

Borland® InterBase® Events

Event mechanism in InterBase

A Borland White Paper

By Sudesh Shetty

September 2003

Borland®

Contents

Introduction to events 3

 Examples of applications using Borland® InterBase® events..... 3

Processing of events 4

Components of events 4

 Database and server component..... 4

 Program component 5

Working of event mechanism..... 5

The database component: 5

The program component: 6

The Server component: 6

Defining an event 7

Waiting for an event 7

Posting of an event 9

Analyzing an event 10

Sample event application..... 10

Less known feature 12

Conclusion..... 14

Introduction to events

With Borland® InterBase®, detecting and reporting changes in the database state are referred to as events. Statements such as INSERT, UPDATE and DELETE change the state of a database. Such statements are the originators of events. The InterBase database server can report these changes to the interested clients through the use of InterBase triggers or stored procedures. When used for this purpose, stored procedures and triggers are known as “event-alerters.”

Event alerters allow the database to broadcast events to interested client applications based on specific occurrences in the database. An event is a simple notification message; no data can be transmitted to or from the listening clients. The event mechanism gives an application a means to:

- Register interest in the occurrence of a specific database change
- Receive notification that the event has occurred

When an application receives notice from the database server that an event has occurred, it can then take appropriate action.

Examples of applications using Borland® InterBase® events

- Financial applications

An example of this is a stock or commodity that changes in value by a predetermined amount. Applications interested in the event are provided with notification of the event the moment it occurs.

- Calendar applications

An event could cause an application to e-mail attendees when a meeting is scheduled.

- Process control

An event could cause an alarm to sound when the temperature fluctuates more than a specified degree.

Processing of events

An application that has registered interest in an event can process the event notification *synchronously* or *asynchronously*. A process that requests *synchronous* notification of events ‘sleeps’ until the event occurs, whereas a process that requests *asynchronous* notification continues to run. This provides maximum flexibility for applications handling events.

While event alerters are not unique to InterBase, the InterBase implementation is. While other relational databases management systems (RDBMS) require the client applications to poll/query the database repetitively to know about database changes, InterBase does not require this. The event mechanism is part of the database engine. This allows the server to notify the client applications the moment a state change occurs. Hence, InterBase does not suffer from problems such as CPU time wastage or network traffic overload that are inherent to the polling mechanism.

Components of events

From a high-level perspective, the event mechanism can be divided into two parts:

- Database and server components, or
- Program components

Database and server component

Database components typically consist of at least one trigger or stored procedure. This procedure, or trigger, in addition to any conditional logic, should also contain at least one

‘post’ verb. The post verb is an extension to InterBase trigger/procedural language and is of the form ‘POST_EVENT.’

Server components consist of the event manager with event table. The event manager maintains a list of events that are posted to it by stored procedures and triggers. It also maintains a list of applications that have registered interest in an event.

Program component

The program components consist of:

- a) An application (user app) that would register an interest in an event and wait for the event to occur, and
- b) Another application (user app or standalone app such as ISQL) that trigger the event by changing the database state.

As we can see, there is a certain amount of interaction required by these components to complete the event mechanism.

Events are under transaction control. This implies that the server will not notify the interested application until the transaction that posted the event commits. However if a transaction happens to post the same event many times, the interested application will be notified by the server only once.

Working of event mechanism

We can understand the event mechanism better by dissecting it from different components’ viewpoint taking a simplistic approach.

The database component:

In a simple case, there would be just one trigger posting an event. For example, assume that one is interested in a particular table being updated. In order to accomplish this, the trigger,

which would ideally fire after the update has completed, has to be defined. Hence the user would be defining an after update trigger on the table. This trigger will simply post an event. Note that it is important to name the event. Without names, there is no way an application can register or wait for the event. In our case the trigger will post the event named 'TBL_UPDATED'. It's that simple.

The program component:

The application, which is interested in an event, has to

- Register with the server, and
- Wait for notification from the server.

The application can register with the server for events using either InterBase event API calls or Embedded SQL (ESQL). The application can wait for events the same way. However, an ESQL program can only wait *synchronously* for the event. If an application needs to wait *asynchronously* it needs to use InterBase event API calls.

In our simplistic approach, the other program component would be the InterBase ISQL application, which will attach to the database and issue an UPDATE statement, thereby firing the after update trigger.

The Server component:

“Event manager” and “event table” make up the server side component of the event mechanism. The event manager marshals the notification of events with the help of the event table. The event table contains a list of processes that have expressed interest in one or more events, as well as a list of events. When an event occurs, the event manager looks up the process list to determine the interested parties and notify them. Several scenarios are possible when an event fires.

- If an event is posted and there are one or more processes that are waiting for it, the event manager then promptly notifies each one of them.

- If an attempt is made to post an event and no processes have registered interest in it, the event manager simply ignores it.
- If an event is posted and a process has registered interest, but has not yet begun to wait for the event, the event manager takes note of the event. When the process tries to wait, it is immediately notified.

