valid; it is merely that the database is not in the state that it needs to be for the demonstration script to be correct.) In order to restore the database to its original state, each demo directory has a subdirectory called "init". Within this directory there is a safety copy of the test.dat file used by the database, which can be copied onto the test.dat file in the demo directory at any time that the database is not running.

NOTE:	**For the fault-tolerance demo, the test.dat from the init subdirectory should be copied into both the logdir1 and the logdir2 subdirectory rather than into the home directory for the demo.**

Also in this directory are the necessary files to re-create the demonstration database from scratch. These files are used as follows:

poly.cfg	this is the configuration file necessary to start a "blank" database and an SQLC session to it such that the original schema and data can be created.
tables.sql	this file contains the definition of the database schema - a set of "create table" statements.
data.sql	this file contains the initial data for the database.
reset.sql	the file contains enough sql to include the tables and data files, save the database to the file "test.dat", and close the database down.

In order to re-create the demonstration databases using these files you would need to start off a command shell, change directory to the 'init' directory, and follow the sequence detailed below:

- Firstly, ensure that no rtrdb process is running anywhere else on the machine. Polyhedra Flash DBMS users should also ensure that any previously created poly.fdb database file is deleted.
- Start a "blank" database, and an SQLC session to it

```
% rtrdb empty
% sqlc sql
```

- at the SQL prompt in the SQLC session, include the reset.sql file to recreate the tables and initial data, save the database and close the database down.

```
SQL> include 'reset';
```

The test.dat file created in the init directory as a result of this procedure will be the same as the file supplied with the release kit.

2.5 Demo 1 – a Simple Database

2.5.1 Description and Aims

In these demonstrations a simple, single table database is started. The table is called "currency" and shows example exchange rates for various currencies against the US Dollar.

With this database we aim to show how a simple database is started, introduce the poly.cfg file, and demonstrate the interactive SQL client "SQLC". The basics of SQL are introduced for users that are not familiar with SQL.

Demo 1a and Demo 1c are concerned with starting the demonstration database and using SQLC, while demo 1f is concerned with stopping the database cleanly. In the exercises Demo 1d and Demo 1e, we show simple clients and illustrate the Active Query mechanism. In Demo 1b, we show how to view data changes dynamically using a web application.

2.5.2 Demo 1a: starting the database

Start a command prompt and change directory to the main demo directory for demonstration 1 - **poly<version>\examples\demo_1.**

To start the database enter at the command prompt:

```
% rtrdb db
```

This starts the database management process, the rtrdb. The Polyhedra copyright messages will be output to stdout -usually the same place as where the database was started from.

Once initiated, the database manager will - in accordance with the configuration information specified in the poly.cfg file in the local directory - claim a tcp socket (on port 8001) on which clients can connect. It will also use the contents of test.dat to determine the initial schema and database contents. Using a standard RTRDB, the database is loaded into memory from the file. Using a Polyhedra Flash DBMS RTRDB, the database is maintained in the file. Each rtrdb process manages one database, and vice versa. You can simultaneously run many databases on a machine, by starting off many rtrdb processes, each with their own copy of the initial data and their own copy of poly.cfg so that they each claim different ports.

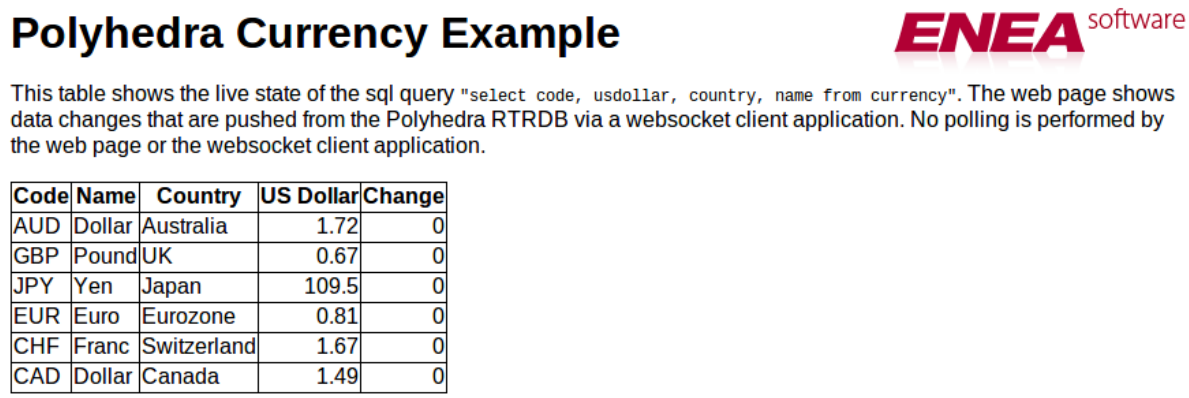
As (in a Polyhedra system) there is a one to one correspondence between databases and database manager tasks, the terms tend to be used interchangeably - purists might say 'starting the database management system' rather than starting the database.

2.5.3 Demo 1b: viewing the currency table via a web browser

First start the websocket client:

```
% websocket
```

Then open your browser (Firefox and Chrome supported) and load the **poly<version>\examples\demo_websocket\currency.html** file from the release kit. You should see the following in your browser:



As you proceed through the next demonstrations you should see the currency changes on the web page.

Below the main table of currencies you can see the JSON formatted messages that the websocket client is sending to the browser. The JavaScript in currency.js opens a websocket connection to the websocket client, requests updates, reads the JSON formatted messages and makes changes to the displayed table accordingly.

By default the websocket client listens for websocket connections on port 3400 and connects to a Polyhedra database on localhost:8001. You may alter this behaviour by starting the websocket client as

```
% websocket <websocket port> <Polyhedra data service>
```

You may also serve the currency.html and currency.js files from a web server in which case the websocket server should be run on the same machine, or you may alter the **currency.js** file.

2.5.4 Demo 1c: using SQLC to query and update the database

The simplest client and most frequently used in these examples, is the interactive SQL client - SQLC. To start the SQLC enter:

```
% sqlc sql
```

Under Windows, a new window will appear with a SQL> prompt. Under Unix, the copyright messages for SQLC will appear on stdout, and the SQL> prompt will appear.

2.5.4.1 Querying using SQLC

The SQL> prompt allows you to enter SQL directly, and see the results. For example, enter:

```
SQL> select * from currency;
```

to see all the data in the example table “currency”. The output should be:

```
+-------+---------------+----------+----------+
| code  | country       | name     | usdollar |
+-------+---------------+----------+----------+
| 'GBP' | 'UK'          | 'Pound'  |     0.67 |
| 'EUR' | 'Eurozone'    | 'Euro'   |     0.81 |
| 'CHF' | 'Switzerland' | 'Franc'  |     1.67 |
| 'CAD' | 'Canada'      | 'Dollar' |     1.49 |
| 'AUD' | 'Australia'   | 'Dollar' |     1.72 |
| 'JPY' | 'Japan'       | 'Yen'    |    109.5 |
+-------+---------------+----------+----------+
Query Done: 6 records selected
```

The table contains the currency code, the country this currency belongs to, and the name of the currency. The 'usdollar' column contains the amount of the currency required to buy 1 US Dollar.

Some other examples for you to try are:

```
SQL> select * from currency where country='Canada';

+-------+----------+----------+----------+
| code  | country  | name     | usdollar |
+-------+----------+----------+----------+
| 'CAD' | 'Canada' | 'Dollar' |     1.49 |
+-------+----------+----------+----------+
Query Done: 1 record selected

SQL> select * from currency where name='Dollar';

+-------+-------------+----------+----------+
| code  | country     | name     | usdollar |
+-------+-------------+----------+----------+
| 'CAD' | 'Canada'    | 'Dollar' |     1.49 |
| 'AUD' | 'Australia' | 'Dollar' |     1.72 |
+-------+-------------+----------+----------+
Query Done: 2 records selected

SQL> select * from currency where usdollar<1;

+-------+------------+---------+----------+
| code  | country    | name    | usdollar |
+-------+------------+---------+----------+
| 'GBP' | 'UK'       | 'Pound' |     0.67 |
| 'EUR' | 'Eurozone' | 'Euro'  |     0.81 |
+-------+------------+---------+----------+
Query Done: 2 records selected
```

In its simplest form, the SQL SELECT statement retrieves the contents of a table; the WHERE clause allows you to specify restrictions on the rows to be retrieved. By replacing the asterisk * by a comma-separated list of columns, you can retrieve a restricted set of columns rather than all of them. Extensions to the syntax allow you to take the cross product of tables, and select which rows and columns are to be returned - however, the result is always tabular.

2.5.4.2 A note for Windows users

The SQLC program on Windows provides a command history mechanism and allows 'command line editing'. This permits the use of the up, down, left and right arrow keys to recall and edit previous lines that you typed: the up and down arrows select the line to edit, and the left and right arrows allow you to move around within the selected line. When you press 'enter', the line (including any text beyond the current cursor position) is acted on, and added to the history. These facilities are described in more detail in section 3 of the sqlc reference manual, sqlc.pdf.

2.5.4.3 Changing data using SQL

You may also use various SQL statements to change the data in the currency table. Try some of the following statements to change data in the currency table. After each, enter "**select * from currency**" to see the changes that have been made.

```
SQL> update currency set usdollar=0.5 where code='GBP';

SQL> commit;
```

This will change the US Dollar value for the British Pound to 0.5.

```
SQL> select * from currency;

+-------+---------------+----------+----------+
| code  | country       | name     | usdollar |
+-------+---------------+----------+----------+
| 'GBP' | 'UK'          | 'Pound'  |      0.5 |
.
.
.
```

You can create records in the table:

```
SQL> insert into currency values ('KRW','Korea','Won',1012);

SQL> commit;
```

This will add a new currency - the Korean Won, worth roughly 1000 Won to the US Dollar.

```
SQL> select * from currency;

+-------+---------------+----------+----------+
| code  | country       | name     | usdollar |
+-------+---------------+----------+----------+
.
.
.
| 'KRW' | 'Korea'       | 'Won'    |     1012 |
+-------+---------------+----------+----------+
```

Entries can be deleted:

```
SQL> delete from currency where code='KRW';

SQL> commit;
```

This will delete the entry for the Korean Won that we have just entered.

```
SQL> select * from currency;

+-------+---------------+----------+----------+
| code  | country       | name     | usdollar |
+-------+---------------+----------+----------+
| 'GBP' | 'UK'          | 'Pound'  |      0.5 |
| 'EUR' | 'Eurozone'    | 'Euro'   |     0.81 |
| 'CHF' | 'Switzerland' | 'Franc'  |     1.67 |
| 'CAD' | 'Canada'      | 'Dollar' |     1.49 |
| 'AUD' | 'Australia'   | 'Dollar' |     1.72 |
| 'JPY' | 'Japan'       | 'Yen'    |    109.5 |
+-------+---------------+----------+----------+
```

Finally, quit the SQLC program. On Unix, this can be done by use of CTRL/D, and on Windows by clicking the close button on the SQLC window; on all platforms, though, the ‘exit’ command can be used:

```
SQL> exit;
```

Further illustrations of SQL syntax can be found in section 3.4 of this document.

2.5.5 Demo 1d – simple clients

A database is only useful when data can be inserted into and retrieved from it under program control. To this end we have provided a selection of clients to illustrate this. To start the clients, first start a command prompt. You do not need to change directory to the current demo directory.

