Firebird Wire Protocol

Carlos Guzman Alvarez, Mark Rotteveel

Version 0.18, 18 May 2025

Table of Contents

Chapter 1. Introduction

This document describes the Firebird wire protocol. Most of the information was obtained by studying the Firebird source code and implementing the wire protocol in the Firebird .NET provider and Jaybird (Firebird JDBC driver).

The protocol is described in the form of the message sent by the client and received from the server. The described protocol is Firebird/InterBase protocol version 10. Earlier (InterBase) versions of the protocol are not in scope for this document. Changes in later protocol versions are described in notes below the description of the relevant version 10 message (currently higher versions are only partially described), or the base message introduced in a later protocol version. Protocol changes to an existing message are generally additive (new fields are added to the end of a message) and cumulative (also apply for higher protocol versions), unless explicitly indicated otherwise.

This document is not complete. Consult the *InterBase 6 API Guide* for additional information on subjects like parsing the status vector, information request items, and the meaning of operations. You can find this manual under “InterBase 6.0 Manuals” in the Reference Manuals section of the Firebird website. We also recommend consulting the Firebird sources and other wire protocol implementations.

Unless otherwise indicated, a client request must be flushed to the server for processing. For some operations the flush can be deferred, so it is sent together with a different operation. Versions 11 and higher of the wire protocol explicitly support (or even require) deferring of operations, including deferring the read of the response.

In the protocol descriptions below, we include the names of the fields of the structs used in the Firebird sources; this can make it easier to search for how it’s used in Firebird itself.

Chapter 2. Protocol versions

Below is a high-level overview of the changes per protocol versions.

Be aware that protocol version greater than 10 are OR'ed with `0x8000` (`FB_PROTOCOL_FLAG`) to differentiate them from newer InterBase protocol versions with the same number. In message exchanges like Identification (connect), this masked version is used, while for example the database info item `fb_info_protocol_version` reports the bare version.

2.1. Protocol 10

The "baseline" protocol of this document. It was introduced in InterBase 6.0, and available in Firebird 1.0 and higher.

2.2. Protocol 11

Protocol 11 was introduced in Firebird 2.1. It introduces support for batching of messages, and lazy — or deferred — responses.

Specifically, it allows you to batch a message creating an object (e.g. a statement or blob), with subsequent operations on that object (e.g. information request, statement prepare, blob get, etc.) by using the *invalid object* handle (`0xFFFF`) instead of the actual handle. This reduces latency, as you don't have to wait for the server response to the create operation — containing the actual handle — before you can use the object.

In some cases, with `ptype_lazy_send`, the server will defer the response to an operation until a subsequent operation is performed. For example, the response to statement allocation (`op_allocate`) is withheld, in the expectation that a prepare (`op_prepare`) follows immediately.

Similarly, freeing a statement (`op_free_statement`) will not send its response immediately. This means that processing the response to a free can only be done later, after sending another operation, and before processing the response to that other operation.

The *invalid object* handle refers to the latest object created. So, while you can batch multiple create operations with use of those objects in a single send, you cannot interleave operations on different objects.

That is, "*create object1, use object1, create object2, use object2*" will work, but "*create object1, create object2, use object1, use object2*" will not work or result in unwanted effects, as after *create object2* handle `0xFFFF` refers to *object2*, not *object1*.

Protocol 11 also introduced "trusted" authentication, which is not (yet) documented.

2.3. Protocol 12

Protocol 12 was introduced in Firebird 2.5. It provides asynchronous cancellation support.

2.4. Protocol 13

Protocol 13 was introduced in Firebird 3.0. It provides the following new features:

- Authentication plugin support
- Wire protocol encryption
- Wire protocol compression
- Database encryption key callback
- Packed (`NULL`-aware) row data

2.5. Protocol 14

Protocol 14 was introduced in Firebird 3.0.1 to fix a bug in Database encryption key callback.

We recommend skipping separate implementation of this protocol version, and implement it as part of protocol 15.

2.6. Protocol 15

Protocol 15 was introduced in Firebird 3.0.2 and provides the following new features:

- Support for Database encryption key callback in the connect phase. This allows connections to encrypted databases that serve as their own security database.

2.7. Protocol 16

Protocol 16 was introduced in Firebird 4.0 and provides the following new features:

- Statement timeouts

2.8. Protocol 17

Protocol 17 was introduced in Firebird 4.0.1 and provides the following new features:

- Sync batch
- Batch information request

2.9. Protocol 18

Protocol 18 was introduced in Firebird 5.0 and provides the following new features:

- Scrollable cursors (see Execute, Fetch scroll, and Cursor information request)

2.10. Protocol 19

Protocol 19 was introduced in Firebird 5.0.3 and provides the following new features:

- Inline blobs (see Inline blob response, Fetch response, Execute, and Fetch)

Chapter 3. Common responses

The wire protocol has a limited set of responses. Some operations have a specific response, which is described together with the operation. Most operation, however, use one (or more) of the responses described in this section. The meaning and content depend on the operation that initiated the response.

3.1. Generic response

`Int32` — `p_operation`

Operation code

If operation equals `op_response` — `9`:

`Int32` — `p_resp_object`

Object handle

Although 32-bit in the protocol, valid handle values are always between 0 and 65535 (0xFFFF), with the "normal" range between 0 and 65000, where `0` either represents the connection itself, or means "no value".

`Int64` — `p_resp_blob_id`

Object ID

`Buffer` — `p_resp_data`

Data (meaning depends on the operation).

`Byte[]` — `p_resp_status_vector`

Status vector

The format of the status vector is `<tag><value>[{tag><value>} ...]<end>`, with `<tag>` an `Int32`, and where parsing of `<value>` depends on `<tag>`; `<end>` is `Int32` `isc_arg_end` — `0`. The length can only be determined by correctly parsing the status vector. The first 8 bytes are always an `Int32` tag (`isc_arg_gds` or `isc_arg_warning`) and an `Int32` value.

- If the status vector starts with `Int32` `isc_arg_gds` — `1` **and** the second `Int32` is non-zero, it is a failure response.
- If it starts with `Int32` `isc_arg_warning` — `18` **and** the second `Int32` is non-zero, it is a success response with warning(s).
- Otherwise, if the second `Int32` is zero, it is a success response

Information about parsing the status vector can be found in the *Interbase 6 API Guide* in the documentation set. It might also be advantageous to look at the sources of Firebird .NET Data Provider or Jaybird.

3.2. SQL response

Success response to `op_execute2` (see Execute) or `op_exec_immediate2` (not yet documented).

`Int32` — **`p_operation`**

Operation code

If operation equals `op_sql_response` — `78`:

`Int32` — **`p_sqldata_messages`**

Count of rows following response (in practice, only `1` or `0`)

`Byte[]` — **Row data**

The row data is a sequence (0..1) of data rows with a special format, see Reading and writing row data . Its length follows from the output row BLR in the execute operation.

You can also consider the row data not a part of the SQL response, but something that is sent **after** the SQL response.

3.3. Fetch response

Success response to `op_fetch` (see Fetch) and `op_fetch_scroll` (see Fetch scroll).

`Int32` — **`p_operation`**

Operation code

If operation equals `op_fetch_response` — `66`:

`Int32` — **`p_sqldata_status`**

Status

- `0` — success
- `100` — end of cursor

`Int32` — **`p_sqldata_messages`**

Count of rows following response (in practice, only `1` or `0`)

A value of `0` indicates end-of-batch (fetch complete). Together with status `100`, it also means end-of-cursor, otherwise there are more rows available for a next fetch.

`Byte[]` — **Row data**

The row data is a sequence (0..1) of data rows with a special format, see Reading and writing row data . Its length follows from the row BLR specified in the first fetch.

You can also consider the row data not a part of the fetch response, but something that is sent **after** the fetch response.

The success response to Fetch and Fetch scroll is not a single `op_fetch_response`, but a sequence of `op_fetch_response` and row data, or — since protocol 19 — a sequence containing a sequence of 0 or

more op_inline_blob, op_fetch_response and row data. That is:

Sequence of responses to a fetch

```
0..n <op-inline-blob ...> -- protocol 19 and higher
<op-fetch-response (status = 0, count = 1)>
<row-data>
0..n <op-inline_blob ...>  -- protocol 19 and higher
<op-fetch-response (status = 0, count = 1)>
<row-data>
...
if end-of-cursor:
  <op-fetch-response (status = 100, count = 0)>
else:
  <op-fetch-response (status = 0, count = 0)>
```

Firebird may return fewer rows than requested in Fetch or Fetch scroll, even if end-of-cursor is not yet reached.

3.4. Inline blob response

Introduced in protocol 19 (Firebird 5.0.3).

Each op_inline_blob — 114 response contains a single blob. The response is sent as part of a stream of responses to Execute (specifically op_execute2), op_exec_immediate2 (not yet documented), Fetch, and Fetch scroll.

Int32 — p_operation

Operation code

If operation equals op_inline_blob — 114:

Int32 — p_tran_id

Transaction handle

The same transaction handle as used when executing the statement.

Int64 — p_blob_id

Blob id

Buffer — p_blob_info

All blob info

Same encoding as p_resp_data in response to a blob information request with all blob information items.

Buffer — p_blob_data

Segmented blob data

Same encoding as `p_resp_data` in response to Get segment).

The transaction handle (`p_tran_id`) and blob id (`p_blob_id`) can — for example — be used as a key to look up inline blobs from a local — attachment specific — cache when the client wants to open a blob. Instead of remotely opening the blob and retrieving data or information from the server, the client can then serve the data from the inline blob.

The server sends each blob of the row in the following Fetch response, or SQL response as an inline blob, if it fits within the specified `p_sqldata_inline_blob_size` (including segment lengths). It is up to the client to decide if they want to cache the inline blob or discard it (e.g. if the cache is full).

3.5. Slice response

Success response to Get slice.

This might not reflect actual encoding in the protocol.

Response to Get slice.

`Int32` — `p_operation`

Operation code

If operation equals `op_slice` — `60`:

`Int32` — `p_slr_length`

Slice length

`Int32`

Slice length (possibly a buffer?, needs verification)

`Buffer`

Slice data

3.6. Dummy response

The server may occasionally send a “dummy” response. This is intended as a keep-alive feature, and is related to the `DummyPacketInterval` server setting and/or `isc_dpb_dummy_packet_interval` /`isc_spb_dummy_packet_interval` connection setting.

Though Firebird normally uses `SO_KEEPALIVE` (which is transparent to the client), clients must be able to handle the dummy response. The appropriate action is to read and ignore this response, and continue with the next response.

`Int32` — `p_operation`

Operation code (`op_dummy` — `71`)

3.7. Exit or disconnect command (aux connection)

As far as we're aware, this is only sent on the aux connection. It is similar to the disconnect request from client to server for the main connection.