Defining an event

An event is created when an application defines a stored procedure or a trigger to post that event. When the InterBase server sees the verb 'POST_EVENT *event_name*' in a trigger or a stored procedure, it creates an entry in the event table. The event name should be unique. If different stored procedures or triggers post the same event, the server will create only one entry by that name in the event table. The sample code shown below will prompt the InterBase server to create an event named 'SELL'

```
CREATE PROCEDURE ORDER AS
BEGIN
    POST_EVENT 'SELL' ;
END;
```

Waiting for an event

Applications can wait for events *synchronously* using the ESQL statement EVENT WAIT or by using the InterBase API call *isc_wait_for_event()*. Applications that need to wait *asynchronously*, can only use the API *isc_que_events()*. There is no ESQL equivalent. Synchronous waiting for the event is straightforward. As soon as the application uses the API *isc_wait_for_event()* or ESQL statement EXEC SQL EVENT WAIT, the application is put to sleep until the event is posted. The sample ESQL code below shows how to wait for the event 'SELL' synchronously:

```
EXEC SQL EVENT INIT E1 ('SELL');
```

```
EXEC SQL EVENT WAIT E1;
```

Asynchronous waiting needs a bit more work. Before an application can use the API `isc_que_events()`, it needs to set up a callback function that can be invoked by the server when the event is posted. The server will invoke the callback function with the posted event list as argument. The callback function would have to copy this list into a more permanent location so that the application can process the events correctly. Here is a sample code that waits for events asynchronously using a callback function.

```
isc_callback callback_fn (char *permanent, short length, char
*temporary)
{
    /* Set a global flag to indicate that the callback was invoked*/
    callback_invoked = TRUE;
    while (length --)
        *permanent++ = *temporary++ ;
    return 0;
}

main()
{
    ...
    isc_que_events (status_vector, &db_hndl, &event_id, length,
    evnt_buffer, callback_fn, permanent);
    ...
    while (1)
    {
        if (callback_invoked == TRUE) {
            /*handle the events */
        }
        else {
            /* Do whatever other processing you want */
        }
    }
    ...
}
```

Posting of an event

An event can be posted using the InterBase trigger/procedural language extension `POST_EVENT`. Since this verb is specific to triggers and procedures it implies that these are the only two ways in which an event can be posted. How does one decide when to use triggers or procedures? The answer is whichever is most convenient. If an application needs to know every time a table gets inserted or updated, it is highly likely a trigger defined for the table is the best place to post that event. If, on the other hand, an application needs to know when a particular table has been completely populated, a stored procedure would be an ideal place to post such an event. In this case, after an application has populated a table, it can execute the stored procedure to notify the waiting application. Sample code for a defining InterBase trigger posting an event is shown below:

```
CREATE TRIGGER TBL_INS FOR TBL_SALES
    ACTIVE AFTER INSERT POSITION 0
    AS
        BEGIN
            POST_EVENT 'INSERTED' ;
        END;
```

Sample SQL code for InterBase stored procedure posting an event is shown below:

```
CREATE PROCEDURE TBL_POPULATED AS
    BEGIN
        POST_EVENT 'FINISHED' ;
    END;
```

Note that the indents in the above sample piece of codes are purely for readability purposes.

Analyzing an event

Once notified, an application will have to use the API *isc_event_count()* to analyze the posted events. Using this API becomes necessary when an application waits for more than 1 event. So far in this paper there has been no mention of applications waiting for more than 1 event. The reason for this was to keep the details and sample code snippets simple. However, InterBase allows an application to wait on one or more events. For example, if an application needs to wait on two events 'SELL' and 'BUY' then the sample ESQL code will be:

```
EXEC SQL EVENT INIT E ('SELL', 'BUY');
```

```
EXEC SQL EVENT WAIT E;
```

The application will "wake up" if any one of these events gets posted. In order to determine which event was posted, the application will have to make use of *isc_event_count()*.

Sample event application

Now we will combine all the information that has been presented so far. This sample event application describes a stock-trading scenario and has a table, *STOCKS* (that contains stock tickers and their current prices), a trigger, *STOCK_UPD_EVN*, (that posts events when the stock price is updated), an application *ALERTER* (which waits for the event), and an application *POSTER* (that updates the table *STOCKS*).