2.5.5.1 Querying the database with a client

Run the supplied “query” client to perform a simple, static query - this is a query that returns a single set of results and then terminates:

```
% query
```

The query client performs the SQL query, the results are returned to the client, which outputs them to the standard output. The query executed is "select * from currency" - the output should be:

```
Connecting to database at 8001...
Connected, now launch query...
Currency code: GBP: 1 US Dollar buys 0.50 UK Pound.
Currency code: EUR: 1 US Dollar buys 0.81 Eurozone Euro.
Currency code: CHF: 1 US Dollar buys 1.67 Switzerland Franc.
Currency code: CAD: 1 US Dollar buys 1.49 Canada Dollar.
Currency code: AUD: 1 US Dollar buys 1.72 Australia Dollar.
Currency code: JPY: 1 US Dollar buys 109.50 Japan Yen.
All rows returned.
```

Use SQLC to modify the values of some of the currencies by entering update statements. For example:

```
SQL> update currency set usdollar=0.75 where code='EUR';

SQL> commit;
```

and check that changes have happened using the "query" client.

```
Connecting to database at 8001...
Connected, now launch query...
Currency code: GBP: 1 US Dollar buys 0.50 UK Pound.
Currency code: EUR: 1 US Dollar buys 0.75 Eurozone Euro.
.
.
.
All rows returned.
```

2.5.5.2 Changing data using a client

There is a supplied client called "transact" that allows the user to perform transactions (changes) on the database. It takes as a parameter the SQL statement you wish to perform on the database, which may be any combination of insert, update and delete statements (separated by semicolons).

For example, to insert a new row into the currency table you may enter:

```
% transact "insert into currency values ('KRW','Korea','Won',1012)"
Connecting to port 8001...
Connected successfully. Start the transaction:
insert into currency values ('KRW','Korea','Won',1012)
Transaction completed successfully.
```

The "query" and "transact" clients form the basis of using the database from a client written in C or C++.

Start the supplied "animator" client:

```
% animate
```

This will periodically update the usdollar column of the currency table, weighted to keep within 10% of the values in the database at the time the client started.

As the "animate" client creates a significant amount of output it would be advisable to start a new command prompt in order to run other clients. Start a new command prompt and run the "query" client several times in succession to see the changing values. You will see the values in the usdollar column are changing.

```
% query
Connecting to database at 8001...
Connected, now launch query...
Currency code: GBP: 1 US Dollar buys 0.54 UK Pound.
Currency code: EUR: 1 US Dollar buys 0.78 Eurozone Euro.
Currency code: CHF: 1 US Dollar buys 1.79 Switzerland Franc.
Currency code: CAD: 1 US Dollar buys 1.60 Canada Dollar.
Currency code: AUD: 1 US Dollar buys 1.84 Australia Dollar.
Currency code: JPY: 1 US Dollar buys 117.30 Japan Yen.
Currency code: KRW: 1 US Dollar buys 1084.05 Korea Won.
All rows returned.

% query
Connecting to database at 8001...
Connected, now launch query...
Currency code: GBP: 1 US Dollar buys 0.50 UK Pound.
Currency code: EUR: 1 US Dollar buys 0.74 Eurozone Euro.
Currency code: CHF: 1 US Dollar buys 1.63 Switzerland Franc.
Currency code: CAD: 1 US Dollar buys 1.49 Canada Dollar.
Currency code: AUD: 1 US Dollar buys 1.68 Australia Dollar.
Currency code: JPY: 1 US Dollar buys 109.40 Japan Yen.
Currency code: KRW: 1 US Dollar buys 990.24 Korea Won.
All rows returned.
```

NOTE:	**Ensure that all clients are stopped (using Control-C) before moving on to the next demonstration.**

2.5.6 Demo 1e – Active Queries

Polyhedra has a unique “active query” feature that allows a client to launch a query in a fashion similar to a simple, static query as we have seen above. However, with an active query, once the initial set of results has been returned, the query is not finished. The database will continue to send changes to that set of results up to the client, so that the client is kept up to date with the changes to data in the database.

These changes are called "deltas" - a single delta may contain information about several changes if many have occurred since the last delta. (For example, many rows updated).

To start the active query demo client, enter:

```
% active
```

at the command prompt. This program runs until it is killed either by the user, or until the database is stopped.

```
Connecting to 8001...
Connected, now launch active query...
Row Added - Code: GBP, 1 Dollar buys: 0.670000.
Row Added - Code: EUR, 1 Dollar buys: 0.810000.
Row Added - Code: CHF, 1 Dollar buys: 1.670000.
Row Added - Code: CAD, 1 Dollar buys: 1.490000.
Row Added - Code: AUD, 1 Dollar buys: 1.720000.
Row Added - Code: JPY, 1 Dollar buys: 109.500000.
Row Added - Code: KRW: 1 Dollar buys: 990.24.
Delta complete - success.
```

The initial result set is returned, and changes to it are displayed in real-time. The database is responsible for updating the client with this information - the client does not poll the database for changes.

Use SQLC or the "transact" client to change the data in the database, using update, insert and delete statements - note that as soon as changes are committed, the active query client is informed by the database, and the change is displayed. For example:

```
SQL> update currency set usdollar=1.0 where code='EUR';

SQL> commit;
```

… will cause the active client to output:

```
Code: EUR, 1 US Dollar now buys 1.000000.
Delta complete - success.
```

We can also delete records, for example:

```
SQL> delete from currency where code='CAD';

SQL> commit;
```

… will cause the active client to output:

```
Currency CAD removed from resultset
Delta complete - success.
```

Doing several operations in a transaction will cause a "delta" to be sent to the client that contains all the changes:

```
SQL> update currency set usdollar=1.5 where code='AUD';

SQL> insert into currency values ('SEK','Sweden','Krona',9.77);

SQL> insert into currency values ('INR','India','Rupee',55.46);

SQL> commit;
```

… will cause the active client to output:

```
Row Added - Code: INR, 1 Dollar buys: 55.460000.
Row Added - Code: SEK, 1 Dollar buys: 9.770000.
Code: AUD, 1 US Dollar now buys 1.500000.
Delta complete - success.
```

Now start the animator client, which changes the data in the database on a continual basis. As it does so, note how the active query client displays the changes as they are received. Also, note that a single "delta" may contain changes to several rows.

```
% animate
```

The output from the active client will be something like:

```
Code: GBP, 1 US Dollar now buys 0.676700.
Code: EUR, 1 US Dollar now buys 0.826200.
Code: CHF, 1 US Dollar now buys 1.686700.
Code: AUD, 1 US Dollar now buys 1.515000.
Code: JPY, 1 US Dollar now buys 110.595000.
Code: KRW: 1 US Dollar now buys 990.24.
Code: SEK, 1 US Dollar now buys 9.867700.
Code: INR, 1 US Dollar now buys 57.123800.
Delta complete - success.

Code: GBP, 1 US Dollar now buys 0.683400.
Code: EUR, 1 US Dollar now buys 0.834300.
Code: CHF, 1 US Dollar now buys 1.703400.
Code: AUD, 1 US Dollar now buys 1.530000.
Code: JPY, 1 US Dollar now buys 111.690000.
Code: KRW: 1 US Dollar now buys 991.34.
Code: SEK, 1 US Dollar now buys 9.965400.
Code: INR, 1 US Dollar now buys 57.678400.
Delta complete - success.
```

2.5.7 Demo 1f: stopping a database

To close down a database in an orderly fashion, from SQLC enter the command "shutdown" (followed, as usual, by a semicolon):

```
SQL> shutdown;
```

On receipt of this command, the RTRDB will ensure the current state of the database is saved to file, and then close all its client connections in an orderly fashion. When the database is restarted, the file will be read and the contents at the time of the shutdown restored.

Alternatively you may just kill the rtrdb process - in this case, however, if you are using a standard RTRDB, the contents of the database are NOT saved.

2.5.8 Conclusions

In this sequence of exercises we have seen how to start a simple database, and how to use SQL with it to query and change the data in the database. We have also seen that simple clients written in C++ can access the database, and perform SQL on it in much the same way as a user can using the SQLC client.

We have also seen the Active Query - one of the unique and most powerful features of Polyhedra. Here, our C client was able to find out what data was in the database, and then be informed of changes to that data. These changes were forwarded to the client by the database without the need for the client to poll the database for the information. Finally, we have seen how a database can be stopped.

2.6 Demo 2 – an inherited table

2.6.1 Description and Aims

Polyhedra is an object-relational database. Clients to the database see a relational database with tables, rows columns, keys, views etc. Internally tables map to object-oriented classes, rows in the tables map onto objects of the class, and columns of the table map onto attributes of the class. Methods are provided by a programming language called CL (Control Language), which we cover in Demo 5.

To allow use of this object-oriented nature of the database, Polyhedra provides a mechanism for defining that a table (O-O class) is derived from another table (class). This is useful where you wish to have a parent table with common attributes that you want other tables to have access to. It is also extremely useful when coupled with the database Control Language (CL), which is the language used for specifying class behaviour.

In this simple example, we shall show how a derived table works. The database of the earlier demo has been extended by the addition of a table called “currency_limits”, derived from the parent “currency” table. This table contains all the columns from the parent table, and adds two new columns, being a high_limit and a low_limit. The SQL statements used to define the two tables were:

```
create table currency
( persistent
, code     large varchar primary key
, country  large varchar
, name     large varchar
, usdollar real
);

create table currency_limits
( derived from currency
, low_limit  real
, high_limit real
);
```

2.6.2 Starting the database

Start a command prompt and change directory to the main demo directory poly<version>\examples\demo_2. Start the database and an SQLC session by entering the following commands:

```
% rtrdb db

% sqlc sql
```

2.6.3 Demo 2a – how the derived table works

From the SQLC session select the data from the currency table and the currency_limits table.

```
SQL> select * from currency;

+-------+---------------+----------+----------+
| code  | country       | name     | usdollar |
+-------+---------------+----------+----------+
| 'GBP' | 'UK'          | 'Pound'  |     0.67 |
| 'EUR' | 'Eurozone'    | 'Euro'   |     0.81 |
| 'CHF' | 'Switzerland' | 'Franc'  |     1.67 |
| 'CAD' | 'Canada'      | 'Dollar' |     1.49 |
| 'AUD' | 'Australia'   | 'Dollar' |     1.72 |
| 'JPY' | 'Japan'       | 'Yen'    |    109.5 |
+-------+---------------+----------+----------+

SQL> select * from currency_limits;

+-------+-------------+----------+----------+-----------+------------+
| code  | country     | name     | usdollar | low_limit | high_limit |
+-------+-------------+----------+----------+-----------+------------+
| 'CAD' | 'Canada'    | 'Dollar' |     1.49 |       1.4 |        1.6 |
| 'AUD' | 'Australia' | 'Dollar' |     1.72 |       1.6 |       1.85 |
| 'JPY' | 'Japan'     | 'Yen'    |    109.5 |       102 |        117 |
+-------+-------------+----------+----------+-----------+------------+
```

See how the data in the currency_limits table is in a "child" table of the parent "currency" table. The parent "currency" table has in it only the columns that belong to the parent "currency" table, but has all the rows that are in the parent table and the child "currency_limits" table.

In the currency_limits table, only the rows that belong to the child "currency_limits" table are there - and the extra columns are shown.

