Accessing Zen v13 from C on a Raspberry Pi Using the Btrieve Interface

A White Paper From

For more information, see our web site at
http://www.goldstarsoftware.com

Accessing Zen v13 from C on a Raspberry Pi Using Btrieve

Last Updated: June 2019

The Actian Zen database engine (formerly known as Actian PSQL) supports a wide variety of application programming interfaces (APIs) to access the data. Some of these interfaces leverage the power of SQL to access your data, while others use a lower-level interface, commonly known as the Btrieve API to provide the needed performance and flexibility.

Due to the simplicity of the Btrieve API (there is only one function call, with 7 parameters), you can trust that your applications written on one platform will be very similar, if not identical, to those written on another platform. This cross-platform compatibility is one of the primary reasons developer have chosen and continue to choose the Actian database environment.

This paper has been created for the Btrieve developer who is already familiar with another environment (such as DOS or Windows) who wants to now build his application written in the C language on the Raspberry Pi.

Note that while the code in Appendix A contains examples of a few operations for you, it does NOT explain the ins and outs of the Btrieve API itself. For more detailed information on the Btrieve API, please consult the on-line documentation, either in the PCC or from the SDK downloads.

Important Pre-requisites

The following pre-requisites should be in place before you start.

1) You should have a Raspberry Pi device and Raspbian operating system properly installed.
2) You should have the Zen Edge v13 database engine installed and running on the Raspberry Pi. You can run the (free) Community Edition, a purchased license, or a 30-day trial license.
3) You should have a user account on your Pi to use for development work, and it should have already been configured with the LD_LIBRARY_PATH and PVSW_ROOT variables.

If you need help with these steps, please check the Actian documentation or look for other white papers on the Goldstar Software web site.

Create a Working Folder for Your Application

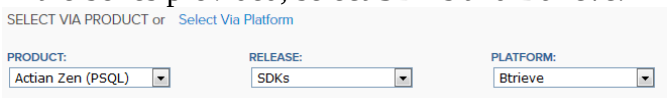
For better organization, you should create a Projects folder in your home directory, and then create a unique working directory for each application to house maintain all of the files and documentation for the project. You can either do this from the GUI screen on the Pi, or you can do it from a command line. In this example, we are creating a Projects folder in our home folder, along with a subfolder for the application, in this case StressTest:

```
bill@raspberrypi:~/Downloads $ cd ~
bill@raspberrypi:~ $ mkdir Projects
bill@raspberrypi:~ $ cd Projects
bill@raspberrypi:~/Projects $ mkdir StressTest
bill@raspberrypi:~/Projects $ cd StressTest
bill@raspberrypi:~/Projects/StressTest $
```

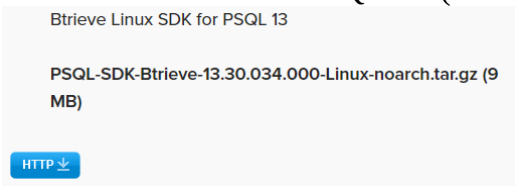
Downloading and Installing the Btrieve SDK

The next piece you need is the SDK download for Btrieve, which is easiest to get from the GUI:

1) Go to https://esd.actian.com/product/Zen_PSQL in a web browser.
2) In the boxes provided, select **SDKs** and **Btrieve**.



3) Scroll down and open up the link for **Btrieve**, then click on the **HTTP** button to download the Btrieve Linux SDK for PSQL 13. (Note that the version may change in the future.)



4) After it downloads, double-click the file to launch the Xarchiver utility, then click on the Extract Files button and click Extract to extract all of the files.
5) You will now have the Btrieve API components in a folder on your Pi. Although the actual folder name can change as the versions are updated, it is currently called `/home/bill/Downloads/PSQL-SDK-Btrieve-13.30.034.000-Linux-noarch`
6) Copy the contents of this new folder to your home folder using the GUI. If you prefer to use the command line for this, it would look like this:
```
cp ~/Downloads/PSQL-SDK-Btrieve-13.30.034.000-Linux-noarch/* ~ -r
```

Create Your C Application

If you already have a C application that you are porting, then you can bring that into your project folder now. If not, then you can use our sample StressTest code (in Appendix A) which provides an example of several key operations, including:

- Opening a File
- Loop While Reading the First Record for 10 Seconds
- Display Timing Results
- Closing the File and Reset the Database Connection

The easiest way to do this is to copy the text from the appendix, open a **nano** editor window, and paste in the sample code as-is. When you are done with this step, you should have the file StressTest.c in your project folder.