Instructs the client to close the aux connection.

`Int32` — `p_operation`

Operation code (`op_exit` — 2 or `op_disconnect` — 6)

After receiving this message, the client should close the aux connection. It's generally only sent just before the main connection is closed.

Chapter 4. Common requests

A few requests in the protocol have a common message format, where the operation code differs, and — possibly — the set of allowed values of other fields.

We describe the request format here, and describe the allowed values in the section for a specific request.

4.1. Generic information request

Client

`Int32` — `p_operation`

Operation code (value depends on the actual operation)

`Int32` — `p_info_object`

Object handle (e.g. statement, transaction, etc.)

`Int32` — `p_info_incarnation`

Incarnation of object (`0`)

TODO: Usage and meaning?

`Buffer` — `p_info_items`

Requested information items

A list of requested information items (each byte is an information item), so-called `SingleTpb` items. Some operations may have items that do have values (e.g. `isc_info_sql_sqlda_start` of Statement information request). Most values are specific to the operation.

The list should end with `isc_info_end` — 1[1].

`Int32` — `p_info_buffer_length`

Length of buffer available for receiving response

In protocol 10, this is a signed Int16, encoded as Int32.
In protocol 11 and higher, this is an unsigned Int32.

For compatibility reasons, values greater than or equal to 4,294,901,760 (i.e. 0xFFFF_0000 or greater) are masked with 0xFFFF, so only the low 16 bits are used.

A too small value may lead to receiving a truncated buffer (last item is `isc_info_truncated` — 2 instead of `isc_info_end` — 1), which necessitates requesting information again with a larger size. Some operations may have additional mechanisms to handle truncation, like

The buffer in the response is sized to the actual length of the response (upto the declared available length), so specifying a larger than necessary size does not inflate the response on the wire. However, specifying an unnecessarily large size can lead to inefficiencies for the server.

Server

Generic response — on success, `p_resp_data` holds the requested information.

A truncated response is considered a success, and can only be determined by parsing `p_resp_data`.

Information about how to parse the information buffer sent by the Firebird server can be found in the InterBase 6.0 documentation set

[1] This is not required, at least not in recent Firebird versions. The server handles the end of the buffer without seeing `isc_info_end` as an implicit `isc_info_end`. However, we're not sure if that was always the case, so for potential compatibility reasons, consider it "required"

Chapter 5. Establishing connection

This chapter describes how to connect to a database or service. Other operations on a database or service, or information specific to connecting to a database or service are documented in Databases and Services.

5.1. Attaching to a database or service

In protocol 10 and 11, attachment to a database or service is done in two steps, first identification (connect) to the server, then attach to — or creation of — a database, or attach to a service. In protocol 13, this was changed to a more complex state machine to handle multiple authentication plugins, wire protocol encryption, and database encryption key callback.

In deviation of the normal description in this documentation, and previous versions of this documentation, we will first cover the individual messages, and then explain the order and logic of message exchange.

5.2. Identification (connect)

Requests connection to the server and specifies which protocol versions the client can use.

`Int32` — `p_operation`

Operation code (`op_connect` — 1)

`Int32` — `p_cnct_operation`

Unused, always use `0`

Some implementations use `op_attach` — 19/`op_service_attach` — 82 for historic(?) reasons.

`Int32` — `p_cnct_cversion`

Connect version:

`CONNECT_VERSION2` — 2	user identification encoding is undefined (Firebird 1.0 — Firebird 2.5)
`CONNECT_VERSION3` — 3	user identification is UTF-8 encoded (since Firebird 3.0 and higher, but backwards compatible as the version wasn't checked before Firebird 3.0)

`Int32` — `p_cnct_client`

Architecture type (e.g. `arch_generic` — 1).

`String` — `p_cnct_file`

Database path or alias

The encoding of this is undefined, which can lead to problems with non-ASCII paths if the server and client use a different encoding.

For a service connection, this value can be the service name (`service_mgr` or empty), or the "expected database" name (same value as `isc_spb_expected_db`).

`Int32` — `p_cnct_count`

Count of protocol versions understood (e.g. 1).

`Buffer` — `p_cnct_user_id`

User identification

TODO: Needs further description

The next block of data declares the protocol(s) that the client supports. It should be sent as many times as protocols are supported (and specified in `p_cnct_count` above). Values depend on the protocol.

If a client sends more than 10 (Firebird 5.0 and older) or 11 (Firebird 6.0) protocols, the surplus are ignored.

`Int32` — `p_cnct_version`

Protocol version (e.g. `PROTOCOL_VERSION10` — `10`).

```
Protocol versions greater than `10` need to be OR'ed with `0x8000` for
differentiation from newer InterBase protocol versions.
For example, `PROTOCOL_VERSION11` is `0x8000 | 11` or `32779` (`0x800B`)
```

`Int32` — `p_cnct_architecture`

Architecture type (e.g. `arch_generic` — 1)

It is possible to use a different architecture value, but then connection is only possible with a server of the same architecture. In addition, it changes how responses and/or data needs to be parsed or encoded (the authors don't know the exact details). In short, use `arch_generic`.

`Int32` — `p_cnct_min_type`

Minimum type (e.g. `ptype_batch_send` — 3)

Connection type (`ptype`) values

`ptype_page` — 1	Page server protocol (never supported in Firebird)
`ptype_rpc` — 2	Simple remote procedure call (not supported since Firebird 3.0)
`ptype_batch_send` — 3	Batch sends, no asynchrony
`ptype_out_of_band` — 4	Batch sends w/ out of band notification (semantics not documented in this manual)
`ptype_lazy_send` — 5	Deferred packets delivery

`Int32` — `p_cnct_max_type`

Maximum type (e.g. `ptype_lazy_send` — 5)

If the client wants to set up wire compression, this `ptype`-code must be OR'ed with `pflag_compress` (`0x100`). See also Known `p_acpt_type` flags below.

`Int32` — `p_cnct_weight`

Preference weight (e.g. 2). Higher values have higher preference. For equal weights, the last supported occurrence will be selected.

5.3. Simple accept

Specifies the protocol selected by the server. This response is — as far as we know — not sent if the server accepts protocol 13 or higher; then the extended Accept with data *or* conditional accept is sent instead.

`Int32` — `p_operation`

Operation code

If operation equals `op_accept` — 3:

`Int32` — `p_acpt_version`

Protocol version accepted by server

`Int32` — `p_acpt_architecture`

Architecture for protocol

`Int32` — `p_acpt_type`

Accepted type and additional flags. Obtain the type by masking with `0xFF` (`p_acpt_type & 0xFF`).

Known `p_acpt_type` *flags*

`pflag_compress` — `0x100`	Turn on compression From client to server, it signals a request to use wire compression. From server to client, it is an acknowledgement, and wire compression **must** be enabled *after* reading this entire response, but *before* reading or writing any other messages.
`pflag_win_sspi_nego` — `0x200`	Win_SSPI supports Negotiate security package Only sent from server to client.

Failure response: Generic response

5.4. Accept with data *or* conditional accept

Introduced in protocol 13.

The `op_accept_data` — `94` and `op_cond_accept` — `98` responses start with the same fields as Simple accept, followed by additional fields for authentication and encryption purposes.

`Int32` — `p_operation`
: Operation code

If operation equals `op_accept_data` — `94` or `op_cond_accept` — `98`:

`Int32` — `p_acpt_version`
: Protocol version number accepted by server

`Int32` — `p_acpt_architecture`
: Architecture for protocol

`Int32` — `p_acpt_type`
: Accepted type and additional flags.

  See also `p_acpt_type` in `op_accept` message

`Buffer` — `p_acpt_data`
: Authentication plugin data

`String` — `p_acpt_plugin`
: Authentication plugin to continue with

`Int32` — `p_acpt_authenticated`
: Authentication complete in a single step (`0` — false, `1` — true)

  This will generally only be `1` if `Legacy_Auth` was tried first, though third-party authentication plugins might also authenticate in a single step.

`Buffer` — `p_acpt_keys`
: “Keys” known by the server (used for configuring authentication and wire encryption)

5.5. Continue authentication

Introduced in protocol 13(?).

This message is used both by client *and* server to exchange authentication information.

`Int32` — `p_operation`
: Operation code (`op_cont_auth` — `92`)

`Buffer` — `p_data`
: Authentication data

`String` — `p_name`
: Name of the current authentication plugin

`String` — `p_list`

On first authentication from client to server: list of (remaining) plugins known to the client, including the current plugin;
on subsequent authentication from client to server, or from server: empty

The list of plugin names can be separated by space, tab, comma or semicolon.

`Buffer` — `p_keys`

From client to server: empty;
from server to client: “keys” known by the server (used for configuring authentication and wire encryption)

5.6. Trusted authentication

Introduced in protocol 11.

`Int32` — `p_operation`

Operation code (`op_trusted_auth` — `90`)

`Buffer` — `p_trau_data`

Trusted authentication data

5.7. Database encryption key callback

Introduced in protocol 13.

Used to exchange information between the client and server parts of a database encryption plugin for the encryption key. The server sends this message, and the client responds with the same message type. Specifics of the message exchange depends on database encryption plugin. It is possible that multiple message of this type are exchanged.

In protocol 13, this message can only occur after authentication and — optionally — establishing wire protocol encryption. In protocol 15 and higher, it can also occur immediately after `op_connect`, if the database is its own security database *and* is encrypted.

If this message is received *before* `op_accept`/`op_accept_data`/`op_cond_accept` (so no protocol version has been confirmed yet), you need to assume protocol 15 behaviour for this message and the client response.

`Int32` — `p_operation`

Operation code (`op_crypt_key_callback` — `97`)

`Buffer` — `p_cc_data`

Crypt callback data

Additions in protocol 14

`Int32` — `p_cc_reply`

Maximum expected reply size (16-bit signed integer encoded as 32-bit int)

Judging by the code in Firebird for protocol 14 and higher, this value may be negative, and should then be considered equivalent to 1.

From client to server, the reply size should be `0`.

5.8. Enable wire encryption

Introduced in protocol 13.

Enables wire encryption by telling the server the selected plugin and key type.

Client

`Int32` — `p_operation`

Operation code (`op_crypt` — `96`)

`String` — `p_plugin` Selected wire encryption plugin

`String` — `p_key` Selected key type

After sending this message, the client must set up wire encryption both for sending and receiving data. Subsequent messages — including the server response to this message — must be sent or received with encryption enabled.