Add a row to the parent table:

```
SQL> insert into currency values ('KRW','Korea','Won',1012);

SQL> commit;
```

and select from currency and currency_limits again. Note that the new row has only appeared in the currency table.

```
+-------+---------------+----------+----------+
| code  | country       | name     | usdollar |
+-------+---------------+----------+----------+
| 'GBP' | 'UK'          | 'Pound'  |     0.67 |
| 'EUR' | 'Eurozone'    | 'Euro'   |     0.81 |
| 'CHF' | 'Switzerland' | 'Franc'  |     1.67 |
| 'KRW' | 'Korea'       | 'Won'    |     1012 |
| 'CAD' | 'Canada'      | 'Dollar' |     1.49 |
| 'AUD' | 'Australia'   | 'Dollar' |     1.72 |
| 'JPY' | 'Japan'       | 'Yen'    |    109.5 |
+-------+---------------+----------+----------+

+-------+-------------+----------+----------+-----------+------------+
| code  | country     | name     | usdollar | low_limit | high_limit |
+-------+-------------+----------+----------+-----------+------------+
| 'CAD' | 'Canada'    | 'Dollar' |     1.49 |       1.4 |        1.6 |
| 'AUD' | 'Australia' | 'Dollar' |     1.72 |       1.6 |       1.85 |
| 'JPY' | 'Japan'     | 'Yen'    |    109.5 |       102 |        117 |
+-------+-------------+----------+----------+-----------+------------+
```

Delete the row for Korean Won:

```
SQL> delete from currency where code='KRW';

SQL> commit;
```

now insert the row into the child "currency_limits" table:

```
SQL> insert into currency_limits values

   | ('KRW','Korea','Won',1012,1000,1020);

SQL> commit;
```

and select from both the currency and currency_limits tables again. Note that that new row appears in both the currency (parent) and currency_limits (child) tables. The data is not held twice - the child "currency_limits" table simply holds extra columns for its rows, visible only when viewing the currency_limits table rather than the base table.

```
+-------+---------------+----------+----------+
| code  | country       | name     | usdollar |
+-------+---------------+----------+----------+
| 'GBP' | 'UK'          | 'Pound'  |     0.67 |
| 'EUR' | 'Eurozone'    | 'Euro'   |     0.81 |
| 'CHF' | 'Switzerland' | 'Franc'  |     1.67 |
| 'CAD' | 'Canada'      | 'Dollar' |     1.49 |
| 'AUD' | 'Australia'   | 'Dollar' |     1.72 |
| 'JPY' | 'Japan'       | 'Yen'    |    109.5 |
| 'KRW' | 'Korea'       | 'Won'    |     1012 |
+-------+---------------+----------+----------+

+-------+-------------+----------+----------+-----------+------------+
| code  | country     | name     | usdollar | low_limit | high_limit |
+-------+-------------+----------+----------+-----------+------------+
| 'CAD' | 'Canada'    | 'Dollar' |     1.49 |       1.4 |        1.6 |
| 'AUD' | 'Australia' | 'Dollar' |     1.72 |       1.6 |       1.85 |
| 'JPY' | 'Japan'     | 'Yen'    |    109.5 |       102 |        117 |
| 'KRW' | 'Korea'     | 'Won'    |     1012 |      1000 |       1020 |
+-------+-------------+----------+----------+-----------+------------+
```

Now change the value of the usdollar column for a row that is in the currency_limits table:

```
SQL> update currency set usdollar=1018 where code='KRW';

SQL> commit;
```

… and select from the currency and currency_limits tables. Note that the new usdollar value is shown in both tables.

```
+-------+---------------+----------+----------+
| code  | country       | name     | usdollar |
+-------+---------------+----------+----------+
.
.
| 'JPY' | 'Japan'       | 'Yen'    |    109.5 |
| 'KRW' | 'Korea'       | 'Won'    |     1018 |
+-------+---------------+----------+----------+

+-------+-------------+----------+----------+-----------+------------+
| code  | country     | name     | usdollar | low_limit | high_limit |
+-------+-------------+----------+----------+-----------+------------+
.
.
| 'JPY' | 'Japan'     | 'Yen'    |    109.5 |       102 |        117 |
| 'KRW' | 'Korea'     | 'Won'    |     1018 |      1000 |       1020 |
+-------+-------------+----------+----------+-----------+------------+
```

Now change it from the currency_limits table:

```
SQL> update currency_limits set usdollar=1005 where code='KRW';

SQL> commit;
```

… and select from the currency and currency_limits tables again. Note that again, the new usdollar value is present in both tables.

```
+-------+---------------+----------+----------+
| code  | country       | name     | usdollar |
+-------+---------------+----------+----------+
.
.
.
| 'KRW' | 'Korea'       | 'Won'    |     1005 |
+-------+---------------+----------+----------+

+-------+-------------+----------+----------+-----------+------------+
| code  | country     | name     | usdollar | low_limit | high_limit |
+-------+-------------+----------+----------+-----------+------------+
.
.
| 'KRW' | 'Korea'     | 'Won'    |     1005 |      1000 |       1020 |
+-------+-------------+----------+----------+-----------+------------+
```

This feature is particularly useful if combined with CL code (Demo 5) where the methods that are attached to the parent table can be overridden by methods attached to the child table. This allows rows that are entered in different tables to have different behaviours, whilst maintaining behaviour that is common to the core set of columns.

2.6.4 Demo 2b – using the derived table with a simple client

We have provided a client that uses the active query mechanism to check when a currency has transgressed one of the limits and raises an alarm.

At this stage in the demo, three of the currencies have been created into the parent "currency" table, and four into the child "currency_limits" table. All are therefore 'currency' objects, and so SQL 'select' and 'update' statements on the currency table will affect them all, but there is also the extra functionality of the child table: if you 'select' from the currency_limits table, then you will see the extra columns, and only the four rows that were inserted into the currency_limits table.

The supplied client application called 'monitor' uses an active query to monitor the currency_limits table. Note that it only reports the four rows that are in the currency_limits table; it does not see the records that are only members of the superclass, 'currency'.

Start the monitor client:

```
% monitor

connecting to database at 8001...
Connected, now launch active query...
Row Added - Code: CAD, 1 Dollar buys: 1.49. Limits - Low: 1.40 High: 1.60
Row Added - Code: KRW, 1 Dollar buys: 1005.00. Limits - Low: 1000.00 High: 1020.00
Row Added - Code: JPY, 1 Dollar buys: 109.50. Limits - Low: 102.00 High: 117.00
Row Added - Code: AUD, 1 Dollar buys: 1.72. Limits - Low: 1.60 High: 1.85
```

Use SQLC, or the transact client to modify the data in the currency or currency_limits table and see how the monitor client reports the currency moving outside the limits specified in the limits table.

```
SQL> update currency_limits set usdollar=100 where code='JPY';

SQL> commit;

*** ALARM *** : JPY has dropped below the low alarm limit 102.00 at 100.00.
```

Try using the animator client to animate the data, and note how when the currency values move outside the currency limits in the currency_limits table, the monitor client reports the transgression.

2.6.5 Conclusions

In these simple exercises we have used a table currency_limits that is derived from the main currency table to provide further functionality to the database. This particular example is somewhat contrived, but does show the way that a derived table behaves.

NOTE:	**Before moving on to the next demonstration, ensure that the database is closed down following the instructions in section 2.5.7. When the database stops, all connected clients will automatically stop.**

2.7 Demo 3 – Data Persistence

NOTE:	**This demo uses the journaling feature and so is not applicable to all editions of Polyhedra. Please refer to the feature.txt file for a list of features supported by each edition of Polyhedra.**

2.7.1 Description and Aims

Polyhedra is primarily an in-memory database. However, it does provide the user with a mechanism to save data to disk, enabling the database to be stopped and later restarted, and to provide a degree of recovery from disaster. The data is saved in a "snapshot file", which contains a copy of the current schema and a copy of the records that were present at the time the snapshot was generated. To be more accurate, it contains a copy of the 'persistent' data, since in Polyhedra it is possible when defining the schema to flag tables - and individual attributes of tables, subject to certain rules for consistency - as 'persistent' or 'transient'; transient data is not preserved over shutdown and restart.

There are two mechanisms by which data persistence is achieved in a running system:

1 Saving the database regularly (or when critical changes have occurred), providing checkpoints of the state of the database;

2 Using the database to provide continual journaling to disk of changes to persistent data.

In the following exercises we show both of these mechanisms at work, and discuss the relative merits of them.

This demonstration is split into two parts, showing the different mechanisms that can be employed to perform data persistence. Each of these has a slightly different configuration to start the database, and so the database will be stopped and started two times for the demonstration.

2.7.2 Demo 3a – data persistence by regular saving

Data may be persisted simply by saving the database regularly. This may be demonstrated purely from a database and an SQLC session. Data that is marked as "persistent" is saved to the load_file.

In this exercise, we shall use a table "trans_currency" - which is identical to the "currency" table, but the usdollar column is now transient. This means that the data in that column is not saved in the load file. This is probably a more accurate reflection of the real world - transient data is usually data that is not known at database load time, and is populated from the current state of the external system. We are fairly sure that when we start the database the British Pound will exist, but we do not know what its exchange rate is, so that would be populated from the real world when the database was started up.

Another argument for transient data is readings supplied from external sensors; when the database restarts, the values when the database stopped - maybe half an hour ago, say - are out of date, so it is better and safer to set the fields to a null value until more recent readings can be obtained and fed into the database.

The SQL to declare this table is as follows:

```
create table trans_currency
( persistent
, code     large varchar primary key
, country  large varchar
, name     large varchar
, usdollar real transient
);
```

2.7.2.1 Starting the database

Start a command prompt and change directory to the main demo directory for this demonstration - poly<version>\examples\demo_3. Start the database and an SQLC session:

```
% rtrdb db
```

```
% sqlc sql
```

2.7.2.2 Using the database

Having started the database, we shall make some changes to it using SQLC, and then save them. We will then restart the database to show how these changes we made have been persisted.

Firstly, check the state of the trans_currency table from the SQLC session:

```
SQL> select * from trans_currency;
```

```
+-------+---------------+----------+----------+
| code  | country       | name     | usdollar |
+-------+---------------+----------+----------+
| 'GBP' | 'UK'          | 'Pound'  | NULL     |
| 'EUR' | 'Eurozone'    | 'Euro'   | NULL     |
| 'CHF' | 'Switzerland' | 'Franc'  | NULL     |
| 'CAD' | 'Canada'      | 'Dollar' | NULL     |
| 'AUD' | 'Australia'   | 'Dollar' | NULL     |
| 'JPY' | 'Japan'       | 'Yen'    | NULL     |
+-------+---------------+----------+----------+
```

Note that the usdollar column is NULL on every row. This is because this column is transient, and the data for it is not saved in the load_file. There is a file called "sensible.sql" containing a series of update statements which will set the usdollar column to a sensible value for each of the currencies. Use the "include" command to bring in the statements in the "sensible.sql" file:

```
SQL> include 'sensible.sql';
```

Check the state of the database again:

```
SQL> select * from trans_currency;

+-------+---------------+----------+----------+
| code  | country       | name     | usdollar |
+-------+---------------+----------+----------+
| 'GBP' | 'UK'          | 'Pound'  |     0.67 |
| 'EUR' | 'Eurozone'    | 'Euro'   |     0.81 |
| 'CHF' | 'Switzerland' | 'Franc'  |     1.67 |
| 'CAD' | 'Canada'      | 'Dollar' |     1.49 |
| 'AUD' | 'Australia'   | 'Dollar' |     1.72 |
| 'JPY' | 'Japan'       | 'Yen'    |    109.5 |
+-------+---------------+----------+----------+
```

Now let us add a new currency to the list:

```
SQL> insert into trans_currency values ('KRW','Korea','Won',1012);

SQL> commit;
```

and save the database:

```
SQL> save into 'test.dat';
```

This will have saved the current state of the database to a file called "test.dat". In the demo_3 directory, you will see that a new file called "test.dat.old" has been created - this is the load_file that we started the database from - the old load_file is always saved into a file with the extension ".old" for disaster recovery purposes.