Database: The sample database is called **stocks.ib**. There is only one table in it called *STOCKS*. This table is defined as:

```
CREATE TABLE STOCKS (SYMBOL CHARACTER (10), PRICE NUMERIC (5,2));
```

Trigger: The trigger *STOCK_UPD_EVNT* to the *STOCKS* table is defined as:

```
CREATE TRIGGER STOCK_UPD_EVENT FOR STOCKS
```

```
ACTIVE AFTER UPDATE
```

```
AS

      BEGIN

          IF (NEW.PRICE - OLD.PRICE) > 1 THEN

              POST_EVENT NEW.SYMBOL;

      END;
```

The trigger logic makes sure that an event is posted only when the stock price has increased by 1. It does not register any other conditions. This logic can easily be tailored to satisfy different needs. The trigger body posts the event NEW.SYMBOL. What does this represent? For example, the PRICE that is being updated belonged to SYMBOL 'BORL.' In this case, the event that gets posted is 'BORL.' If the PRICE that got updated was 'ACME,' the event that will be posted is 'ACME.' In this way, one trigger can handle all the ticker symbols in the table. When this trigger fires, the event manager checks the event table for the posted event, and if found, the event manager then will wake up the waiting process.

Application ALERTER: Only event-specific parts of this C-embedded SQL program will be seen. First, the application has to connect to the database to get access to the event mechanism. Once connected, the application has to register interest in an event. It will then wait for the event. After it has been notified, the application can perform whatever is required to process the event.

```
main() {
...
EXEC SQL SET DATABASE DB = 'stocks.ib';
EXEC SQL CONNECT DB;
...
...
EXEC SQL EVENT INIT BORL_EVNT ('BORL');
EXEC SQL EVENT WAIT BORL_EVNT;
...
}
```

After establishing connection to the database, the application notifies the server about its interest in the event named BORL by using EVENT INIT statement. It refers to this request by the name BORL_EVNT. When the EVENT INIT statement is executed by the application, the server takes note of the event name i.e. BORL and stores it in its event table.

The application then waits for the event by executing EVENT WAIT statement. The program is put to sleep as soon as it executes the EVENT WAIT statement and will awaken only when the event BORL is posted. Once awake, the application can perform the appropriate task, such as bringing up a message box to execute another application that could send messages to the cell phones or PDAs.

Application POSTER: This sample application will attach to the database and update a particular stock symbol's price. The code fragment is shown below

```
main() {  
...  
EXEC SQL SET DATABASE DB = 'stocks.ib';  
EXEC SQL CONNECT DB;  
EXEC SQL UPDATE STOCKS SET PRICE = PRICE + 3 WHERE SYMBOL = 'BORL';  
EXEC SQL COMMIT;  
...  
}
```

Less known feature

While this paper has demonstrated ways in which events can be posted through a stored procedure, it is possible for an application to post an event using other methods. This is a less-known feature that requires in-depth knowledge of the InterBase internal binary language representation (blr) in order to program an event. ESQL program statements are parsed by the InterBase pre-compiler, GPRE, to produce blr statements. These blr statements can be compiled using InterBase API. The blr verb that is needed to post an event is: blr_post. Sometimes it is desirable to post events from programs. One example would be if two applications running on two different platforms need to synchronize, it would be ideal if the programs can post events to each other rather than executing a trigger or stored procedure.

Note, however, that for this to succeed both of the applications need to be attached to the same database.

The following ESQL program, when compiled and executed, will post an event named 'new_order.'

```
EXEC SQL BEGIN DECLARE SECTION;

EXEC SQL SET DATABASE empdb = "d:/work/Events/employee.gdb";

EXEC SQL END DECLARE SECTION;

static char blr_stuff [] = { blr_version5, blr_post, blr_literal,
blr_text, 9,0, 'n', 'e', 'w', '_', 'o', 'r', 'd', 'e', 'r', blr_eoc
};

int main()

{

    int    req_handle=0;

    EXEC SQL CONNECT empdb;

    EXEC SQL SET TRANSACTION READ COMMITTED;

    if (isc_compile_request(isc_status, &empdb, &req_handle,
sizeof(blr_stuff), blr_stuff))

        isc_print_status (isc_status);

    if (isc_start_request(isc_status, &req_handle, &gds__trans, 0))

        isc_print_status (isc_status);

    EXEC SQL COMMIT;

    EXEC SQL DISCONNECT empdb;

}
```

Conclusion

If you need to program an application activity that is based on a changed state in a database, then InterBase's event mechanism is an effective technique. For instance, this mechanism can be used to send an email announcement to each store in a retail chain when the central office changes a price. Similarly, by combining events notification with various external technologies one could send an automatic notification of an equipment failure to the PDAs or mobile phones of field service staff.

Made in Borland® Copyright © 2003 Borland Software Corporation. All rights reserved. All Borland brand and product names are trademarks or registered trademarks of Borland Software Corporation in the United States and other countries. All other marks are the property of their respective owners. Corporate Headquarters: 100 Enterprise Way, Scotts Valley, CA 95066-3249 • 831-431-1000 • www.borland.com • Offices in: Australia, Brazil, Canada, China, Czech Republic, Finland, France, Germany, Hong Kong, Hungary, India, Ireland, Italy, Japan, Korea, Mexico, the Netherlands, New Zealand, Russia, Singapore, Spain, Sweden, Taiwan, the United Kingdom, and the United States. • 20952