Server

Generic response

5.9. Attachment to database or service

This message is used for:

- Attaching to a database (`op_attach` — `19`) — see also Attach database
- Creating a database (`op_create` — `20`) — see also Create database
- Attaching to a service (`op_service_attach` — `82`) — see also Attach

Client

`Int32` — `p_operation`

Operation code (`op_attach` — `19`, `op_create` — `20`, or `op_service_attach` — `82`)

`Int32` — `p_atch_database`

Unused, always use `0`

`String` — `p_atch_file`

Database path or alias, or service name (e.g. `service_mgr`).

If `isc_dpb_utf8_filename` is present in the database parameter buffer below, the encoding is UTF-8, otherwise, the encoding is undefined. The `isc_dpb_utf8_filename` item is supported since Firebird 2.5.

`Buffer` — `p_atch_dpb`

Database or service parameter buffer

Server

In protocol 10 and 11:

Generic response

In protocol 13 and higher:

It's complicated.

5.10. Connection rejection

Server response rejecting the connection. This is usually sent if `op_connect` only sent protocols the server can't support.

`Int32` — `p_operation` Operation code (`op_reject` — 4)

If this message is received, the client should report error `isc_connect_reject` (335544421) or equivalent.

5.11. Connection handshake

5.11.1. Protocol 10 — 12

For protocol 10 — 12, the connection handshake is pretty simple.

1. Client → Identification (connect)
2. Server
   - ← `op_accept` — 3 — Server accepts and reports selected protocol, continue with step 3
   - ← `op_reject` — 4 — Server can't fulfill the requested protocol
     - Report error `isc_connect_reject` (335544421) or equivalent
     - Close connection
   - ← `op_response` — 9 — Error or other problem
     - If `p_resp_status_vector` has an error, report it, otherwise report error `isc_login` (335544472) or equivalent
     - Close connection

3. Client → Attachment to database or service with `op_attach`, `op_create` or `op_service_attach`
4. Server ← `op_response` — 9
   - If `p_resp_status_vector` has no error or only a warning, connection is successful and can be used for other operations
   - Otherwise, connection is unsuccessful
     - Report error
     - Close connection (client → Disconnect)

5.11.2. Protocol 13 and higher

For protocol 13 and higher, the handshake is more complex.

This might not be the best way to document the connection handshake. We're open to suggestions.

1. Client → Identification (connect)

   The `p_user_identification` should include:

   - `CNCT_plugin_name` with the current authentication plugin
   - `CNCT_plugin_list` with the authentication plugins supported by the client (including the current plugin); this list is separated by space, comma or semicolon
   - `CNCT_specific_data` with authentication plugin data (NOTE: this tag has a special "multipart" encoding as the data is generally longer than the 255 bytes supported for a single tag value)
2. Server
   - ← `op_crypt_key_callback` — 97 (read as protocol 15)
     - Client → Database encryption key callback (write as protocol 15) and continue with step 2
   - ← `op_accept` — 3 — Record selected protocol and type, continue with step 5 (attach)
   - ← `op_accept_data` — 94 or `op_cond_accept` — 98

     Record the selected protocol and type, and use that for sending and receiving subsequent messages. Enable wire compression if acknowledged by server.

     If `p_acpt_authenticated == 1`, mark authentication completed

     - If `op_accept_data` — 94, process the data, plugin and keys, and continue with step 5 (attach)
     - If `op_cond_accept` — 98, continue with step 3 (pre-attach-auth), item for `op_cond_accept`
   - ← `op_reject` — 4 — Server can't accept any of the protocols or protocol options
     - report error `isc_connect_reject` (335544421) or equivalent
     - close connection

- ← `op_response` — `9` — Error or other problem
  - If `p_resp_status_vector` has an error, report it, otherwise report error `isc_login` (`335544472`) or equivalent
  - close connection (end of this flow)

3. Server — pre-attach auth

   If the requested authentication plugin name (`p_acpt_plugin`/`p_name`) is non-empty and different from the current authentication plugin name, switch to that authentication plugin.

   If the client cannot fulfill the server request for an authentication plugin or has no current authentication plugin, error `isc_login` (`335544472`) or equivalent should be reported, and the connection closed (end of this flow)

   If coming from step 2, treat this as if `op_cond_accept` was just received.

   - ← `op_cond_accept` — `98`: process `p_acpt_data`, `p_acpt_plugin` and `p_acpt_keys`, and continue with step 4
   - ← `op_cont_auth` — `92`: process `p_data`, `p_name` (plugin name) and `p_keys`, and continue with step 4
   - ← `op_crypt_key_callback` — `97`
     - Client → Database encryption key callback and continue with step 3 (pre-attach auth)
   - ← `op_trusted_auth` — `90` (not documented yet, probably only post-attach auth with protocol 11 and 12(?))
   - ← `op_response` — `9`
     - If `p_resp_statusvector` has an error, report it and close the connection (end of this flow)
     - Otherwise, this signals pre-attach auth (or post-attach auth) completed
       - Process keys from `p_resp_data`
       - If authentication was **not** previously completed, and wire encryption is not disabled, set up wire encryption
         - Client → Enable wire encryption
         - Set up wire encryption on incoming and outgoing stream
         - Server ← `op_response` — `9`: if `p_resp_statusvector` has error, report it and close connection (Disconnect), (end of flow)
       - Mark authentication completed
     - Continue with step 5 (attach);
       or if used as post-attach auth, attach successfully completed (end of flow)

4. Client — pre-attach auth → Continue authentication with:

   - `p_data` — authentication plugin data
   - `p_name` — current authentication plugin
   - `p_list` — list of remaining authentication plugins, including current plugin (separated by

space, comma, or semicolon)

This only needs to be sent *once*; for subsequent messages an empty buffer can be sent.

Continue with step 3 (Server — pre-attach auth)

5. Client — attach → Attachment to database or service with `op_attach`, `op_create` or `op_service_attach`

   If authentication was not yet complete at this point (as far as we know, only when `op_accept` — `3` or `op_accept_data` — `94` was received in the previous step), and protocol 13 or higher was selected, the database or service parameter buffer should include the following tags:

   If protocol 13 or higher is used, the "wide" parameter buffer variant (`isc_dpb_version2`/`isc_spb_version3` or higher) must be used given the size of the client authentication data (`…_specific_auth_data`).

   - `isc_dpb_auth_plugin_list`/`isc_spb_auth_plugin_list` — with remaining authentication plugins (separated by space, comma or semicolon)
   - `isc_dpb_auth_plugin_name`/`isc_spb_auth_plugin_name` — current authentication plugin
   - `isc_dpb_specific_auth_data`/`isc_spb_specific_auth_data` — client authentication data

   It should not include any of these tags (**if** protocol 13 or higher):

   - `isc_dpb_password`/`isc_spb_password`
   - `isc_dpb_password_enc`/`isc_spb_password_enc`
   - `isc_dpb_trusted_auth`/`isc_spb_trusted_auth`

6. Server/client — post-attach auth

   This is the same as steps 3 and 4 (pre-attach auth), except `op_cond_accept` cannot occur, and its "Continue with step 5 (attach)" should be read as "Connection successful (end of flow)" (also noted there).

   If the pre-attach auth flow was previously entered, this will essentially be only an `op_response` — `9` with either an error or acceptance (connection success).

5.12. Detach

Detaches from the database (`op_detach` — `21`) or service (`op_service_detach` — `83`). After detach the connection is still open, to disconnect send Disconnect (`op_disconnect`).

Client

`Int32` — `p_operation`

Operation code (`op_detach` — `21`, or `op_service_detach` — `83`)

`Int32` — `p_rlse_object`

Unused, always use `0`

Server

Generic response

5.13. Disconnect

Client

`Int32` — `p_operation`

Operation code (`op_disconnect`)

Server

No response, remote socket close.

Closing the connection (socket) without sending an `op_disconnect` will result in “Connection reset by peer” (error `10054` (Windows) or `104` (Linux)) in `firebird.log`.

Chapter 6. Databases

6.1. Attach database

Attach to an existing database. Use message Attachment to database or service with `op_attach` — 19.

Table 1. Example of parameters sent in the DPB

Parameter	Description	Value	Optional
`isc_dpb_version1`	Version (must be first item!)		
`isc_dpb_dummy_packet_interval`	Dummy packet interval	120	*
`isc_dpb_sql_dialect`	SQL dialect	3	
`isc_dpb_lc_ctype`	Character set	UTF8	
`isc_dpb_sql_role_name`	User role	RDB$ADMIN	*
`isc_dpb_connect_timeout`	Connection timeout	10	*
`isc_dpb_user_name`	User name	SYSDBA	
`isc_dpb_password`	User password	masterkey	

6.2. Create database

Create a database and connect to it. Create uses Attachment to database or service with `p_operation` `op_create` — `20`.

There are a number of DPB items to configure the newly created database, including page size (`isc_dpb_page_size`) — which cannot be modified after creation.

The `CREATE DATABASE` statement

Although Firebird has a `CREATE DATABASE` statement, the documented syntax is not fully supported by Firebird server. Part of the syntax (e.g. database name, user, password, page size) are parsed by *fbclient* to execute the `op_create` (or equivalent for embedded).

After the database is successfully created, *fbclient* then uses execute immediate (`op_execute_immediate`) without transaction to execute a reduced `CREATE DATABASE` statement for additional configuration of the database.

6.3. Drop

Drops the currently attached database.

Client

`Int32` — `p_operation`
: Operation code (`op_drop_database`)

`Int32` — `p_rlse_object`
: Unused, always use `0`

Server

Generic response

6.4. Detach and disconnect

Send Detach with `op_detach` — `21`, followed by Disconnect.

6.5. Database information request

Requests database or server information.

Uses the Generic information request message with:

`p_operation`	`op_info_database` — `40`
`p_info_object`	Unused, always use `0`
`p_info_items`	Values of enum `db_info_types` in Firebird's `inf_pub.h`.

6.6. Cancellation

Protocol 12 and higher.

Cancels a running operation on the server.

Operation `fb_cancel_abort` — `4` should not be sent to the server, but instead the client should simply close the socket connection.

Client

`Int32` — `p_operation`
: Operation code (`op_cancel`)

`Int32` — `p_co_kind`
: Cancellation kind, one of:

  `fb_cancel_disable` — `1`
  : disable cancellation until `fb_cancel_enable` is sent

  `fb_cancel_enable` — `2`
  : enable cancellation if it was disabled previously

`fb_cancel_raise` — `3`

: cancel current operation

`fb_cancel_abort` — `4`

: See note above, this *kind* should not be sent to the server.

As cancellation is generally performed asynchronously to be effective, the client implementation must take special care how the operation is sent.

For example, if you use a lock for socket operations, this operation will need to ignore it (running the risk of interfering/corrupting another send operation), or you need to split your locks in a lock for writing, and a lock for reading, or have some other way of detecting that another thread is not currently sending data.