At this point, do not shutdown the database, but kill the rtrdb task instead. To demonstrate what data has been persisted to disk, restart the database and an SQLC session from the command prompt:

```
% rtrdb db

% sqlc sql
```

Use the SQLC session to check what data survived across the shutdown:

```
SQL> select * from trans_currency;

+-------+---------------+----------+----------+
| code  | country       | name     | usdollar |
+-------+---------------+----------+----------+
| 'GBP' | 'UK'          | 'Pound'  | NULL     |
| 'EUR' | 'Eurozone'    | 'Euro'   | NULL     |
| 'CHF' | 'Switzerland' | 'Franc'  | NULL     |
| 'CAD' | 'Canada'      | 'Dollar' | NULL     |
| 'AUD' | 'Australia'   | 'Dollar' | NULL     |
| 'JPY' | 'Japan'       | 'Yen'    | NULL     |
| 'KRW' | 'Korea'       | 'Won'    | NULL     |
+-------+---------------+----------+----------+
```

Note that the new currency - the Korean Won - has survived across the restart, as the code, country and name columns are all persistent. The values in the usdollar column were not saved, as that column is transient.

NOTE:	**Ensure the database is shut down before moving on to Demo 3b, by killing the rtrdb task or issuing the 'shutdown' command at the SQLC prompt.**

2.7.3 Demo 3b – data persistence

The rtrdb has the functionality to provide continual journaling to disk in a non-blocking fashion of the changes to persistent data. This allows the database to be restarted following an uncontrolled shutdown (e.g. power failure) and brought back up to the state it was in just before the failure (except for the transient data, of course).

The database load_file has appended to it a series of journal records. As a transaction completes (that affects persistent data), a journal record is created for it, and this is appended to the end of the load_file when the database has the opportunity to do so.

The load_file grows over time, as each transaction that changes persistent data causes a journal record to be appended to the end of this file on disk. Under control of a client a new snapshot may be taken at any time, and typically this occurs periodically or when the load_file has grown to an unacceptably large size. A special 'journalcontrol' table in the database with a single row records the size of the load_file in one of its attributes, and this can be monitored by an active query if

you wish; alternatively, it is possible to attach CL code to this table so that no external client is needed to trigger the new snapshots.

This exercise uses the standard demo schema, defining the table called currency with all attributes persistent.

2.7.3.1 Starting the database

To start the database for this exercise, start a command prompt and change directory to the main demo directory for this demo - poly<version>\examples\demo_3. Start the database and an SQLC session:

```
% rtrdb db_persist
% sqlc sql
```

Note the change to the command used to fire off the rtrdb; the rtrdb uses the argument in conjunction with the contents of the poly.cfg file to pick up configuration information applicable to this run of the database. In this case, there is an extra parameter in poly.cfg instructing it to enable the journaling mechanism.

2.7.3.2 Using the database

Confirm the initial state of the database by selecting from the currency table:

```
SQL> select * from currency;
```

In this demo, we are again using the standard “currency” table, where all columns are persistent, so the usdollar attributes are only null if they were null when the database was saved.

We shall use the animator client to demonstrate how the load_file on disk grows in size as transactions are performed on the database. Start the animator client:

```
% animate
```

Watch the size of the file on the disk, it will grow over time (some operating systems, such as Windows Vista, might cache writes to disk so that the file may appear to grow only sporadically). It will keep growing until a "save" is issued. To demonstrate this, from the SQLC session, issue a "save" statement:

```
SQL> save;
```

You will note that on disk, the load_file was renamed to "test.dat.old", and new file called "test.dat" was created. This new file is considerably smaller than the old load_file, as it contains just the current state of the database.

In order to prevent the load_file growing indefinitely, you may use a client that monitors the size of the file and issues “save” statements when the file has got too large. We have provided a simple implementation of this in the client “persist”. This takes as a parameter the maximum size of the file. Whilst the animator is running, start the persist client with a suitable maximum file size, say 12K:

```
% persist 12000
```

If the test.dat file is already over 12000 bytes in length a “save” is issued immediately; as a result, the test.dat file is renamed to test.dat.old, and a new test.dat file is created. As this file grows in size over time, eventually it will be larger than the maximum file size of 12000, and another save into will be issued, and a new test.dat is created.

Try killing the database - when you restart the database it will read the test.dat file, gaining from it the initial schema and data, and then applying all the journal records in turn until it reaches the end of the file. The database will start at the position it was in directly before the process was killed.

2.7.4 Conclusions

We have seen how data persistence can be done in two ways using Polyhedra:

(1) The simplest mechanism is simply to save a snapshot of the current database to a disk file at regular intervals (or when a set of critical changes have been done). This provides a method of check-pointing the database to disk whenever it is deemed necessary by the application.

(2) More robustly, the database can be configured to save changes in a non-blocking fashion to persistent data by appending the changes to the end of the load_file on disk as time goes on. In this way the load_file contains an initial snapshot of the state of the database, and then a series of journal records appended to the end of the file containing all the changes to persistent data since the last snapshot.

We have also seen how a simple client can monitor the size of the growing load_file by using an active query on the journalcontrol table. When the file has grown too large, then the client issues a "save" statement to the database, and causes

a new snapshot of the current state of the database to be written to disk, ready to have journal records appended to the end of it as the database changes.

NOTE:	**Before moving on to the next demonstration, ensure that the database is closed down following the instructions in section 2.5.7. When the database stops, all connected clients will automatically stop.**

2.8 Demo 4 – Fault Tolerance

NOTE:	**This demo uses the fault-tolerance feature and so is not applicable to all editions of Polyhedra. Please refer to the feature.txt file for a list of features supported by each edition of Polyhedra.** **If using Polyhedra Flash DBMS, the optional FT module must be linked into the RTRDB to run this demonstration.**

2.8.1 Description and Aims

Polyhedra can be configured in a Fault Tolerant manner consisting of a master database that is active, and a "hot" standby database running in memory that is fed the results of transactions as they complete on the master. There is an arbitration mechanism where the RTRDB sends heartbeats to an arbitrator, and the arbitrator controls fail-over from the active database to the standby.

In this demo set we have supplied a simple arbitration mechanism. It is up to the customer to provide a full arbitration mechanism, as each application has different criteria for deciding when to fail-over from one board to another. Such a fail-over may depend on many more things than just the database, and determination of which board is master and which is standby may not be possible purely by a software solution.

2.8.2 Starting the databases

To start the fault tolerant configuration, the first process to be started is the arbitrator. This is started by changing to the demo_4 directory and entering:

```
cd demo_4
clc arbiter
```

This starts a simple arbitrator that will assume that the first database to start will be the master, and the second is the standby. It will cause a fail-over when it fails to receive a timely heartbeat from the RTRDB for the master database, provided that the standby is already running.

This CL-coded arbitrator will display debug messages when it starts, and throughout the life of the arbitrator. It will also display a message for each heartbeat that is received if the resource "heartbeat_debugs" is set in the 'arbiter' section of the poly.cfg file in the directory **poly<version>\examples\demo_4**:

```
heartbeat_debugs   = yes
```

will display a debug message for each heartbeat.

```
heartbeat_debugs   = no
```

will cause the arbitrator to not display a message for each debug. When the arbitrator is started it will display messages similar to the following:

```
Component clc (V09.00.0000/WIN32-i386)
Copyright (C) 1994-2015 by Enea Software AB
DEBUG: The time is: 01-JUL-2015 16:00:59.088
DEBUG: Arbitrator started.
DEBUG: Heartbeat interval is 5000000 microseconds.
DEBUG: Arbitrator server ready on port 7200
```

Start the master database:

```
% rtrdb db1
```

this starts the rtrdb process. The RTRDB will connect to the arbitrator and the arbitrator will display messages similar to the following.

```
DEBUG: First heartbeat
DEBUG: 16:01:45: Heartbeat from 127.0.0.1:8051
DEBUG: RTRDB connected from 127.0.0.1:8051
DEBUG: 16:01:50: Heartbeat from 127.0.0.1:8051
DEBUG: 16:01:55: Heartbeat from 127.0.0.1:8051
.
.
.
```

and the arbitrator will inform the RTRDB that it is to be the master. The rtrdb will then read the load_file (logdir1\test.dat) off the disk and instantiate it in memory.

Clients may now connect and perform transactions and queries.

The standby database may be started:

```
% rtrdb db2
```

Here the RTRDB is started, and it connects to the arbitrator to discover its master/standby status. The arbitrator will return a message indicating that the RTRDB is Standby, and will display output similar to the following:

```
DEBUG: First heartbeat
DEBUG: 16:02:06: Heartbeat from 127.0.0.1:8052
DEBUG: RTRDB connected from 127.0.0.1:8052
DEBUG: 16:02:11: Heartbeat from 127.0.0.1:8052
DEBUG: 16:02:15: Heartbeat from 127.0.0.1:8051
.
.
.
```

The RTRDB then connects to the master RTRDB and requests a copy of the active database's data. This is provided, and after completion the databases run as a fault tolerant pair.

2.8.3 Demo 4a

In this demonstration we shall show how a client remains connected to the fault tolerant pair across a fail-over.

Start an SQLC session - this is a fault tolerant connection to the databases:

```
% sqlc sql_ft
```

Select from the currency table:

```
SQL> select * from currency;
```

Add a new row to the currency table:

```
SQL> insert into currency values ('KRW','Korea','Won',1012);

SQL> commit;
```

Now fail the master database. This may be achieved by just killing the active rtrdb process.

The master database will fail. The arbitrator will notice the failure almost immediately, and sends a message to the standby database telling it to become active. It also displays debug messages similar to the following:

```
DEBUG: Active RTRDB 127.0.0.1:8051 has disconnected.
DEBUG: Send 127.0.0.1:8052 active.
DEBUG: 16:05:11: Heartbeat from 127.0.0.1:8052
.
.
.
```

The SQLC session will invisibly switch its connection to the new active database.

Perform a select from the currency table:

```
SQL> select * from currency;
```

You will note that the new row just added is there - the change in data had been passed over to the standby database.

As there is now only one database running, you will need to start a new standby database. From the command prompt, restart the first database - it will connect to the current active database and start up as a standby:

```
% rtrdb db1
```

The arbitrator will receive the connection from the new rtrdb, and will display debug messages similar to the following:

```
DEBUG: First heartbeat
DEBUG: 16:10:42: Heartbeat from 127.0.0.1:8051
DEBUG: RTRDB connected from 127.0.0.1:8051
DEBUG: 16:10:46: Heartbeat from 127.0.0.1:8052
DEBUG: 16:10:47: Heartbeat from 127.0.0.1:8051
DEBUG: 16:10:51: Heartbeat from 127.0.0.1:8052
DEBUG: 16:10:52: Heartbeat from 127.0.0.1:8051
.
.
.
```

2.8.4 Demo 4b

This demonstration assumes that you have just completed Demo 4a.