Before we compile and run the code, let's review a few important elements in this code:

- This testing tool opens up the database file provided on the command line, then reads the first record from the file (in physical order) with a StepFirst operation for a period of approximately 10 seconds. (An exact time is not required, because we're going to divide the total loops by the elapsed time anyway, which saves some overhead.)
- Two functions, *PrintTime* and *timediff*, are used to format the time for the display and to calculate the elapsed time.
- Lines 54 through 57 are used to set up the *BTRV* function parameters that will be used throughout the code.
- Line 58 handles opening the file in Normal mode. If the engine cannot open the file, an error is displayed and we quit immediately.

- Line 70-71 loop over 1000 **StepFirst** operations on the file. This reads the first record of the file 1000 times. While the first request may have to retrieve data from storage, subsequent calls are all handles by the Zen engine cache. Because of this, this test is a more true measure of ONLY the database calls, excluding storage performance or latency, so it is mostly dependent only on the CPU clock speed.
- Lines 72-73 compute the elapsed time in whole seconds. The calculation is not exact because we are NOT checking the nanoseconds field of the *timespec* structure. However, this is close enough for our purposes, as this is merely a benchmarking tool.
- Lines 75-79 handle displaying the results of the test. Note that *LoopCount* will always be an even multiple of 1000, but the *exacttime* value will be calculated now so that we can determine the actual throughput.
- Lines 81 and 82 close the file and terminate the connection to the database engine, releasing all locks, file handles, and other resources. You should **always** do this before your application terminates.

Create a Makefile for Your Application

Instead of reinventing the wheel, we recommend simply repurposing Actian's Makefile for your project.

1) First, copy the Makefile from the SDK into your folder.
   `cp ~/samples/btrieve/Makefile .`
2) Next, edit the file (using the **nano** editor) to change the Makefile in two sections.
   a. First, change the EXECS and OBJS line as shown here:
   ```
   EXECS = StressTest
   OBJS = StressTest.o
   ```
   b. Then, after the "all:" reference, remove the two old programs and add StressTest in the same format, as shown here:
   ```
   all: $(EXECS)

   StressTest:
   ```
3) Write the Makefile back to the disk (Ctrl-O/Enter) and then Exit (Ctrl-X) nano.

Build Your C Application

Now that all the prep work has been completed, compiling the application should be as simple as executing "make all" from a command line. On my system, I received a large number of "invalid string offset" messages, but these do not seem to have any impact, and the program successfully compiled.

Running Your C Application

To run your newly-built application, just run it. Note that Linux does not run applications from the current directory by default, so you will need to specify the current directory like this:

```
bill@raspberrypi:~/Projects/StressTest $ ./StressTest <filepath>
```

Be sure to provide a valid path to a database file. Here is a sample run using the Person.mkd file from the DEMODATA directory:

```
bill@raspberrypi:~/Projects/StressTest $ ./StressTest /usr/local/psql/data/DEMODATA/Person.mkd
StressTest (C)2019 Goldstar Software Inc.
Opening Data File /usr/local/psql/data/DEMODATA/Person.mkd...
18:51:03.952: Starting the test...
18:51:13.027: Testing Complete...
Loop Count = 211000
Process Time:  9.074 seconds
Iterations/s: 23252.524
```

This shows us that our lowly Raspberry Pi 3 is capable of over 23000 database operations per second!

Expanding Your Knowledge

Want to know if you understand what is going on? Try these modifications:

1) Change the program to read the first record by Key 1. Which line do you need to change? You have to change TWO parameters – did you get them both?
2) Experiment with longer or shorter elapsed times. Does this materially change the computation?
3) Try changing the inner loop size from 1000 to something else, like 10 or 10000. How does this impact the results?
4) If you are running this test on an Intel server that supports CPU Power Modes (typically High Performance, Balanced, or Power Saving) in the operating system or BIOS, how does changing this configuration value impact the test results?
5) As a benchmarking tool, we don't care about the data being read. Can you modify this application for your own table and display the first field from the data record inside the loop?