Server

No formal response, cancellation is signalled as a Generic response with a failure for the cancelled operation.

Chapter 7. Transactions

7.1. Start transaction

Starts a transaction with the transaction options specified in the transaction parameter buffer.

Client

`Int32` — `p_operation`

Operation code (`op_transaction` — `29`)

`Int32` — `p_sttr_database`

Unused, always use `0`

`Buffer` — `p_sttr_tpb`

Transaction parameter buffer

Server

Generic response — on success, `p_resp_object` is the new transaction handle.

The `SET TRANSACTION` statement

Instead of using `op_transaction` to start a transaction, it is also possible to use the `SET TRANSACTION` statement.

This statement needs to be executed with execute immediate (`op_execute_immediate`) without transaction. On success, the `p_resp_object` holds the transaction handle.

7.1.1. Deviations for protocol version 11

Request flushing and response processing can be deferred.

If `ptype_batch_send` or higher is used, other transaction operations can be sent immediately after starting the transaction. They can use the *invalid object* handle (`0xFFFF`) instead of the — not yet received — transaction handle. This probably only makes sense for Transaction information request.

7.2. Commit transaction

Commits an active or prepared transaction.

Client

`Int32` — `p_operation`

Operation code (`op_commit` — `30`)

`Int32` — `p_rlse_object`

Transaction handle

Server

Generic response

7.3. Rollback transaction

Rolls back an active or prepared transaction.

Client

`Int32` — `p_operation`

Operation code (`op_rollback` — `31`)

`Int32` — `p_rlse_object`

Transaction handle

Server

Generic response

7.4. Commit retaining

Commits an active or prepared transaction, retaining the transaction context.

Client

`Int32` — `p_operation`

Operation code (`op_commit_retaining` — `50`)

`Int32` — `p_rlse_object`

Transaction handle

Server

Generic response

7.5. Rollback retaining

Rolls back an active or prepared transaction, retaining the transaction context.

Client

`Int32` — `p_operation`

Operation code (`op_rollback_retaining` — `86`)

`Int32` — `p_rlse_object`

Transaction handle

Server

Generic response

7.6. Prepare

Performs the first stage of a two-phase commit. After prepare, a transaction is *in-limbo* until committed or rolled back.

7.6.1. Simple prepare

Client

`Int32` — `p_operation`

Operation code (`op_prepare` — 32)

`Int32` — `p_rlse_object`

Transaction handle

Server

Generic response

7.6.2. Prepare with message

Associates a message (byte data) with the prepared transaction. This information is stored in `RDB$TRANSACTIONS` and can be used for recovery purposes.

Client

`Int32` — `p_operation`

Operation code (`op_prepare2` — 51)

`Int32` — `p_prep_transaction`

Transaction handle

`Buffer` — `p_prep_data`

Recovery information

Server

Generic response

7.7. Reconnect transaction

Reconnects a prepared ("in-limbo") transaction for 2-phase commit or rollback.

This operation can be used for recovery operations if a connection was closed or killed after preparing a transaction, but not yet committing or rolling it back.

Client

`Int32` — `p_operation`

Operation code (`op_reconnect` — 33)

`Int32` — `p_sttr_database`

Unused, always use `0`

`Buffer` — `p_sttr_tpb`

Transaction id to reconnect, encoded in little-endian.

For Firebird 2.5 and lower, always 4 bytes (`Int32` little-endian).

For Firebird 3.0 and higher, transaction ids greater than 0x7FFF_FFFF (2^{31} - 1) must be encoded in 8 bytes (`Int64` little-endian), while smaller ids may be encoded in 4 bytes (`Int32` little-endian).

This encoding is atypical, as it's essentially a transaction parameter buffer without version or item tags.

Server

Generic response — on success, `p_resp_object` holds the transaction handle.

7.7.1. Deviations for protocol version 11

Request flushing and response processing can be deferred.

If `ptype_batch_send` or higher is used, other transaction operations can be sent immediately after reconnecting the transaction. They can use the *invalid object* handle (`0xFFFF`) instead of the — not yet received — transaction handle.

7.8. Transaction information request

Requests information on the transaction bound to the transaction handle.

Uses the Generic information request message with:

`p_operation`	`op_info_transaction` — 42
`p_info_object`	Transaction handle
`p_info_items`	Values of constants in Firebird's `inf_pub.h` starting with `isc_info_tra_` or `fbinfo_tra_`.

Chapter 8. Statements

8.1. Allocate

Allocates a statement handle on the server.

Client

`Int32` — `p_operation`

Operation code (`op_allocate_statement` — `62`)

`Int32` — `p_rlse_object`

Unused, always use `0`

Server

Generic response — on success, `p_resp_object` is the allocated statement handle.

8.1.1. Deviations for protocol version 11

In protocol 11 and higher with `ptype_lazy_send`, the response to `op_allocate_statement` is deferred; it requires another operation on the connection before the response is sent.

In general, this means the *allocate* operation should be sent together with a *prepare* operation using the *invalid object* handle (`0xFFFF`).

8.2. Free

Frees resources held by the statement.

Client

`Int32` — `p_operation`

Operation code (`op_free_statement` — `67`)

`Int32` — `p_sqlfree_statement`

Statement handle

`Int32` — `p_sqlfree_option`

`DSQL_close` — `1`	Closes the cursor opened after statement execute.
`DSQL_drop` — `2`	Releases the statement handle.

`DSQL_unprepare` — `4` *Firebird 2.5 or higher*
Close resources associated with statement handle, and unprepares the current statement text. The statement handle itself is retained.

It is not necessary to unprepare before preparing a new statement text on the same handle.

The server treats these as flag values, so they can be combined with OR, but doing so makes little sense, as an *unprepare* also closes the cursor, and a *drop* effectively closes the cursor and unprepares the current statement text.

Server

Generic response

8.2.1. Deviations for protocol version 11

Request flushing can be deferred for `ptype_batch_send` or higher. For `ptype_lazy_send`, the response to `op_free_statement` is deferred; it requires another operation on the connection before the response is sent.

For `DSQL_drop` and `DSQL_unprepare`, we recommend flushing immediately so the server at least processes the request, which will prevent longer than necessary retention of metadata locks.

8.3. Prepare

Client

`Int32` — `p_operation`
Operation code (`op_prepare_statement` — `68`)

`Int32` — `p_sqlst_transaction`
Transaction handle

`Int32` — `p_sqlst_statement`
Statement handle

`Int32` — `p_sqlst_SQL_dialect`
SQL dialect (1 or 3)

This should generally match the connection dialect.

`String` — `p_sqlst_SQL_str`
Statement to be prepared

`Buffer` — `p_sqlst_items`
Statement information items, including describe and describe bind

Example of requested information items

- `isc_info_sql_select`
- `isc_info_sql_describe_vars`
- `isc_info_sql_sqlda_seq`
- `isc_info_sql_type`
- `isc_info_sql_sub_type`
- `isc_info_sql_length`
- `isc_info_sql_scale`
- `isc_info_sql_field`
- `isc_info_sql_relation`

`Int32` — `p_sqlst_buffer_length`

Target buffer length for information response

See also the description of `p_info_buffer_length` in Generic information request.

Server

Generic response — on success, `p_resp_data` holds the statement description (matching the requested information items)

For statements with a lot of columns and/or parameters, it may be necessary to handle truncation of the buffer by repeating the describe and/or describe bind information request using Statement information request and using `isc_info_sql_sqlda_start` to inform the server from which column or parameter to continue.

For an example, see Jaybird's `StatementInfoProcessor.handleTruncatedInfo(…)`.

8.3.1. Deviations for protocol version 11

The statement handle can no longer be allocated separately (or at least, its response is deferred). The initial Allocate operation **must** be sent together with the first prepare operation. When allocating and preparing together, the value of the statement handle of the *prepare* message must be `0xFFFF` (*invalid object* handle). The responses must be processed in order: first *allocate* response, then *prepare* response.

Once a statement handle has been allocated, it can be reused by sending a *prepare* message with its statement handle.

8.4. Describe

Requesting a description of output parameters (columns) of a query is done using the statement information request message

Example of requested information items

- `isc_info_sql_select`

- isc_info_sql_describe_vars
- isc_info_sql_sqlda_seq
- isc_info_sql_type
- isc_info_sql_sub_type
- isc_info_sql_length
- isc_info_sql_scale
- isc_info_sql_field
- isc_info_sql_relation

The initial request can be done as part of Prepare. The information can be requested together with Describe bind (input parameters).

8.5. Describe bind (input parameters)

Describe of input parameters of a query is done using the statement information request message

Example of requested information items

- isc_info_sql_bind
- isc_info_sql_describe_vars
- isc_info_sql_sqlda_seq
- isc_info_sql_type
- isc_info_sql_sub_type
- isc_info_sql_length
- isc_info_sql_scale
- isc_info_sql_field
- isc_info_sql_relation

The initial request can be done as part of Prepare. The information can be requested together with Describe.

8.6. Execute

Client

Int32 — p_operation

Operation code

op_execute — 62	DDL and DML statements
op_execute2 — 76	Executable stored procedures with return values, or singleton RETURNING (i.e. statements described as isc_info_sql_stmt_exec_procedure)

`Int32` — `p_sqldata_statement`

Statement handle

`Int32` — `p_sqldata_transaction`

Transaction handle

`Buffer` — `p_sqldata_blr`

Row BLR of parameters

If there are no parameters, send a zero-length buffer.

`Int32` — `p_sqldata_message_number`

Unused, always use `0`

`Int32` — `p_sqldata_messages`

Number of messages — `1` if there are parameters, `0` if there are no parameters

`Byte[]` — **Row data**

Row data of parameter values matching the row BLR specified in `p_sqldata_blr`

If `p_sqldata_messages` is `0`, nothing is sent

If using `op_execute2` — `76` (the statement is a stored procedure and there are output parameters):

`Buffer` — `p_sqldata_out_blr`

Row BLR of the output row

`Int32` — `p_sqldata_out_message_number`

Unused, always use `0`

Additions in protocol 16

`UInt32` — `p_sqldata_timeout`

Statement timeout value in milliseconds (`0` — use connection-level statement timeout)

Additions in protocol 18

`UInt32` — `p_sqldata_cursor_flags`

Cursor flags

`CURSOR_TYPE_SCROLLABLE` — `0x01` request scrollable cursor

Additions in protocol 19

`UInt32` — `p_sqldata_inline_blob_size`

Maximum inline blob size

A value of `0` disables inline blobs. The server may use a lower limit than requested. In the Firebird 5.0.3 and Firebird 6 implementation at the time of writing, the server has a maximum of 65535 bytes.