In this demonstration we shall use the animator to show how the database and the clients keep on functioning across a fail-over. We will also see how active queries continue in an uninterrupted way across a fail-over.

Both the animator client and the active query client can be set up to connect to a fault tolerant pair of databases. This is done by specifying the data services of the active and standby databases on the command line, and the option "FT". The only change required to the client is that it specifies both data services to which it may wish to connect, and sets the appropriate flags to make the connection fault tolerant.

To start the animator such that it is connected to the fault tolerant pair of databases the command is:

```
% animate 8001,8002 FT
```

and start the active query client such that it is connected to the fault tolerant pair:

```
% active 8001,8002 FT
```

The output from the active query client shows the changing values in the database.

Try failing the active database. You will see a momentary pause as the fail-over takes place, and the standby takes over and becomes active. The clients automatically reconnect to the new database and carry on working.

You will note that the active query carried on running without a hitch (or at worst a momentary pause). The fault tolerance mechanism allows active queries to be re-launched on the new active database, and the client to be informed as data changes. The client does not need to do any special programming to handle the fail-over.

Having killed the rtrdb, there is only one database running. To simulate the replacement of the failed board start the other rtrdb up again:

```
% rtrdb db2
```

Note that it comes up as a standby. The active database may now be failed and the new standby will take over. You may repeat this as often as you wish.

2.8.5 Conclusions

In these demonstrations we have shown how to run up a fault tolerant configuration, using a very simplistic arbitration mechanism. There is a fuller discussion of the arbitration mechanism in the Further Directions section, and a discussion of the set up of a fault tolerant system where the active and standby databases are on different machines (as would be normal).

We have shown that by using Polyhedra's Fault Tolerant set up, a pair of databases are kept in step, with one as the active "master" database, and one as the standby. On failure of the master database (or the machine it is residing on), the standby takes over within moments. The standby changing to an active database is a mode change - and so there is no startup process required for the standby to become operational, it simply changes mode to be an active database.

Using a Polyhedra client API, the connection from a client to the database is also fault tolerant - when the active database fails, and the standby takes over, the client reconnects seamlessly to the new database. The user code does not even have to be informed of the fail-over. Active queries remain in place, and the client carries on working with the new database.

NOTE:	**Before moving on to the next demonstration, ensure that all clients, the arbitrator and both databases are closed down.**

2.9 Demo 5 – CL Code

NOTE:	**If using Polyhedra Flash DBMS, the optional CL module must be linked into the RTRDB to run this demonstration.**

2.9.1 Description and Aims

This demonstration is designed to show the CL programming language - a tool provided for object method scripting. This language runs inside the database as part of a transaction. It has direct access to the data and the appropriate primitives to work directly on the data. The language determines the behaviour of the data in the database, turning a simple, static data store into an active database where rows in a table are objects. The language is event driven, and responds to changes in the data in the database.

In this demo the database from demo 2 (inheritance) will be used. The original currency table now has a CL script attached to it that ensures that the usdollar column of a row can never be negative. If a user does attempt to put a negative value in this column, then the CL code will clamp the usdollar value to zero, and output a debug message indicating that this has happened. Debug messages go to the standard output of the database process - or the database window when running on Windows.

With CL, you are given the option of specifying object methods (behaviour) in RTRDB tables. CL provides a rich "English-like" syntax with over 44 statements and 120 function calls including pre-built scripts for Handlers, Procedures and Functions to support conditionals, switches, loops, procedures, functions, recursion, etc. CL provides full support for event handling from external, user interface and data change events.

The currency_limits table, derived from the currency table, has more complicated CL code that compares the usdollar value with the high_limit and low_limit values and sets a "status" column in the table to "OK", "HIGH" or "LOW". It also prints a debug indicating the new state. Additional code also checks that the limits are meaningful, rejecting transactions that break these conditions.

The CL code is in the file db.cl in the main demo directory. Each table (currency and currency_limits) has a ‘script’, a set of methods that are run when indicated events occur or when invoked by other CL code.

2.9.2 Starting the Database

Start a command prompt and change directory to the main demo directory - poly<version>\examples\demo_5. Start the database and an sqlc session:

```
% rtrdb db
```

```
% sqlc sql
```

The rtrdb in this demonstration will be producing output, and so it may be appropriate to arrange to start the sqlc session from a separate command shell (see section 2.4.3 for more details).

2.9.3 Demo 5a

In this exercise we will show how the CL script attached to the currency table clamps the usdollar value to zero if an attempt is made to change it to be negative.

From the SQLC session change the British Pound value to be negative:

```
SQL> update currency set usdollar = -2 where code = 'GBP';
```

```
SQL> commit;
```

Note that the rtrdb process outputs a debug as follows:

```
DEBUG: US Dollar value cannot be negative
```

Check the contents of the currency table to see that the value of -2 was clamped to 0:

```
SQL> select * from currency;
```

```
+-------+---------------+----------+----------+
| code  | country       | name     | usdollar |
+-------+---------------+----------+----------+
| 'GBP' | 'UK'          | 'Pound'  |        0 |
.
.
.
```

2.9.4 Demo 5b

In this exercise, we will show how the handler for the usdollar column changing is overridden in the child table "currency_limits". In addition, we will see how the CL code can make decisions about how to change data based on other values on the row.

Use the SQLC session to change the data in the currency_limits table:

```
SQL> update currency_limits set usdollar=1.9 where code='CHF';

SQL> commit;
```

Note the debug message from the rtrdb:

```
DEBUG: CHF is over high limit
```

Check the contents of the table:

```
SQL> select code, usdollar, low_limit, high_limit, status
   | from currency_limits;
+-------+----------+-----------+------------+--------+
| code  | usdollar | low_limit | high_limit | status |
+-------+----------+-----------+------------+--------+
| 'CHF' |      1.9 |      1.52 |       1.72 | 'HIGH' |
| 'CAD' |     1.49 |       1.3 |        1.7 | 'OK'   |
| 'AUD' |     1.72 |       1.3 |          2 | 'OK'   |
| 'JPY' |    109.5 |       102 |        118 | 'OK'   |
+-------+----------+-----------+------------+--------+
```

Note that even if we change the currency table, the database is aware that the row is actually in the child table, and fires the correct CL handler to deal with the change.

```
SQL> update currency set usdollar=98 where code='JPY';

SQL> commit;
```

Note the debug and contents of the table following this command:

```
DEBUG: JPY is under low limit
+-------+----------+-----------+------------+--------+
| code  | usdollar | low_limit | high_limit | status |
+-------+----------+-----------+------------+--------+
| 'CHF' |      1.9 |      1.52 |       1.72 | 'HIGH' |
| 'CAD' |     1.49 |       1.3 |        1.7 | 'OK'   |
| 'AUD' |     1.72 |       1.3 |          2 | 'OK'   |
| 'JPY' |       98 |       102 |        118 | 'LOW'  |
+-------+----------+-----------+------------+--------+
```

Finally, use the SQLC session to change the high limit in the currency_limits table to an invalid value:

```
SQL> update currency_limits set high_limit=1 where code='CHF';

SQL> commit;
```

Note the error message from the rtrdb, reported by the SQLC:

```
Error: invalid limits for CHF
```

The CL code has found the change does not satisfy its criteria, and so rejects the whole transaction, leaving the database unaltered.

2.9.5 Conclusions

In this sequence of exercises we have seen how the CL code can respond to changes in the database. It can use the data in the database to make decisions about what changes to make to data, and does this as part of the transaction. CL Code is a very powerful tool for enforcing application level integrity within your database (such as the "status" column in the currency_limits table), and can be used to affect data on rows other than the one being changed, and in other tables in the database. CL code can also be used to launch active and static queries on other Polyhedra databases.

This feature of Polyhedra is, in its simplest form, analogous to database triggers firing stored SQL procedures, but CL is far more powerful and can be used to do a great many more things than SQL procedures. There is a library of CL examples available from Polyhedra - please contact us for further details.

NOTE:	**Before leaving this demonstration set, ensure that the database is closed down following the instructions in section 2.5.7.**

2.10 Overall summary

During the whole series of demonstrations, you will have seen many of the features of Polyhedra. It is, in its simplest form, a relational database - but provides features far above and beyond this. The database itself is "active", the data held in tables is transformed into classes where the behaviour is specified by using the CL language. The database also provides the "active query" mechanism, which pushes changes to data up to the client and so avoids the need for it to poll.

Polyhedra provides a mechanism for efficient data persistence so that data survives across restarts of the database, and also for a Fault Tolerant configuration for high availability.

We have not in this demonstration suite illustrated the Historian module, which allows a history of changes to data to be written out to disk. This timestamped history may be queried by SQL to allow changes in data to be replayed, or analysed and trended. We have also not examined the other possible interfaces to the database - there are a Type 4 JDBC driver, an ODBC driver and an OLE-DB provider available.

Polyhedra can also be used to claim a TCP/IP socket that may be used to communicate directly with CL code running inside the database. In this way we have built web server software where the browser is connected directly to the database engine.

We hope that you will continue to evaluate Polyhedra - the next step is to consult the "Further Directions" section below for explanations of how to build your own databases and clients.

3. Further Directions

3.1 Introduction

Having looked at the Polyhedra suite through the various demonstrations supplied, you will no doubt be keen to build your own databases and clients. This section is designed to help you achieve this. It covers the following topics.

- Designing and building your own database
- Compiling the example clients and writing your own
- The differences between the ODBC and Callback APIs.
- The arbitration system for fault tolerant configurations
- The control of data persistence
- Training

3.2 Reference manuals

Polyhedra has a number of reference manuals, which can be downloaded from our web site by registered customers and evaluators. For those who wish to download the complete set, we make them available as a ZIP format file and contains most of the manuals in a compressed format. You will need a suitable utility - such as PKZip or WinZip on a windows platform, or unzip on a UNIX platform, to unpack these files into a set of .pdf files.

3.3 Building your own database

Building your own database breaks down into the following steps:

1) Design the database schema - the set of tables, etc. that make up the database. This is best done in a text file called, say, "tables.sql" rather than simply typing them in directly via the SQLC window, as this makes it easy to change at a later date. (As you will have seen, SQLC allows you to 'include' another file containing SQL statements as though you had typed them directly into SQLC.)
2) Set up initial data for your database. It is often the case that a database comes with some data already in place. This can be simply a database schema version number, or may be initial configuration for your system. Again, this is best created in a text file called something like "data.sql".
3) Create the database. Start the rtrdb process and an SQLC client, and include the files necessary to create your schema and insert your initial data. Save the database to a "load_file", from which you can load the database on any platform.
4) Write any CL code you have for your database.
5) Start the database, including the CL code and use SQLC to test that the CL code works as expected.
6) To deploy, you will almost certainly distribute only a single database load_file. To create this file, including any CL code you have written use the "save into 'name.dat' with cl" statement from SQLC. Saved databases are platform-independent, and so can be used on other machines even if they are running a different operating system or have a different hardware architecture.