From here, you're only limited by your imagination!

Finding More Help

If you have other problems getting this to work, you can contact Actian directly through their web forums at https://communities.actian.com/s/ for more help. If you need some additional hand-holding, Goldstar Software may be able to assist you as well through our Developer Jump Start Program. You can contact us at 1-708-647-7665 or via the web at http://www.goldstarsoftware.com.

Appendix A: Sample StressTest Application

The following application makes use of the Btrieve API and can be used as instructional and sample code. (Blank lines have been removed to make it fit on one page.)

```
// Sample StressTest Application
// Written by Bill Bach
// From Goldstar Software Inc.
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <sys/timeb.h>
#include <time.h>
#include <sys/types.h>
#include <ftw.h>
#include "btrconst.h"
#include "btrapi.h"
// These variables are used to access the Btrieve API
BTI_BYTE posBlock[POS_BLOCK_SIZE];
BTI_BYTE dataBuffer[MAX_DATABUF_SIZE];
BTI_WORD dataLength;
BTI_BYTE keyBuffer[MAX_KEY_SIZE];
BTI_SINT keyNumber;
BTI_SINT StatusCode;

void PrintTime(char *text)
{
    struct timeb timebuffer;
    char *timeline;
    ftime( &timebuffer );
    timeline = ctime( & ( timebuffer.time ) );
    printf("%.8s.%.3hu: %s\n",&timeline[11], timebuffer.millitm,text);
}
double timediff(struct timespec start, struct timespec end)
{
    struct timespec temp;
    if ((end.tv_nsec-start.tv_nsec)<0) {
        temp.tv_sec = end.tv_sec-start.tv_sec-1;
        temp.tv_nsec = 1000000000+end.tv_nsec-start.tv_nsec;
    } else {
        temp.tv_sec = end.tv_sec-start.tv_sec;
        temp.tv_nsec = end.tv_nsec-start.tv_nsec;
    }
    return((double)temp.tv_sec + (double)temp.tv_nsec / 1000000000.0);
}
int main(int argc,char *argv[])
{
        unsigned long LoopCount;
        int i, elapsed_seconds;
        struct timespec start, elapsed;
        double exacttime;
        printf("StressTest (C)2019 Goldstar Software Inc.\n");
        if(argc<2 || argv[1][0]=='?' || argv[1][1]=='?')
        {
            printf("\nUsage:  StressTest Filename\n\n    This tool can stress test a Zen database engine.\n");
            return(1);
        }
        printf("Opening Data File %s...\n",argv[1]);
        memset(posBlock,0,POS_BLOCK_SIZE);
        dataBuffer[0]=0;
        dataLength=0;
        strcpy(keyBuffer,argv[1]);
        StatusCode=BTRV(B_OPEN,posBlock,dataBuffer,&dataLength,keyBuffer,0);
        if(StatusCode)
        {
            printf("Unable to open file %s, Status was %d.\n",argv[1],StatusCode);
            return(1);
        }
        LoopCount = elapsed_seconds = 0;
        dataLength = MAX_DATABUF_SIZE;
        PrintTime("Starting the test...");
        clock_gettime(CLOCK_REALTIME, &start);
        while (elapsed_seconds < 10)    // We'll run for approximately 10 seconds, doing 1000 database calls each loop
        {
            for (i = 0; i < 1000; i++, LoopCount++)
                 StatusCode = BTRV(B_STEP_FIRST, posBlock, dataBuffer, &dataLength, keyBuffer, 0);
            clock_gettime(CLOCK_REALTIME, &elapsed);
            elapsed_seconds = elapsed.tv_sec - start.tv_sec;
        }
        PrintTime("Testing Complete...");
        exacttime = timediff(start, elapsed);
        printf("Loop Count = %lu\n",LoopCount);
        printf("Process Time: %6.3f seconds\n", exacttime);
        printf("Iterations/s: %-12.3f\n", (double)LoopCount / exacttime);
        // Close the file and reset the database connection
        StatusCode=BTRV(B_CLOSE,posBlock,dataBuffer,&dataLength,keyBuffer,0);
        StatusCode=BTRV(B_RESET,posBlock,dataBuffer,&dataLength,keyBuffer,0);
        return(0);
}
```