Server

For `op_execute` — `63`:

Generic response

For `op_execute2` — `76`:

Success response: (zero or more Inline blob response), SQL response followed by Generic response

The Inline blob response messages are only included in protocol 19 or higher, and only if the statement was executed with a non-zero value for `p_sqldata_inline_blob_size`, and if the row has blobs smaller than that size.

Failure response: only Generic response

8.7. Rows affected by query execution

Obtaining the rows affected by a query is done using the statement information request message

List of requested information items

- `isc_info_sql_records`

8.8. Fetch

Client

`Int32` — `p_operation`
: Operation code (`op_fetch` — `65`)

`Int32` — `p_sqldata_statement`
: Statement handle

`Buffer` — `p_sqldata_blr`
: Row BLR of the output rows

  Only needs to be sent on first fetch; subsequent fetches can send a zero-length buffer.

`Int32` — `p_sqldata_message_number`
: Message number (always `0`)

`Int32` — `p_sqldata_messages`
: Message count/fetch size (e.g. `200`)

  The server may decide to return fewer rows than requested, even if the end-of-cursor wasn't reached yet.

Server

Success response: one or more (zero or more Inline blob response), Fetch response

The Inline blob response messages are only included in protocol 19 or higher, and only if the statement was executed with a non-zero value for `p_sqldata_inline_blob_size`, and if the row has blobs smaller than that size. See also Sequence of responses to a fetch.

Failure response: Generic response — with an error in `p_resp_status_vector`

It is possible to receive Generic response with an error in the status vector after one or more fetch responses.

8.9. Fetch scroll

Introduced in protocol 18 (Firebird 5.0).

Fetches from a scrollable cursor.

This message is an extended version of Fetch.

Client

`Int32` — `p_operation`

Operation code (`op_fetch_scroll` — 112)

`Int32` — `p_sqldata_statement`

Statement handle

`Buffer` — `p_sqldata_blr`

Row BLR of the output rows

Only needs to be sent on first fetch; subsequent fetches can send a zero-length buffer.

`Int32` — `p_sqldata_message_number`

Unused, always use `0`

`Int32` — `p_sqldata_messages`

Message count/fetch size (e.g. `200`)

The server may decide to return fewer rows than requested, even if the end-of-cursor wasn't reached yet.

Ignored for `p_sqldata_fetch_op` other than `fetch_next`, or if the statement is a `SELECT ⋯ FOR UPDATE`; for those fetch operations or conditions, at most 1 row is fetched.

`Int32` — `p_sqldata_fetch_op`

Fetch operation

Valid values (see also enum `P_FETCH` in `protocol.h`):

`fetch_next` — `0`	Fetch next rows (same as using Fetch)
`fetch_prior` — `1`	Fetch previous row
`fetch_first` — `2`	Fetch first row
`fetch_last` — `3`	Fetch last row
`fetch_absolute` — `4`	Fetch row by absolute position Negative values of `p_sqldata_fetch_pos` are from the end of the cursor (i.e. `-1` is the last row, `-2` the row before the last row, etc.)
`fetch_relative` — `5`	Fetch row relative to the current position of the server-side cursor

If the current cursor is *not* a scrollable cursor, only `fetch_next` is accepted.

`Int32` — `p_sqldata_fetch_pos`

Requested position

Ignored if `p_sqldata_fetch_op` is not `fetch_absolute` or `fetch_relative`.

If combining `fetch_next` with a fetch size greater than 1 with other scroll operations, you may need to keep your own position accounting to ensure you scroll to the right row.

Server

Success response: one or more (zero or more Inline blob response), Fetch response

The Inline blob response messages are only included in protocol 19 or higher, and only if the statement was executed with a non-zero value for `p_sqldata_inline_blob_size`, and if the row has blobs smaller than that size. See also Sequence of responses to a fetch.

Failure response: Generic response — with an error in `p_resp_status_vector`

It is possible to receive Generic response with an error in the status vector after one or more fetch responses.

8.10. Set cursor name

Client

`Int32` — `p_operation`

Operation code (`op_set_cursor` — `69`)

`Int32` — `p_sqlcur_statement`

Statement handle

`String` — `p_sqlcur_cursor_name`

Cursor name (null terminated!)

`Int32` — `p_sqlcur_type`

Cursor type

Reserved for future use, always use `0`.

Server

Generic response

8.11. Statement information request

Requests information on the statement prepared on the statement handle, including information on its input parameters and output columns or parameters, or information on the server-side statement handle itself.

Uses the Generic information request message with:

`p_operation` `op_info_sql` — `70`

`p_info_object` Statement handle

`p_info_items` Values of constants in Firebird's `inf_pub.h` starting with `isc_info_sql_`.

8.12. Cursor information request

Requests information on an open cursor of a statement handle.

Uses the Generic information request message with:

`p_operation` `op_info_cursor` — `113`

`p_info_object` Statement handle

`p_info_items` Values of constants in Firebird's `IResultSet` class in `IdlFbInterfaces.h` starting with `INF_`.

Known items:

`INF_RECORD_COUNT` — `10` Cursor size (total number of records in scrollable cursor)

If the cursor is not scrollable, the returned value is -1 to indicate the value is unknown.

This request should only be sent after at least one fetch (Fetch or Fetch scroll). Attempts to request cursor information between execute and the first fetch may result in SQLDA errors on fetch.

Chapter 9. Blobs

9.1. Create/Open

Client

`Int32` — `p_operation`

Operation code

`op_create_blob` — `34`	Creates a new blob
`op_create_blob2` — `57`	Creates a new blob with a blob parameter buffer
`op_open_blob` — `35`	Opens an existing blob
`op_open_blob2` — `56`	Opens an existing blob with a blob parameter buffer

`Buffer` — `p_blob_bpb`

Blob parameter buffer

Only sent for `op_create_blob2` — `57` and `op_open_blob2` — `56`.

`Int32` — `p_blob_transaction`

Transaction handle

`Int64` — `p_blob_id`

Blob ID

Server

Generic response — on success

+

a. `p_resp_object` is the blob handle
b. `p_resp_blob_id` is the blob id (for `op_create_blob` --`35`/ `op_create_blob2` — `57`)

9.1.1. Deviations for protocol version 11

Request flushing and response processing can be deferred.

If `ptype_batch_send` or higher is used, other blob operations can be sent immediately after the open/create. They can use the *invalid object* handle (`0xFFFF`) instead of the — not yet received — blob handle.

9.2. Get segment

Client

`Int32` — `p_operation`

Operation code (`op_get_segment` — 36)

`Int32` — `p_sgmt_blob`

Blob handle

`Int32` — `p_sgmt_length`

Segment length

Maximum length is 32767 for Firebird 2.5 and older, 65535 for Firebird 3.0 and higher.

`Buffer` — `p_sgmt_segment`

Always a zero-length buffer

Server

Generic response — on success, `p_resp_data` is the blob segment

The response buffer in `p_resp_data` contains zero or more segments. Each segment starts with 2-bytes for the length (little-endian), followed by that length of data.

9.2.1. Deviations for protocol version 11

Request flushing and response processing can be deferred.

If `ptype_batch_send` or higher is used, `op_get_segment` can be batched with Create/Open (and other blob operations) by using the *invalid object* handle (`0xFFFF`).

9.3. Put segment

Client

`Int32` — `p_operation`

Operation code (`op_put_segment` — 37)

`Int32` — `p_sgmt_blob`

Blob handle

`Int32` — `p_sgmt_length`

Length of segment data (effectively ignored; possibly only in recent Firebird versions)

`Buffer` — `p_sgmt_segment`

Blob segment

If the blob was created as a segmented blob, the maximum length is 32765 (Firebird 2.5 and older) or 65533 (Firebird 3.0 and higher).

For stream blobs, there is no length limitation other than the maximum buffer length (TODO: verify, might only be for recent versions).

Server

Generic response

9.3.1. Deviations for protocol version 11

Request flushing and response processing can be deferred.

If `ptype_batch_send` or higher is used, `op_put_segment` can be batched with Create/Open (and other blob operations) by using the *invalid object* handle (`0xFFFF`).

9.4. Batch segments

Similar to Put segment, but allows to send multiple segments.

Client

`Int32` — `p_operation`

Operation code (`op_batch_segments` — 44)

`Int32` — `p_sgmt_blob`

Blob handle

`Int32` — `p_sgmt_length`

Length of segment data (effectively ignored; possibly only in recent Firebird versions)

`Buffer` — `p_sgmt_segment`

Blob segments

The buffer can contain one or more segments, which are prefixed with 2 bytes of length (little-endian), followed by the data. The maximum length per segment is 32765 (Firebird 2.5 and older) or 65533 (Firebird 3.0 and higher).

Server

Generic response

9.4.1. Deviations for protocol version 11

Request flushing and response processing can be deferred.

If `ptype_batch_send` or higher is used, `op_batch_segment` can be batched with Create/Open (and other blob operations) by using the *invalid object* handle (`0xFFFF`).

9.5. Seek

Seek is only supported for blobs that were created as a stream blob. Seek is not fully supported for blobs longer than 2 GiB (4 GiB?).

Client

`Int32` — `p_operation`
: Operation code (`op_seek_blob` — `61`)

`Int32` — `p_seek_blob`
: Blob handle

`Int32` — `p_seek_mode`
: Seek mode

`blb_seek_from_head` — `0`	absolute seek from start of blob
`blb_seek_relative` — `1`	relative seek from current position
`blb_seek_from_tail` — `2`	absolute seek from end of blob

`Int32` — `p_seek_offset`
: Offset

Server

Generic response — on success, `p_resp_object` is the current position.

9.5.1. Deviations for protocol version 11

Request flushing and response processing can be deferred.

If `ptype_batch_send` or higher is used, `op_seek_blob` can be batched with Create/Open (and other blob operations) by using the *invalid object* handle (`0xFFFF`).

9.6. Cancel

Cancels and invalidates the blob handle. If this was a newly created blob, the blob is disposed.

Client

`Int32` — `p_operation`
: Operation code (`op_cancel_blob` — `38`)

`Int32` — `p_rlse_object`
: Blob handle

Server

Generic response

9.6.1. Deviations for protocol version 11

Request flushing and response processing can be deferred.

If `ptype_batch_send` or higher is used, `op_cancel_blob` can be batched with Create/Open (and other blob operations) by using the *invalid object* handle (`0xFFFF`). Though doing this probably makes little sense for `op_cancel_blob`.

9.7. Close

Closes and invalidates the blob handle.

Client

`Int32` — `p_operation`

Operation code (`op_close_blob` — `39`)

`Int32` — `p_rlse_object`

Blob handle

Server

Generic response

9.7.1. Deviations for protocol version 11

Request flushing and response processing can be deferred.

If `ptype_batch_send` or higher is used, `op_close_blob` can be batched with Create/Open (and other blob operations) by using the *invalid object* handle (`0xFFFF`).