In order to help you create your initial database, a directory called "template" has been included containing placeholder files for you to use. This directory includes the following files:

poly.cfg	Configuration file to allow you to start the database at the various stages in creating the database
tables.sql	Template file containing the schema definition
data.sql	Template file containing the initial data
db.cl	Template file containing CL code for the database

In order to create a database using these files start a command prompt, change directory to the template directory and follow the sequence detailed below. Note that this sequence will work with both the standard product and Polyhedra Flash DBMS. However, there is a shorter alternative when using Polyhedra Flash DBMS, which is also given below.

1. Start a blank database, an SQLC session, and include the tables.sql and data.sql to create the schema; save the database and shut it down:

```
% rtrdb empty
% sqlc sql
SQL> include 'tables.sql';
SQL> include 'data.sql';
SQL> save into 'test.dat';
SQL> shutdown;
```

2. This will create a file called test.dat. In order to start this database, with its initial schema and data, enter the following at the command prompt:

```
% rtrdb db
% sqlc sql
```

3. If you create CL code for this database, then it can reside in the file db.cl. The poly.cfg file contains an entry point that will allow you to start the database and have it read the CL file and compile the code. To start the database in this way, enter at the command prompt:

```
% rtrdb dbcl
% sqlc sql
SQL> save into 'with_cl.dat' with CL;
SQL> shutdown;
```

4. At this stage, 'with_cl.dat' will hold the same schema definition and data as 'test.dat' originally did, but will also have a copy of the machine-independent 'bytecode' into which CL is compiled.

When using Polyhedra Flash DBMS stages 1 and 2 in the sequence above can be replaced by:

- Start a database, an SQLC session, and include the tables.sql and data.sql to create the schema; shut down the database:

```
% rtrdb db
% sqlc sql
SQL> include 'tables.sql';
SQL> include 'data.sql';
SQL> shutdown;
```

NOTE:	**When using Polyhedra Flash DBMS do not execute a SAVE INTO statement specifying a filename matching the current load_file resource setting. Doing so would overwrite the database file currently in use by the RTRDB and cause undefined results.**

3.4 Some Sample SQL

For ease of reference, particularly for those without extensive experience of SQL, some sample SQL statements are given below, including some Polyhedra-specific extensions. For a full definition and description of the SQL language supported by Polyhedra, you should look at the Polyhedra SQL reference manual, sql.pdf.

3.4.1 Creating a table

```
create table x
( persistent
, id        integer primary key
, data      integer
, flag      bool
, value     real
, timestamp datetime
);
```

3.4.2 Creating a table with a multicolumn primary key

```
create table userarea
( persistent
, username large varchar not null
, area large varchar
, primary key (username, area)
);
```

3.4.3 Creating two tables referencing each other

```
create schema
  create table p
  ( persistent
  , id      integer primary key
  , partner integer references q
 )
  create table q
  ( persistent
  , id      integer primary key
  , partner integer references p
  )
;
```

3.4.4 Deleting a table

```
drop table y;
```

3.4.5 Creating a table and a view onto it

```
create table areatable
( persistent
, id      integer primary key
, area    large varchar
, message large varchar
);
create view alarm as
    select * from alarmtable where area in
           ( select area from areauser where username=GetUser () );
```

3.4.6 Creating a record

```
insert into userarea values ('Joe', 'machine room');
insert into userarea (name, area) values ('john', 'library');
```

3.4.7 Insert records according to data from another table

```
insert into userarea (name, area)
    select name, 'reception' from user;
```

3.4.8 Update records

```
update x set flag = true, timestamp=now() where value > 0;
update x set value =5;
```

3.4.9 Delete records

```
delete from x;
delete from userarea where name = 'john';
```

3.5 Compiling the example clients and writing your own

The example clients supplied with the demonstration suite are provided with all the source code that was used to build them. This is all located in the directory poly<version>\examples\clients. These directories are laid out as follows:

clientname

callback

Source code file

odbc

Source code file

The actual source code file is always named "<clientname>.cxx" or "<clientname>.c".

3.5.1 Compiling under Windows

Details of how to link Polyhedra clients and servers for Windows may be found in poly<version>\win32\doc.

3.5.2 Compiling under Unix

Compiling under Unix is system dependent, since (a) the command name to invoke the compiler can vary; and (b) different versions put certain operating system functions in different libraries. The form of the command line will also differ depending on whether you are compiling a C or C++ program, and on whether the client is using the Callback API or the ODBC API as different Polyhedra libraries need to be linked in.

Polyhedra provides global memory management routines to improve performance. These routines need not be included in the build, in which case the libmem.a library should be replaced with the libnomem.a library.

Full details of how to link Polyhedra clients and servers for Unix platforms may be found in poly<version>/<platform>/doc where <platform> is linux, solaris, etc.

3.5.2.1 Compiling C++ Callback API clients

On Linux, a client written in C++ and using the callback API can be compiled by a command such as…

```
% c++ active.cxx -o active polyposix.o -lclientapi -lmem -lpat -lpthread
```

… or you could have used g++ instead of c++, as on this platform they are synonyms. The client can be linked without Polyhedra memory management routines using the command…

```
% c++ active.cxx -o active polyposix.o -lclientapi -lnomem -lpat -lpthread
```

On certain Unix platforms, it may be necessary to link in the standard socket, ns or xnet libraries, such as in the following command for use on Solaris 9:

```
% g++ active.cxx -o active polyposix.o -lclientapi -lmem -lpat -lpthread -lsocket -lnsl
-lrt
```

If you do get a link error complaining about unsatisfied references, you can use the 'man' command to find out if the missing function is a standard one, supplied by the operating system. If so, the manual entry should not just tell you the header file in which the function is declared, but also the command line you need to compile and link a program using that command: copy across the '-l' library arguments to the end of the command line you are using to compile and link your client application.

3.5.2.2 Compiling C Callback API clients

If your callback client is written in C rather than C++, the command to invoke is cc or gcc rather than c++ or g++, since this will ensure the C form of the interface routine names are declared in our header files (which use the __cplusplus conditional compilation flag to work out which form of the declarations to include). Thus, on Linux you could say:

```
% cc dosql.c -o dosql polyposix.o -lclientapi -lmem -lpat -lpthread -lstdc++
```

Polyhedra itself is written in C++, and so when linking it is necessary to scan the standard C++ support library. When compiling and linking a C++ program that you have written, this is automatically scanned. On Solaris, you would say:

```
% gcc dosql.c -o dosql polyposix.o -lclientapi -lmem -lpat -lpthread -lsocket -lnsl -lrt
-lstdc++
```

See the preceding section on compiling C++ clients regarding additional libraries that may need to be linked in on some Unix platforms.

3.5.2.3 Compiling ODBC API clients

On Linux, a client written in C and using the ODBC API can be compiled by a command such as…

```
% cc active.c -o active polyposix.o -lpolyodbc -ldbbase -lmem -lpat -lpthread -lstdc++
```

… or you could have used gcc instead of cc, as on this platform they are synonyms. If you use the C++ compiler rather than the C compiler (by replacing the 'cc' command by 'g++', say) the C++ support library will always be scanned and

there is no need to mention it in the command line. On certain Unix platforms, it may be necessary to link in the standard socket, ns or xnet libraries, such as in the following, alternative command for use on Solaris 9:

```
% gcc active.c -o active polyposix.o -lpolyodbc -ldbbase -lmem -lpat -lpthread -lsocket
-lnsl -lrt -lstdc++
```

3.5.3 Compiling under OSE

The client source code for the examples clients does compile for OSE and can be used to build load modules (as provided), but we do not currently provide files to facilitate this.

Similarly we do not currently provide files for building the Polyhedra components RTRDB, CLC and SQLC. However we do provide an example Embedded Interface implementation as used by the pre-built load modules. This can be found in the file poly<version>/ose/examples/common/src/polyose.cxx.

More details of how to link Polyhedra clients and servers for OSE may be found in poly<version>/ose/doc.

3.5.4 Differences between the ODBC and Call-back APIs

When writing your own clients, each client may be either written using the Callback API or the ODBC API - never both, though you may use a mix of clients connecting to the same database.

The Callback API is the thinnest layer between the programmer and the underlying messages between the client and the server. It is entirely asynchronous - the client is never blocked waiting for the server to complete a task - driven by a scheduler that resides in the API.

Each library function is designed to perform an operation with the database and will take one or more functions as arguments, to be called when the operation completes, or has data for the application. These functions are called "callbacks".

The "main loop" of your program lies in a function called the User Function that is called repeatedly when the API has no other function to call.

The callback API makes no attempt to hold the data returned in a query. It merely passes the data up to the user application, and the application can hold as much or as little of this information as it likes, in whatever structures are appropriate to it.

The ODBC API uses the standard set of ODBC function calls. It is by default synchronous - each function blocks until the operation has completed and a result has been returned from the server. However, it allows the programmer to operate in an asynchronous mode, where function calls return immediately with a "Not Finished Yet" return code: it is then up to the programmer to repeatedly call the operation again to determine when the operation has completed. Polyhedra has added extra functions to the API to allow the asynchronous mode to be used without the necessity for polling, so that the main loop of your program can just wait for any event and then process all outstanding events before dropping back into waiting for new events.

Unlike the Callback API. the ODBC API holds the results of queries for the programmer, ‘client-side’. The programmer can access the query results by using cursors and the appropriate "Fetch" function calls. The fetch function calls do not involve a client-server interaction, as the result is already held by the library in the client’s address space.

In general, the ODBC API is more long winded, but easier to program in. The Callback API is more difficult to use if you are unfamiliar with the underlying model, but is lighter in terms of memory and the user functions are more directly coupled to the underlying client-server communications protocol.

3.6 Fault Tolerant systems

Within the demonstration suite, there is a fault tolerance demonstration. This uses a very simple arbitrator, which makes no attempt to replicate the environment of a "real" system. We provide template code to build a true arbitrator, and the protocol and arbitration system is discussed in an appendix to this document.

On the standby machine, a true arbitration system needs to be able to distinguish between a network failure, and the active machine dying. This can be very hard, and is frequently only possible to do with an arbitrator that is integrated with the hardware. Furthermore, the arbitration mechanism that will need to be built for deployment will arbitrate which machine is master and which is standby. This will be based on much more than whether or not the database is running. The decision to switch over may be triggered by processes other than the database failing. As such it is always up to the customer to design and build their final arbitrator.

The poly.cfg file used to do the Fault Tolerant demonstration is rather complicated. This is because the file is effectively the configuration file for two copies of the database and the sql sessions for both, all running on the same machine (and so having to use distinct port numbers). The poly.cfg for one side of a fault tolerant pair would look more like this:

```
-------------------------
common:
data_service = 8001

db:common
type      = rtrdb
load_file = test.dat
journal_service       = 8051
other_journal_service  = $(OTHER_MC):8051
arbitrator_service = 7020

sql:common
type = sqlc
-------------------------
```

In the above, OTHER_MC would an environment variable that contains the IP address of the other machine in the fault-tolerant pair; alternatively, the address can be entered directly into the poly.cfg file.

3.7 Persistence Control Client - “persist”

NOTE:	**This section is not applicable to Polyhedra Flash DBMS which provides implicit data persistence.**

The data persistence control client “persist” that is used in Demo 3 is very simple. It merely launches an active query on the journalcontrol table, and uses the filesize attribute of that table to control when it issues the save command. This has a number of drawbacks:

- No effort is made to make regular snapshots of the data.
- No effort is made to schedule snapshots for "quiet" times of low database load.
- No effort is made to reduce the flood of deltas the client will receive if the database is handling a long series of transactions.