Chapter 10. Arrays

10.1. Get slice

Client

`Int32` — `p_operation`
: Operation code (`op_get_slice` — 58)

`Int32` — `p_slc_transaction`
: Transaction handle

`Int64` — `p_slc_id`
: Array handle

`Int32` — `p_slc_length`
: Slice length

`Buffer` — `p_slc_sdl`
: Slice descriptor (SDL)

`Buffer` — `p_slc_parameters`
: Slice parameters (always empty?, needs verification)

`Buffer` — `p_slc_slice`
: Slice data (always empty)

Server

Success response: Slice response

Failure response: Generic response

10.2. Put slice

Client

`Int32` — `p_operation`
: Operation code (`op_put_slice` — 59)

`Int32` — `p_slc_transaction`
: transaction handle

`Int64` — `p_slc_id`
: Array handle

`Int32` — `p_slc_length`

Slice length

`Buffer` — `p_slc_sdl`

Slice descriptor (SDL)

`Buffer` — `p_slc_parameters`

Slice parameters (always empty?, needs verification)

`Buffer`` — `p_slc_slice`

Slice data

Server

Generic response — on success, `p_resp_blob_id` is the array handle.

Chapter 11. Batches

Statement batches were introduced in protocol 16 (Firebird 4.0).

11.1. Create

Client

`Int32` — `p_operation`

Operation code (`op_batch_create` — `99`)

`Int32` — `p_batch_statement`

Statement handle

`Buffer` — `p_batch_blr`

Row BLR of batch parameters in `p_batch_data` of Send messages

`UInt32` — `p_batch_msglen`

Message length

`Buffer` — `p_batch_pb`

Batch parameters buffer

If `ptype_lazy` or higher, flushing and response processing can be deferred.

Server

Generic response

11.2. Send messages

Client

`Int32` — `p_operation`

Operation code (`op_batch_msg` — `100`)

`Int32` — `p_batch_statement`

Statement handle

`UInt32` — `p_batch_messages`

Number of messages

`Byte[]` — `p_batch_data`

Batched values (row data repeats `p_batch_messages` times).

The length of each row data follows from the row BLR in `p_batch_blr` of the batch create message. Each row data must be padded to a multiple of 4:

```
<row-data 0>
<0-3 bytes padding> ①
<row-data 1>
<0-3 bytes padding> ①
...
<row-data N>
<0-3 bytes padding> ①
```

① Padding depends on the actual length of the preceding row data.
Generally, the row data should already be a multiple of 4 due to its internal padding, so possibly this is an interpretation error or reverse-engineering mistake on our side.

Server

Generic response

11.3. Execute batch

Client

`Int32` — `p_operation`

Operation code (`op_batch_exec` — `101`)

`Int32` — `p_batch_statement`

Statement handle

`Int32` — `p_batch_transaction`

Transaction handle

Server

Success response:

`Int32` — `p_operation`

Operation code

If operation equals `op_batch_cs` — `103`:

Batch completion state

`Int32` — `p_batch_statement`

Statement handle

`UInt32` — `p_batch_reccount`

Total records count

`UInt32` — `p_batch_updates`

Number of update counters (records updated per each message)

`UInt32` — `p_batch_vectors`

Number of per-message error blocks (message number in batch and status vector of an error processing it)

`UInt32` — `p_batch_errors`

Number of simplified per-message error blocks (message number in batch without status vector)

`Byte[]`

Update counters (records updated per each message), array of `Int32`, length is equal to `p_batch_updates`

Length is `p_batch_updates * 4` bytes long.

`Byte[]`

Detailed info about errors in batch (for each error server sends number of message (`Int32`) and status vector in standard way (exactly like in op_response). Number of such pairs is equal to `p_batch_vectors`.

Length can only be determined by correctly parsing the `<Int32><statusvector>` pairs.

`Byte[]`

Simplified error blocks (for each error server sends number of message (`Int32`) w/o status vector). Used when too many errors took place. Number of elements is equal to `p_batch_errors`.

Length is `p_batch_errors * 4` bytes.

Failure response: Generic response

11.4. Release batch

Client

`Int32` — `p_operation`

Operation code (`op_batch_rls` — `102`)

`Int32` — `p_batch_statement`

Statement handle

Server

Generic response

11.5. Cancel batch

Client

`Int32` — `p_operation`

Operation code (`op_batch_cancel` — `109`)

`Int32` — `p_batch_statement`

Statement handle

Server

Generic response

11.6. Sync batch

Introduced in protocol 17 (Firebird 4.0.1).

Used to force the server to acknowledge previously sent lazy intermediate operations (e.g. `op_batch_msg`, `op_batch_regblob`, `op_batch_blob_stream` and possibly others).

Client

`Int32` — `p_operation`

Operation code (`op_batch_sync` — `110`)

Server

Generic response

11.7. Set default blob parameters

Client

`Int32` — `p_operation`

Operation code (`op_batch_set_bpb` — `106`)

`Int32` — `p_batch_statement`

Statement handle

`Buffer` — `p_batch_blob_bpb`

Default BLOB parameter buffer

Server

Generic response

11.8. Register existing blob

Client

`Int32` — `p_operation`

Operation code (`op_batch_regblob` — `104`)

`Int32` — `p_batch_statement`

Statement handle

`Int64` — `p_batch_exist_id`

Existing BLOB ID

`Int64` — `p_batch_blob_id`

Batch temporary BLOB ID

Server

Generic response

11.9. Stream of BLOB data

This description needs further verification and possibly correction. For example, it seems to mix up Buffer and Byte[]. We're also not able to match some fields to the implementation. For example, the repeated "Record length" seems to be absent, or may actually refer to the `p_batch_blob_data` buffer length.

Client

`Int32` — `p_operation`

Operation code (`op_batch_blob_stream`)

`Int32` — `p_batch_statement`

Statement handle

`Buffer[]` — `p_batch_blob_data`

BLOB stream

This stream is a sequence of blob records. Each blob records contains:

`UInt32`

Record length

The following three fields are called **BLOB header**

`Int64`

Batch temporary BLOB ID

`UInt32`

BLOB size

`UInt32`

BLOB parameters buffer size

`Buffer`

BLOB parameters buffer

`Buffer`

BLOB data (length - BLOB size bytes) (*what does this mean?*)

BLOB headers and records in a stream need not match, i.e. one record may contain many BLOBs and BLOB may stretch from one record to next.

Server

Generic response

11.10. Batch information request

Uses the Generic information request message with:

`p_operation`	`op_info_batch` — 111
`p_info_object`	Statement handle
`p_info_items`	Values of `INF_` constants of `IBatch` (in `IdlFbInterfaces.h`)

Chapter 12. Services

12.1. Attach

Attach to a service. Use message Attachment to database or service with `op_service_attach` — 82.

Note on `p_atch_file`:

Current Firebird versions only support one service: `service_mgr`. Since Firebird 3.0, this can also be an empty string (empty buffer) with the same meaning.

12.2. Detach

Send Detach with `op_service_detach` — 83, followed by Disconnect.

12.3. Start

Although the message looks similar to Generic information request, it has different semantics.

Client

`Int32` — `p_operation`
: Operation code (`op_service_start` — 85)

`Int32` — `p_info_object`
: Unused, always use `0`

`Int32` — `p_info_incarnation`
: Incarnation of object (`0`)

  TODO: Usage and meaning?

`Buffer` — `p_info_items`
: Service parameter buffer

Server

Generic response

12.4. Query service

Although the message looks similar to Generic information request, it has different semantics.

Client

`Int32` — `p_operation`
: Operation code (`op_service_info` — 84)

`Int32` — `p_info_object`

Unused, always use `0`

`Int32` — `p_info_incarnation`

Incarnation of object (`0`)

TODO: Usage and meaning?

`Buffer` — `p_info_items`

Service parameter buffer

`Buffer` — `p_info_recv_items`

Requested information items

`Int32` — `p_info_buffer_length`

Requested information items buffer length

Server

Generic response — on success, `p_resp_data` contains the requested information.

Chapter 13. Events

13.1. Connection request

Client

`Int32` — `p_operation`

Operation code (`op_connect_request` — 53)

`Int32` — `p_req_type`

Unused, but always use `P_REQ_async` (1) for backwards compatibility

`Int32` — `p_req_object`

Unused, always use `0`

`Int32` — `p_req_partner`

Unused, always use `0`

Server

Generic response — with on success:

`p_resp_data`

Aux connection information

This is part of the `sockaddr_in` structure.

It is not in XDR format

`Int16`

Socket family (can be ignored)

`Int16`

Aux connection port

Remaining bytes

To be ignored: always use the hostname or IP address of the original connection.

After a successful response, the client needs to create a connection to the specified port (the “aux connection” or auxiliary connection). The server uses this aux connection for asynchronous notification of events.

13.2. Queue events

Each queued event is notified at most once. After notification, the event needs to be requeued if the client is still interested.

If a queued event was not notified, but the client is no longer interested, it can be cancelled.

Notification of the queued events happens on the aux connection. See Event notification for further details.

Client

Must be sent on the main (database) connection.

`Int32` — `p_operation`
: Operation code (`op_que_events` — 48)

`Int32` — `p_event_database`
: Unused, always use `0`

`Buffer` — `p_event_items`
: Event parameter buffer

  `Byte`
  : Version (`EPB_version1` — 1)

    The following fields are dependent on the version tag.

  `Byte`
  : Length of event name

  `Byte[]`
  : Event name

  `Int32` **(little-endian)**
  : Current known event count (`0` when first queueing, for requeueing use the count of the previous notification)

`Int32` — `p_event_ast`
: Unused, always set `0`

`Int32` — `p_event_arg`
: Unused, always set `0`

`Int32` — `p_event_rid`
: Local event id — generated by the client

Server

Generic response

13.3. Cancel events

Client

Must be sent on the main (database) connection.

`Int32` — `p_operation`
: Operation code (`op_cancel_events` — `49`)

`Int32` — `p_event_database`
: Unused, always use `0`

`Int32` — `p_event_rid`
: Local event id — same id as used to queue the event

Server

Generic response

13.4. Event notification

Event notification happens on the aux connection.

`Int32` — `p_operation`
: Operation code (`op_event` — `52`)

`Int32` — `p_event_database`
: Unused, always `0`

`Buffer` — `p_event_items`
: Event data

  `Byte`
  : Version tag (`EPB_version1` — `1`)

    The following fields are dependent on the version tag.