It would also be possible to code this client in CL - writing a CL script attached to the journalcontrol table which triggered the "save" when it was appropriate.

As such, the “persist” client should only be regarded as a starting point for building a proper persistence control client. The principle of monitoring the file size and issuing a "save" statement when the file size has passed some boundary is still correct though.

3.8 Training

Polyhedra provides training at various levels. We offer a standard introductory four-day course, which covers all aspects of the Polyhedra suite to a limited level; when provided on-site, it can be tuned to your needs, and may have a consultancy element. We can also provide training for specific elements of the system to a much deeper level.

We also provide consultancy where necessary, for example to advise on the use of Polyhedra, to assist with the design of the Polyhedra-specific part of your application, or to perform a design or code review.

4. Appendix: An arbitration mechanism for Polyhedra.

4.1 Background: Fault Tolerance in Polyhedra

4.1.1 The need for fault tolerance

Database systems supporting next-generation wireless and broadband network infrastructure demand the same reliability as proprietary systems in the traditional telecom and utilities industries. To meet this challenge, Polyhedra has created the leading solution for providing transaction database service on a continuous basis using standard hardware and software components without the need for proprietary application development. Polyhedra fault tolerance provides the highest level of availability to ensure client transactions are executed without interruption. It does this by providing a complete and flexible set of functionality for instant fail-over database service to a standby database server that maintains client application connections as well as transactional reliability. A continuous database service is provided, with switchover - whether as a result of board failure or on request - taking just a fraction of a second.

It is important to note that database replication, while supported by Polyhedra, does not of itself provide fault-tolerant database fail-over; the critical ability to automatically provide continuous database service to client operations, in the event of primary database failure, at machine speed, and without human intervention. More importantly, database replication by itself cannot continue to provide database service without losing client connections or transactions in the process. In terms of real-time database processing, sub-second switchover is vital.

Already at the core of some of the world's most advanced and important Third Generation Wireless, IP network infrastructure, and industrial control systems, Polyhedra is setting and maintaining the standard for continuous database availability and ultra-high speed SQL database performance.

4.1.2 Fault-tolerant services

Database fault tolerance is provided by Polyhedra with a master and a standby database server configuration. The Standby database is kept up to date with the changes that occur in the master database, and so it is ready to go live as soon as it is told to do so - either as a result of a problem with the master database or because a controlled switch-over was requested. The roles of the databases can be switched at any time. Fail-over is accomplished at machine speed ensuring transactional reliability and continuous availability of a live database service to clients. Automatic re-synchronisation of data is accomplished in the background once a new Standby is brought into service.

4.1.3 Fault-tolerant clients

The Client APIs provide support for implementing fault tolerant clients. That is, clients that provide automatic, and imperceptible fail-over from alternative data services in a system configured to provide dual redundant databases. The fault tolerant features supported by the API are provided in a flexible manner. Memory usage by the API is an important consideration on many platforms and some features of fault tolerance require additional memory overhead. For this reason the fault tolerant features supported by the API are optional. The fault tolerant functionality supported is on a per (logical) client connection basis.

The Polyhedra Client APIs address the following aspects for implementing a fault tolerant client:

- Opening a fault tolerant client connection
- Fault tolerant transactions
- Obtaining information about the data service connection

Polyhedra Fault tolerance is platform-independent and essentially identical across supported platforms. It is even possible to have the master and standby databases on different types of computer, running different operating systems.

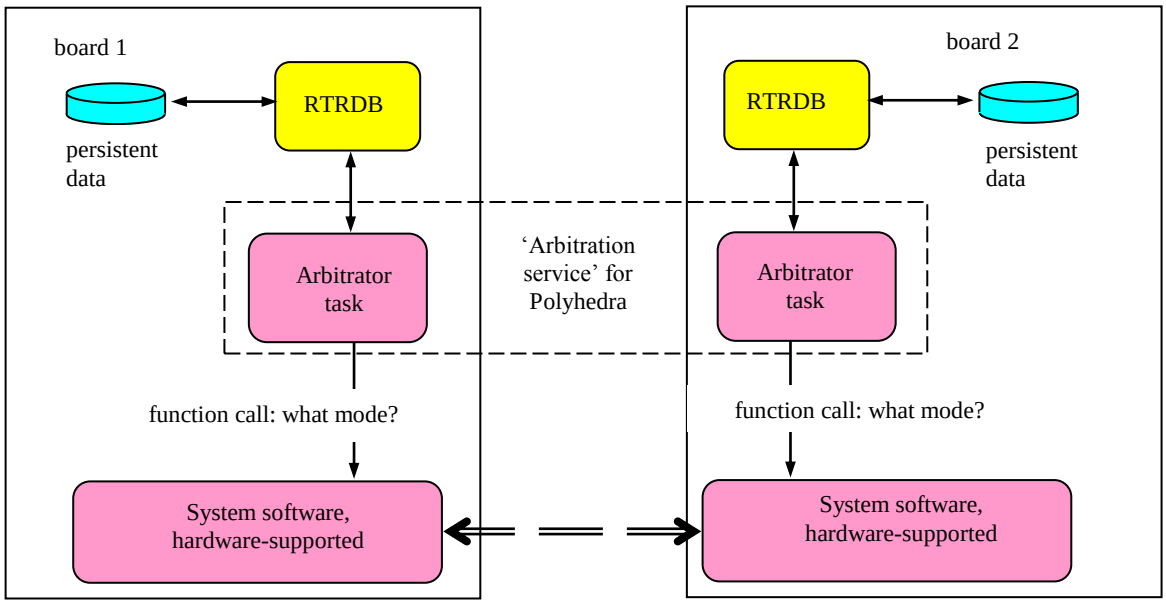
4.2 The need for arbitration

If you are developing a 2-machine fault-tolerant system, you will need a reliable way of detecting which of the two is 'master' at any time, and which one is 'standby' - for chaos can ensue if both machines consider themselves master. The decision method will vary widely; a common technique is to design boards with a hardware-supported watchdog

mechanism, whereby a software function can query a hardware latch to see if the board is master - if so, the software on the board is expected to call a watchdog routine every so often to prevent the latch tripping over (which would signal the partner board to become master).

Various other mechanisms are possible, each with their own advantages and disadvantages, so when one or more Polyhedra databases are running on a board it is not the role of Polyhedra either to supply a mechanism to determine whether the board is master or standby, nor should we dictate which mechanism should be used. Instead, the Polyhedra RTRDB component on each board expects that there is an arbitrator process/task to which it can connect via TCP/IP, and with which it can communicate to determine the board's master/standby status; the arbitrator tasks on the two machines between them provide an 'arbitrator service' to the Polyhedra RTRDBs, even though they might not directly communicate. The protocol of messages between the RTRDB and the arbitrator service is described in the last subsection this appendix.

This document describes an arbitrator written in C which is supplied as demonstration code, which calls a function to determine whether the board on which it runs should be master. A dummy implementation of the function is provided, which would have to be replaced for use on an embedded system. The architecture of the overall mechanism is as illustrated by the diagram below:



Each arbitrator task services the local RTRDB. The two arbitrators do not communicate with each other, and yet - due to the services provided by the system software (which itself will be relying on hardware mechanisms) - they are able to offer a reliable, distributed arbitration service to the RTRDBs. ('Reliable' here means that one will not get a situation whereby both RTRDBs think their database should be master, except perhaps for a brief moment when a switchover occurs.)

4.3 A simple arbitrator, using a system-supplied function

Suppose there is a system-supplied function named, for example, *CheckIfMaster()* which whenever it is called returns 0 on the board that is the standby, 1 if it is the master board, and which suspends until it knows if the machine does not yet know if it should be master. In this case, the arbitrator task is very simple. The routine takes one argument, a port number, and it claims this TCP/IP port to allow the RTRDB to connect to it. If the open fails, the routine can give up with a suitable return code; in other cases, it can drop into its main loop as described below.

4.3.1 The main loop

The main loop of the program calls a standard library function, *select* to wait for a connection attempt, an incoming message over an existing connection. When the *select* function returns, the program looks to see what woke it up. If it is a connection attempt, it accepts the connection and creates a little data structure to record the socket ID of the newly-opened connection. If it is an incoming message from an RTRDB, it finds out - by calling the *CheckIfMaster* function - whether the RTRDB should be master or standby, and sends the RTRDB a message telling it the mode in which it should be operating. If the arbitrator realises a connection has been closed by the other end, it simply closes its end of the connection and deletes

the associated data structure that it had created earlier. Once the reason the program had been woken up has been determined and appropriate action taken, the program goes back to the start of the loop and calls *select* again.

4.3.2 The *CheckIfAlive* implementation

In order to test operations in the absence of a general-purpose arbitration mechanism, the sample arbitrator program stores a character flag, and the *CheckIfAlive* checks this flag against the last character of the 'name' field in the messages from the RTRDB: if there is a match, the RTRDB is assumed to be master. This implementation has the benefit that the arbitrator program can be used in test environments where both databases are running on the same computer; typically, one might have one rtrdb announcing itself as db1 and the other as db2.

To test dynamic change of the master/standby status, the sample arbitrator described in this document has extra code to allow for a client to connect in via, say, telnet; the first letter of each incoming line is used to select a new mode.

4.4 Improving the arbitrator for real use

Before using this code in earnest in an embedded environment, some changes are needed, to remove code introduced to aid testing on Unix & Windows, and to integrate into the on-board support for determining board status. In particular, you should:

- remove the code in the main loop (shaded green in the listings below) that handles connections from tasks that are not RTRDBs;
- remove the code (again shaded green) for initialising the *board_mode* global variable; and, most importantly,
- remove the code (shaded red) that defines the *CheckIfMaster* function, replacing all instances of *CheckIfMaster* by the name of the system-supplied function that allows a task to find out the board status.

Various other improvements would be needed, such as making more use of non-blocking socket operations, and ensuring that all resources are freed whenever leaving the program. The code given here should be treated as a prototype.

4.5 Configuring Polyhedra to use the arbitrator

Assuming the arbitrator has been modified as indicated above to work on embedded systems, then the Polyhedra configuration file used on each of the two machines can simplify greatly (compared to the full version used in the fault-tolerant demos). For example, as they are running on separate machines, the RTRDBs can use the same journal service and data service numbers on each machine. In fact, the only difference between the configurations on the two machine is that each RTRDB knows to know the address of the machine running the other database. The file poly.cfg could simplify enormously:

```
DB:
suppress_dvi             = yes
suppress_log             = yes
type                     = rtrdb
load_file                = test.dat
data_service             = 7002
arbitrator_protocol      = tcp
arbitrator_service       = 7200
journal_service          = 7202
other_journal_service    = $(OTHER_MC):7202
```

One would EITHER substitute the correct IP address of the 'other' machine for *$(OTHER_MC)* on each machine, OR remove the line defining the *other_journal_service* resource from the poly.cfg file, and arrange to start each rtrdb with a command such as...

```
% rtrdb -r other_journal_service=10.1.2.101:7202 db
```

4.6 The program code

```
 1  #include <stdio.h>
 2  #include <sys/socket.h>
 3  #include <netinet/in.h>
 4  #include <netdb.h>
 5
 6  /* ------------------------------------------------------------------------ */
 7  /*                      C h e c k I f A c t i v e                           */
 8  /*                      -------------------------                           */
 9  /* define a function that will return information about the board mode;     */
10  /* in this simple example code, we check whether the last character of the  */
11  /* string supplied by the rtrdb in its heartbeat matches the global constant  */
```

```
12  /* board_mode. In a REAL implementation, we would call a system function to  */
13  /* determine the mode, ignoring the string supplied by the rtrdb.             */
14  /*                                                                            */
15  /*           REPLACE THIS DEFINITION on a real implementation.                */
16  /* -------------------------------------------------------------------------- */
17
18  unsigned char board_mode;
19
20  int CheckIfMaster (unsigned char c)
21  { return (c == board_mode);
22  }
23
24  /* -------------------------------------------------------------------------- */
25  /*                           s t r e a m h a n d l e                          */
26  /*                           -----------------------                          */
27  /* define a structure for recording connection info, and a function to send  */
28  /* a message to a nominated connection to tell it what mode it should be in. */
29  /* -------------------------------------------------------------------------- */
30
31  struct streamhandle
32  { int                  id;
33    char                 letter;
34    int                  tnum;
35    struct streamhandle* next;
36  };
37
38  void BuildAndSendMessage (struct streamhandle* sh)
39  {
40    unsigned char buf[13];
41
42    buf[0] = 'A';
43
44    /* mode */
45
46    if (CheckIfMaster (sh->letter))
47      { buf[1] = 1;
48        buf[2] = 0;
49        buf[3] = 0;
50        buf[4] = 0;
51      }
52    else
53      { buf[1] = 2;
54        buf[2] = 0;
55        buf[3] = 0;
56        buf[4] = 0;
57      }
58
59    /* transaction number - let's make byte ordering explicit! */
60
61    buf[5] = (sh->tnum      ) & 255;
62    buf[6] = (sh->tnum >>  8) & 255;
63    buf[7] = (sh->tnum >> 16) & 255;
64    buf[8] = (sh->tnum >> 24)      ;
65
66    /* heartbeat
67       here, 1 second (1,000,000 microseconds, hex 0F4240)
68    */
69
70    buf[ 9] = 0x40;
71    buf[10] = 0x42;
72    buf[11] = 0x0F;
73    buf[12] = 0x00;
74
75    write(sh->id, buf, 13);
76  }
77
78  /* -------------------------------------------------------------------------- */
79  /*                               m a i n                                      */
80  /*                               -------                                      */
81  /* on embedded platforms, do NOT define main in this program!                 */
82  /* -------------------------------------------------------------------------- */
83
84  int main (int argc, char* argv [])
85  { return arbitrator (argc, argv);
86  }
87
88  /* -------------------------------------------------------------------------- */
89  /*                         a r b i t r a t o r                                */
90  /*                         -------------------                                */
91  /* define the main work routine of this program, which prepares to listen    */
```

```
 92  /* for connections, and then waits for connection attempts, disconnections   */
 93  /* and messages.                                                             */
 94  /* ------------------------------------------------------------------------- */
 95
 96  int arbitrator (int argc, char* argv [])
 97  {
 98    int                  s;           /* connection socket  */
 99    short                p;           /* port number to use */
100    struct sockaddr_in   sa;
101    struct servent       *sp;
102    struct hostent       *hp;
103    fd_set               rd_fds;
104    struct streamhandle* streamchain = 0;
105
106    if (argc != 3)
107      { fprintf (stderr,
108                 "please give %s two arguments, a port and 0/1.\n",
109                 argv[0]
110                );
111        return 10;
112      }
113
114    board_mode = (argv[2])[0];
115
116    /* find out info about the socket I am supposed to claim. */
117
118    if ((sp = getservbyname(argv[1],"tcp")) == NULL)
119      { p = htons (atoi (argv[1]));
120        if (p == 0)
121          { fprintf(stderr, "service '%s' not registered on this machine\n",
122                    argv[1]);
123            return 10;
124          }
125      }
126    else
127      { p = sp->s_port;
128      }
129
130    /* port OK, so claim it and listen for connections. */
131
132    if ((hp = gethostbyname("127.0.0.1")) == NULL)
133      { fprintf(stderr, "can't get local host info\n");
134        return 10;
135      }
136    sa.sin_addr.s_addr = INADDR_ANY;
137    sa.sin_port        = p;
138    sa.sin_family      = hp->h_addrtype;
139
140    if ((s = socket(hp->h_addrtype, SOCK_STREAM, 0)) < 0)
141      {       /* failed to create socket. Give up. */
142        perror("Socket");
143        return 10;
144      }
145    if (bind(s, &sa, sizeof sa) < 0)
146      { /* failed to bind the port to the socket; give up, leaving O/S to
147           tidy up the socket (lazy, but OK on UNIX, Windows).
148        */
149        perror("Bind");
150        return 10;
151      }
152
153    listen (s, 5);
154
155    /*          -----------------
156                M A I N   L O O P
157                -----------------
158       reacting both to connection attempts and
159       also to traffic over open connections.
160    */
161
162    while (1)
163      {
164        int                 n;
165        char*               c;
166        int                 dbg = 0;
167        struct sockaddr_in  isa;
168        int                 i = sizeof isa;
169        unsigned char       buf[1000];
170        struct streamhandle* sh = streamchain;
171
```

```
172         /*            -------
173                       W A I T
174                       -------
175            wait for something to happen; on return
176            from the select, rd_fds will indicate
177            which stream(s) something happened on.
178         */
179
180         FD_ZERO(&rd_fds);
181         FD_SET(s, &rd_fds);
182
183         while (sh != 0)
184           { FD_SET(sh->id, &rd_fds);
185             sh = sh->next;
186           }
187         n = select(FD_SETSIZE, &rd_fds, NULL, NULL, NULL);
188
189         /* is there an incoming connection or message (or is a stream closing)? */
190
191         sh = streamchain;
192         while (sh != 0)
193           { if (FD_ISSET(sh->id, &rd_fds))
194               { n = read(sh->id, buf, 1000);
195                 if (n <= 0)
196                   { /* we were told there was something on this stream - but it
197                        is empty, so it must have been closed by the far end,
198                        or broken. Close our end of the connection.
199                     */
200
201                     close(sh->id);
202                     if (streamchain==sh)
203                       streamchain = sh->next;
204                     else
205                       { /* find the streamhandle that points at sh, and
206                            make its 'next' pointer skip over sh
207                         */
208                         struct streamhandle* sh2 = streamchain;
209                         while (sh2->next != sh)
210                           sh2 = sh2->next;
211                         sh2->next = sh->next;
212                       }
213                     free (sh);
214                     break;
215                   }
216                 else if (buf[0] == 'J')
217                   { unsigned char* str = buf+9;
218                     sh->letter = str[strlen(str)-1];
219                     sh->tnum   = ((buf[8] << 24) |
220                                   (buf[7] << 16) |
221                                   (buf[6] <<  8) |
222                                   (buf[5]      ));
223
224                     /* build up the response to the RTRDB, calling the
225                        function CheckIfMaster to determine if the
226                        rtrdb is to be told master or standby.
227                     */
228
229                     BuildAndSendMessage (sh);
230                   }
231                 else
232                   { /* to assist debugging and to allow mode swapping,
233                        allow for a connection asking me to swap mode.
234                        Thus one can connect in via, say telnet (by a
235                        command such as 'telnet localhost 7200') and
236                        the first letter of a line is significant
237                     */
238                     struct streamhandle* sh2 = streamchain;
239                     board_mode =  buf[0];
240
241                     /* tell all RTRDBS about the changed mode */
242
243                     while (sh2 != 0)
244                       { if (sh2->letter != 0)
245                           BuildAndSendMessage (sh2);
246                         sh2 = sh2->next;
247                       }
248                   }
249               }
250
251             /* move onto the next streamhandle, to check for input */
```

```
252             sh = sh->next;
253           }
254
255         /* is someone trying to connect? */
256
257         if (FD_ISSET(s, &rd_fds))
258           { int v;
259             if ((v = accept(s, &isa, &i)) < 0)
260               { perror("Accept");
261                 return 10;
262               }
263             sh = (struct streamhandle *) malloc (sizeof (struct streamhandle));
264             sh->id      = v;
265             sh->letter  = 0;
266             sh->next    = streamchain;
267             streamchain = sh;
268           }
269       }
270   }
```

4.7 The TCP-Arbitrator Protocol

When it starts up, if the arbitrator_protocol resource is set to 'tcp', then the RTRDB will attempt to connect to the port indicated by the arbitrator_service resource; if this connection attempt fails, the RTRDB stops. The RTRDB then sends a single message to announce itself and to prompt the arbitrator to tell it (a) the operating mode (standby, master) in which it should be running, and (b) the heartbeat interval between messages from the RTRDB to the arbitrator.

The messages from RTRDB to arbitrator are all of the form:

```
'J'     1 byte
<mode>  4 byte
<trans> 4 byte
<name>  C string, including null byte
```

The mode values indicate what mode the RTRDB thinks it ought to be running in:

```
0    Unknown
1    Master
2    Standby
```

The transaction number indicates the number of the most recently completed transaction. The name field is the journal service name of the RTRDB that has sent the message, NOT its data service.

Note that the byte ordering is fixed, regardless of the platforms being used: thus the message represented in hex by **4a02000000320100003a3732303100** can be broken down as follows:

```
4a           the letter J
02000000     mode 2: standby
32010000     transaction 306 (0x132)
3a3732303100 ":7201" as a C string
```

Likewise, the messages from the arbitrator to the RTRDB are also all of the same form:

```
'A'         1 byte
<mode>      4 byte
<trans>     4 byte
<interval>  4 byte (micro-seconds)
```

The mode values are as before...

```
1    Master
2    Standby
```

... except the arbitrator is not expected to set the mode to 0 (unknown). The transaction number is the latest transaction number reported by the RTRDB to the arbitrator. The interval field gives the maximum number of microseconds the RTRDB will wait after receiving a message from the arbitrator before it sends a heartbeat; a value of zero indicates that no heartbeat messages are to be sent by the RTRDB. As with messages from the RTRDB to the arbitrator, the least significant byte of the 4-byte fields appears before the other bytes of the field, so the message represented in hex by **4101000000320100004042 0f00** can be broken down as follows:

```
41          the letter A
01000000    mode 1: master
32010000    transaction 306, hex 00000132
40420f00    1 second (1,000,000 microseconds, hex 000f4240)
```

If the stream is closed or broken, the arbitrator is entitled to assume the RTRDB has stopped. If no heartbeat messages are received for some time - say, twice the heartbeat interval - an arbitrator service could assume that the RTRDB is dead - however, where there are other means of determining liveliness of the board, the arbitrator is at liberty to ignore the heartbeat messages or set the heartbeat interval to zero or to a very large value. The only requirements on the arbitrator are

- (a) that it is to confine itself to the form of messages described above,
- (b) it must respond to the initial message from the RTRDB,
- (c) it must avoid a build-up of unread messages,
- (d) it should send a message confirming the heartbeat interval, last known transaction number and required state at approximately the heartbeat interval (perhaps in response to the message from the RTRDB) so that the RTRDB knows it is still alive; and,
- (e) it must avoid closing the stream except where essential (for the RTRDB can treat this as a signal to stop immediately, though the RTRDB is entitled to re-open the connection if it wishes).

The arbitrator can send its messages whenever it wants - it does not to have to wait for a heartbeat from the RTRDB, for example, in order to signal a change in the operating mode.