  `Byte`
  : Length of event name

  `Byte[]`
  : Name of the event

  `Int32` **(little-endian)**
  : Event count

`Int32` — `p_event_ast`
: Unused

`Int32` — `p_event_arg`
: Unused

`Int32` — `p_event_rid`

Local event id — same id as used to queue the event

Chapter 14. Reading and writing row data

For sending rows (on execute) and receiving rows (on execute (`op_execute2`/`op_exec_immediate2`) or fetch), there are two things the client needs:

- The Row BLR (Binary Language Representation)

  The client only needs to write row BLR, it does not need to read it.

- The actual row data of a single row (or parameter)

  The client needs to read and write row data.

14.1. Row BLR (Binary Language Representation)

The row BLR is a description of how the client will send row data with parameters, or expects the server to send the row data.

This BLR does not need to be identical to the `isc_info_sql_select` or `isc_info_sql_bind` description, but if you use other types, they must be convertible, and for parameters, lengths can only be *shorter* than the length of the target type (or at least, longer lengths will result in a string truncation error if the actual value is longer). On the other hand, for output rows, describing a shorter length may result in truncation error if the actual value is longer.

This can, for example, be used to let the user use a different datatype, and offload the conversion to the target datatype to the server. For `CHAR`/`BINARY` parameters, it can make sense — to reduce the transfer size — to send parameter values with their actual length, i.e. without padding with space (0x20) for non-binary, or NUL (0x00) for binary (including `CHAR(n) CHARACTER SET OCTETS`). However, special care must be taken for multibyte character sets like `UTF8`.

The row BLR needs to be sent for:

1. Execute statement
   a. for parameters: in `p_sqldata_blr` (if there are no parameters, an empty buffer suffices)
   b. for output row (`op_execute2`): in `p_sqldata_out_blr`
2. `op_exec_immediate2` (not yet documented)
3. Fetch and Fetch scroll: in `p_sqldata_blr`

   The row BLR only needs to be sent for the first fetch after an execute. If sent on subsequent fetch, the server will ignore it (so changing the BLR between fetches has no effect).

4. Create batch: in `p_batch_blr`

   Describes the parameter row values added to the batch using Send messages.

14.1.1. Row BLR format

We'll describe the row format in terms of its envelope, individual parameters, and their types.

Unless explicitly specified, values are 1 byte.

The values of the `blr_*` constants can be found in `blr.h` (in `[src/]include/firebird/impl/`).

Envelope

```
{ blr_version5 | blr_version4 } ①
blr_begin
blr_message
0 ②
<parameter-count> (Int16 litle endian)
sequence of <parameter-description> ③
blr_end
blr_eoc
```

① Use `blr_version5` for dialect 3, use `blr_version4` for dialect 1.

② We're not sure of the meaning; possibly related to the value of `p_sqldata_message_number` /`p_sqldata_out_message_number`.

③ See Parameter description

Parameter description

```
<type-description> ①
blr_short ②
0 ③
```

① See Type description

② Null indicator

③ We're not sure, possibly "end of parameter"?

Type description

The `SQL_*` constants used below are defined in `sqlda_pub.h` (in `[src/]include/firebird/impl/`). These are the type values reported by `isc_info_sql_type` for the bind descriptions, with the nullable-bit cleared (i.e. `type-value & ~1`).

Simple types

For most types, the type description is a single byte:

```
<blr-type-code>
```

Where *blr-type-code* is one of:

`blr_double`	`SQL_DOUBLE` (`DOUBLE PRECISION`)

blr_float	SQL_FLOAT (FLOAT)
blr_d_float	SQL_D_FLOAT (internal double precision type, not expected to surface in practice)
blr_sql_date	SQL_TYPE_DATE (DATE) Not valid in dialect 1 (blr_version4)
blr_sql_time	SQL_TYPE_TIME (TIME) Not valid in dialect 1 (blr_version4)
blr_timestamp	SQL_TIMESTAMP (TIMESTAMP or dialect 1 DATE)
blr_bool	SQL_BOOLEAN (BOOLEAN)
blr_dec64	SQL_DEC16 (DECIMAL(16))
blr_dec128	SQL_DEC34 (DECIMAL(34))
blr_timestamp_tz	SQL_TIMESTAMP_TZ (TIMESTAMP WITH TIME ZONE)
blr_sql_time_tz	SQL_TIME_TZ (TIME WITH TIME ZONE)
blr_ex_timestamp_tz	SQL_TIMESTAMP_TZ_EX (EXTENDED TIMESTAMP WITH TIME ZONE — special bind-only variant of TIMESTAMP WITH TIME ZONE configurable with SET BIND or isc_dpb_set_bind)
blr_ex_time_tz	SQL_TIME_TZ_EX (EXTENDED TIME WITH TIME ZONE — special bind-only variant of TIME WITH TIME ZONE configurable with SET BIND or isc_dpb_set_bind)

Fixed point numerical types

The fixed point numerical data types (including their parent integer types) are described as:

```
<blr-type-code>
<scale>
```

Where *scale* is `0` for integer types or NUMERIC(p[,s])/DECIMAL(p[,s]) with s absent or `s = 0`. For non-zero s, *scale* is `-s`. This is the value reported for `isc_info_sql_scale`.

Where *blr-type-code* is one of:

blr_short	SQL_SHORT (SMALLINT, or NUMERIC(p[,s]) with `1 <= p <= 4`)
blr_long	SQL_LONG (INTEGER, NUMERIC(p[,s]) with `4 < p <= 9`, or DECIMAL(p[,s]) with `1 < p <= 9`)
blr_int64	SQL_INT64 (BIGINT, or NUMERIC(p[,s])/DECIMAL(p[,s]) with `9 < p <= 18`)
blr_int128	SQL_INT128 (INT128, or NUMERIC(p[,s])/DECIMAL(p[,s]) with `18 < p <= 38`)

Character/binary string types

The current way to generate BLR for character/binary string types is:

```
<blr-type-code>
<character-set-id> ① ②
<collation-id>     ① ③
<length> (Int16 little-endian) ④
```

① An alternative interpretation, instead of 1 byte *character-set-id* and 1 byte *collation-id*, is an Int16 little-endian with the value reported for `isc_info_sql_sub_type` (which for character/binary string types happens to be the character set id and collation id).

② As far as we're aware, you can also use this to specify a different character set than the character set of the target/source type, and the server will convert for you. You may need to adjust *length* accordingly if these character sets have different bytes per character.
We haven't verified if this actually works, so test it first.

③ The collation-id might be ignored by the server, or alternatively it might be advisable to set to `0` (e.g Jaybird does this), we're not entirely sure about this.

④ For `blr_varying2` this is the maximum length in bytes (excluding the length prefix). For `blr_text2` this is the actual length. For variable-length encodings like UTF8, Firebird performs calculations with the *maximum* bytes per character value (so `CHAR(n)` in UTF8 has a length of `4 * N`).

Where *blr-type-code* is one of:

`blr_varying2`	`SQL_VARYING` (`VARCHAR`/`VARBINARY`)
`blr_text2`	`SQL_TEXT` (`CHAR`/`BINARY`)

As far as we know, all Firebird versions should support this encoding.

An alternative — "legacy" — encoding is:

```
<blr-type-code>
<length> (Int16 little-endian)
```

Where *blr-type-code* is one of:

`blr_varying`	`SQL_VARYING` (`VARCHAR`/`VARBINARY`)
`blr_text`	`SQL_TEXT` (`CHAR`/`BINARY`)

Blob types

Used for `SQL_BLOB`. This type description only applies to Firebird 2.5 and higher. When connecting to an older Firebird version, use the type description of Array types.

```
blr_blob2
<subtype> (Int16 little-endian)
```

```
<character-set-id> ① ②
<collation-id>     ① ③
```

① An alternative interpretation, instead of 1 byte *character-set-id* and 1 byte *collation-id*, is an Int16 little-endian with the value reported for `isc_info_sql_scale` (which for `BLOB SUB_TYPE TEXT` happens to be the character set id and collation id).

② As far as we're aware, you can also use this to specify a different character set than the character set of the target/source type, and the server will convert for you. You may need to adjust *length* accordingly if these character sets have different bytes per character.
We haven't verified if this actually works, so test it first.

③ The collation-id might be ignored by the server, or alternatively it might be advisable to set to `0` (e.g Jaybird does this), we're not entirely sure about this.

Array types

This is used for arrays (either `SQL_QUAD` or `SQL_ARRAY`), and for blobs (`SQL_BLOB`) with Firebird 2.1 and earlier.

```
blr_quad
<scale> ①
```

① The value reported for `isc_info_sql_scale` (?meaning unclear?).
Firebird code sometimes sets *scale* to `0` for `SQL_ARRAY` and `SQL_BLOB`.

Null type

The null type, `SQL_NULL`, is only used for parameters (e.g. `? IS [NOT] NULL`). As far as we know, you could use any type, but the following description as a `CHAR(0)` is the most efficient, because you'll send no data at all, and only an Int16 null-indicator in protocol 10 - 12 or a single bit in the null-bitset in protocol 13 and higher.

```
blr_text
0
0
```

14.2. Row data format

The format of the row data is determined by the row BLR the client sent to the server. We describe the format below in terms of the `blr_*` type codes, to tie it in with the BLR.

In Firebird sources, reading/writing of row data is associated `dtype_*` values of `dsc_pub.h` (in `[sql/]include/firebird/impl`) instead; although there is not always a one-to-one correspondence between the `blr_*` and `dtype_*` names, they are generally close enough to be matched easily

Numeric values are encoded in big-endian (or network-order), due to the use of `arch_generic` as the architecture type of the connection (see also Identification (connect)).

Given how the protocol evolved, we will first show the format for protocol 10 to 12 and the format for protocol 13 and higher, and then cover the encoding of individual datatypes.

14.2.1. Row data outline

Outline for protocol 10 - 12

For protocol 10 to 12, the row is sent as:

```
sequence of <column-data>
```

Where *column-data* is:

```
<column-value>   ① ②
<null-indicator> (Int32) ③
```

① See Row data column value.

② Even if the value is null, a `Byte[]` (e.g. all `0x00`) of the correct size must be written for the BLR description. In the case of `blr_varying`/`blr_varying2` a zero-length `Buffer` can be used (i.e. only an `Int32` with value `0`).

③ `0` for non-null, `-1` for null

Outline for protocol 13 and higher

For protocol 13 and higher, the row is sent as:

```
<null-bitset> ① ②
sequence of non-null <column-value> ③
```

① The bitset is a `Byte[]` of length `(column-count + 7) / 8`, padded to a multiple of 4 (i.e. similar to a `Buffer`, but without the length prefixed). When a column is null, its bit must be set, if it is non-null, its bit must be cleared (not set).

② The encoding of the bitset is little-endian, this means the first column is the lowest bit of the first byte, or, if column *n* is null, `(bytes[n/8] & (1<<(n%8))) != 0`.

③ See Row data column value.

In the protocol 13 row data format, only non-null columns are sent. This means that columns that have their null bit set should be skipped (i.e. not written or read).

14.2.2. Row data column value

The following table describes how columns values must be encoded for their `blr_*` type.

BLR type	Encoded form	Remarks
`blr_text` `blr_text2`	`Byte[]` of *length* padded to a multiple of 4	Value must be padded to *length* with 0x20 (space) for non-binary or 0x00 (NUL) for binary. The padding to a multiple of 4 should use the same padding character (?possibly can always be 0x00?). For UTF8, a value is `length / 4` characters long, excess bytes are set to 0x20 (space). For example, a `CHAR(10)` with 10x `a` is encoded as 10x 0x61 followed by 30x 0x20 for a *length* of 40! For UNICODE_FSS, a value is `length / 3` characters long, but Firebird 3.0 and older allow more characters, as long as they fit in *length*. If *length* is `0`, like for `SQL_NULL` as described in Null type, then *no* data is sent.
`blr_varying` `blr_varying2`	`Buffer` with the actual encoded form, with a maximum length of *length*	Contrary to a normal `Buffer`, the padding of the buffer should be 0x20 (space) for non-binary and 0x00 (NUL) for binary (?possibly can always be 0x00?). For UTF8, a value is a maximum of `length / 4` characters long. For example, a `VARCHAR(10)` has a *length* of 40, and a value of 10x `a` is encoded as 10x 0x61, for an actual length of 10 bytes + 2 bytes of padding. For UNICODE_FSS, a value is a maximum of `length / 3` characters long, but Firebird 3.0 and older allow more characters, as long as they fit in *length*.
`blr_quad` `blr_blob2`	2x `Int32` *or* `Int64` *or* `Byte[]` with length 8	Value is the blob id
`blr_short`	`Int32` *or* `Byte[]` of length 4	Although `blr_short` is a 16-bit integer, it is sent as a 32-bit integer
`blr_long`	`Int32` *or* `Byte[]` of length 4	
`blr_int64`	`Int64` *or* `Byte[]` of length 8	
`blr_int128`	`Int128` *or* `Byte[]` of length 16	

BLR type	Encoded form	Remarks
`blr_float`	`Byte[]` of length 4	IEEE 754 binary floating-point “single format” bit layout
`blr_double` `blr_d_float`	`Byte[]` of length 8	IEEE 754 binary floating-point “double format” bit layout
`blr_dec64`	`Byte[]` of length 8	IEEE-754 decimal floating point “decimal64 format” bit layout
`blr_dec128`	`Byte[]` of length 16	IEEE-754 decimal floating point “decimal128 format” bit layout
`blr_bool`	`Byte` padded to a length of 4	`1` for true, `0` for false. Alternative interpretations: • a `Byte[]` of length 4, with first byte `1` for true, `0` for false • An `Int32` little-endian with `1` for true, `0` for false
`blr_sql_date`	`Int32` *or* `Byte[]` of length 4	Not valid in dialect 1 (`blr_version4`) Integer with the number of days since 17 November 1858 (a.k.a. a Modified Julian Date).
`blr_sql_time`	`Int32` *or* `Byte[]` of length 4	Not valid in dialect 1 (`blr_version4`) Integer with the number of 100 microseconds (a.k.a. fractions) since midnight.
`blr_timestamp`	2x `Int32` *or* `Byte[]` of length 8	**`Int32`** date (see also `blr_sql_date`) **`Int32`** time (see also `blr_sql_time`)
`blr_sql_time_tz`	2x `Int32` *or* `Byte[]` of length 8	**`Int32`** time in UTC (see also `blr_sql_time`) **`Int32`** time zone value (offset or time zone id) See also Time zone values

BLR type	Encoded form	Remarks
`blr_timestamp_tz`	3x `Int32` *or* `Byte[]` of length 12	**`Int32`** date in UTC (see also `blr_sql_date`) **`Int32`** time in UTC (see also `blr_sql_time`) **`Int32`** time zone value (offset or time zone id) See also Time zone values
`blr_ex_time_tz`	3x `Int32` *or* `Byte[]` of length 12	**`Int32`** time in UTC (see also `blr_sql_time`) **`Int32`** time zone value (offset or time zone id) **`Int32`** offset value See also Time zone values
`blr_ex_timestamp_tz`	4x `Int32` *or* `Byte[]` of length 16	**`Int32`** date in UTC (see also `blr_sql_date`) **`Int32`** time in UTC (see also `blr_sql_time`) **`Int32`** time zone value (offset or time zone id) **`Int32`** offset value See also Time zone values

Time zone values

The time zone value is an unsigned 16-bit integer encoded as an `Int32`, containing either an offset value or a time zone id.

The "extended" time zone types (`blr_ex_time_tz`/`blr_ex_timestamp_tz`), have a second time zone value that is always an offset. This value can be used when you don't want to derive offsets for named zones yourself. When sending values to the server, this second value can be set to `0`, as the server always uses the first time zone value.

If the value is between `0` and `2878`, it is an offset. You can get the actual offset in minutes with `value - 1439`, making the range `[-1439,+1439]`, or [-23:59,+23:59]. Be aware that in practice, offsets have a

smaller range. If the author recalls correctly, the normal range is something like [-12:00,+14:00].

Time zone libraries may also have their own limits, for example Java's `java.time.ZoneOffset` only allows [-18:00, +18:00] or [-1080,+1080]. We recommend treating values that are out of range of your time zone library as offset +00:00 (i.e. UTC/GMT).

If the value is greater than `2878` (up to `65535`, or `GMT`) it is a time zone id (i.e. `(2878,65535]`). The actual range — for Firebird 5.0.2 — is `[64899,65535]` (ids are assigned downwards from `65535`).

The time zone ids can be mapped to a time zone name using the `RDB$TIME_ZONES` table. New Firebird releases (or possibly ICU time zone data updates) may introduce new time zone ids, but existing ids should be stable.

We recommend keeping your own list instead of querying `RDB$TIME_ZONES`. Depending on your time zone library, it may be necessary to "fix" or correct the mapping, for example by using a different name for the same zone, or to even use UTC or GMT if the zone is unknown to your time zone library. As an example, `65034` is listed with name `Factory` which seems to be ICU specific (?), but is the same as GMT.

Appendix A: External Data Representation (XDR)

The Firebird wire protocol uses XDR for exchange of messages between client and server. The encoding of integers is big-endian (network order).

However, some data **inside** the messages may be little-endian (also known as VAX encoding within Firebird sources).

Appendix B: Data types

`Int32`

Integer 32-bits

In some cases — e.g. object handles, and *some* lengths — this is actually a 16-bit "short" encoded as a 32-bit integer with the high bits zero.

Whether the number should be interpreted as signed or unsigned may depend on the context; when we are sure it's unsigned, we'll generally specify `UInt32` documented next.

`UInt32`

Unsigned integer 32-bits

`Int64`

Integer 64-bits

Alternatively, especially for blob and arrays ids, can be interpreted as two `Int32`, a.k.a. a "quad". Interpretation as a 64-bit integer — even for blob and array ids — is generally simpler, and should not make a difference.

Whether the number should be interpreted as signed or unsigned may depend on the context.

`Buffer`

Composed of

`Int32`

Length of buffer data **without** padding

`Byte[]`

Buffer data

`Byte[]`

Padding of 0 to 3 bytes to align the message to a multiple of 4 (e.g. calculated as `(4 - length) & 3`).

That is, for some `N >= 0`, when the buffer length is:

- `N * 4` bytes → no padding
- `N * 4 + 1` bytes → 3 bytes padding
- `N * 4 + 2` bytes → 2 bytes padding
- `N * 4 + 3` bytes → 1 byte padding

`Byte[]`

An array of bytes

Length follows from another field in the message, from correct parsing of the value, or from other specifics of the message.

`String`

A text string, read or written as a `Buffer`, encoded in the connection character set or some message or context specific character set

Appendix C: Revision history

Revision History			
0.18	18 May 2025	MR	• Documented `op_inline_blob` (protocol 19) and updated execute and fetch documentation • Documented `op_fetch_scroll` (protocol 18) • Documented `op_info_cursor` (protocol 18) • Documented row BLR format • Documented row data format
0.17	17 May 2025	MR	• Reordered revision history, so latest change is at the top • Documented `op_dummy` • Documented `op_event` • Documented `op_exit`/`op_disconnect` on aux connection • Documented `op_reconnect` • Improved protocol 11 descriptions • Documented protocol 12 (`op_cancel`) • Documented the value next to the constant names • Partially documented protocol 11 `op_trusted_auth` (message, not logic) • Documented protocol 13 connect/authentication/attach: `op_accept_data`, `op_cond_accept`, `op_cont_auth`, `op_crypt_key_callback`, `op_crypt`, authentication and attach flow • Described info request message in one place
0.16	13 Apr 2025	MR	• Added Firebird struct field names to message descriptions for reference • Updated, corrected and expanded field descriptions • Documented `op_put_segment` • Added missing field in `p_sgmt_length` in `op_batch_segments` • Documented protocol 11 batching of operations for blobs • Documented protocol 16 timeout (`p_sqldata_timeout`) for `op_execute`/`op_execute2` • Documented protocol 18 cursor flags (`p_sqldata_cursor_flags`) for `op_execute`/`op_execute2` • Documented protocol 19 inline blob size (`p_sqldata_inline_blob_size`) for `op_execute`/`op_execute2` (but not yet `op_inline_blob`!)
0.15	26 Dec 2021	AP	Document batch execution
0.14	04 Aug 2020	MR	Conversion to AsciiDoc, minor copy-editing

Revision History

0.13	13 Sep 2014	Updated and expanded protocol information
0.12	21 Jun 2004	Updated services information.
0.11	20 Jun 2004	• Added new segmentedlist. • Updated Statements.Prepare documentation. • Updated Statements.Execute documentation. • Updated Blobs.GetSegment documentation. • Updated Blobs.Seek documentation.
0.10	19 Jun 2004	Changed rendering of important tags using Paul Vinkenoog fix.
0.9	18 Jun 2004	• Improved segmentedlist usage. • Fixed rendering of important tags.
0.8	17 Jun 2004	Added two new segmented lists.
0.7	16 Jun 2004	Modified document ID to wireprotocol.
0.6	07 Jun 2004	Added events system documentation.
0.5	06 Jun 2004	Fixed issues reported by Paul Vinkenoog.
0.4	05 Jun 2004	Fixed issues reported by Paul Vinkenoog.
0.3	03 Jun 2004	Added new subsections to the Statements section.
0.2	02 Jun 2004	Fixed issues reported by Paul Vinkenoog.
0.1	31 May 2004	First draft for review.