The RFB Protocol

This document is based on "The RFB Protocol" by Tristan Richardson of RealVNC Ltd (formerly of Olivetti Research Ltd / AT&T Labs Cambridge).

Contents

1 Introduction

RFB ("remote framebuffer") is a simple protocol for remote access to graphical user interfaces. Because it works at the framebuffer level it is applicable to all windowing systems and applications, including X11, Windows and Macintosh. RFB is the protocol used in VNC (Virtual Network Computing).

The remote endpoint where the user sits (i.e. the display plus keyboard and/or pointer) is called the RFB client or viewer. The endpoint where changes to the framebuffer originate (i.e. the windowing system and applications) is known as the RFB server.

RFB is truly a "thin client" protocol. The emphasis in the design of the RFB protocol is to make very few requirements of the client. In this way, clients can run on the widest range of hardware, and the task of implementing a client is made as simple as possible.

The protocol also makes the client stateless. If a client disconnects from a given server and subsequently reconnects to that same server, the state of the user interface is preserved. Furthermore, a different client endpoint can be used to connect to the same RFB server. At the new endpoint, the user will see exactly the same graphical user interface as at the original endpoint. In effect, the interface to the user's applications becomes completely mobile. Wherever suitable network connectivity exists, the user can access their own personal applications, and the state of these applications is preserved between accesses from different locations. This provides the user with a familiar, uniform view of the computing infrastructure wherever they go.

2 Display Protocol

The display side of the protocol is based around a single graphics primitive: "put a rectangle of pixel data at a given x,y position". At first glance this might seem an inefficient way of drawing many user interface components. However, allowing various different encodings for the pixel data gives us a large degree of flexibility in how to trade off various parameters such as network bandwidth, client drawing speed and server processing speed.

A sequence of these rectangles makes a *framebuffer update* (or simply *update*). An update represents a change from one valid framebuffer state to another, so in some ways is similar to a frame of video. The rectangles in an update are usually disjoint but this is not necessarily the case.

The update protocol is demand-driven by the client. That is, an update is only sent from the server to the client in response to an explicit request from the client. This gives the protocol an adaptive quality. The slower the client and the network are, the lower the rate of updates becomes. With typical applications, changes to the same area of the framebuffer tend to happen soon after one another. With a slow client and/or network, transient states of the framebuffer can be ignored, resulting in less network traffic and less drawing for the client.

2.1 Screen Model

In its simplest form, the RFB protocol uses a single, rectangular framebuffer. All updates are contained within this buffer and may not extend outside of it. A client with basic functionality simply presents this buffer to the user, padding or cropping it as necessary to fit the user's display.

More advanced RFB clients and servers have the ability to extend this model and add multiple screens. The purpose being to create a server-side representation of the client's physical layout. Applications can use this information to properly position themselves with regard to screen borders.

In the multiple-screen model, there is still just a single framebuffer and framebuffer updates are unaffected by the screen layout. This assures compatibility between basic clients and advanced servers. Screens are added to this model and act like viewports into the framebuffer. A basic client acts as if there is a single screen covering the entire framebuffer.

The server may support up to 255 screens, which must be contained fully within the current framebuffer. Multiple screens may overlap partially or completely.

The client must keep track of the contents of the entire framebuffer, not just the areas currently covered by a screen. Similarly, the server is free to use encodings that rely on contents currently not visible inside any screen. For example it may issue a *CopyRect* rectangle from any part of the framebuffer that should already be known to the client.

The client can request changes to the framebuffer size and screen layout. The server is free to approve or deny these requests at will, but must always inform the client of the result. See the SetDesktopSize message for details.

If the framebuffer size changes, for whatever reason, then all data in it is invalidated and considered undefined. The server must not use any encoding that relies on the previous framebuffer contents. Note however that the semantics for *DesktopSize* are not well-defined and do not follow this behaviour in all server implementations. See the DesktopSize Pseudo-encoding chapter for full details.

Changing only the screen layout does not affect the framebuffer contents. The client must therefore keep track of the current framebuffer dimensions and compare it with the one received in the *ExtendedDesktopSize* rectangle. Only when they differ may it discard the framebuffer contents.

3 Input Protocol

The input side of the protocol is based on a standard workstation model of a keyboard and multi-button pointing device. Input events are simply sent to the server by the client whenever the user presses a key or pointer button, or whenever the pointing device is moved. These input events can also be synthesised from other non-standard I/O devices. For example, a pen-based handwriting recognition engine might generate keyboard events.

If you have an input source that does not fit this standard workstation model, the General Input Interface (gii) protocol extension provides possibilities for input sources with more axes, relative movement and more buttons.

4 Representation of Pixel Data

Initial interaction between the RFB client and server involves a negotiation of the *format* and *encoding* with which pixel data will be sent. This negotiation has been designed to make the job of the client as easy as possible. The bottom line is that the server must always be able to supply pixel data in the form the client wants. However if the client is able to cope equally with several different formats or encodings, it may choose one which is easier for the server to produce.

Pixel *format* refers to the representation of individual colours by pixel values. The most common pixel formats are 24-bit or 16-bit "true colour", where bit-fields within the pixel value translate directly to red, green and blue intensities, and 8-bit "colour map" where an arbitrary mapping can be used to translate from pixel values to the RGB intensities.

Encoding refers to how a rectangle of pixel data will be sent on the wire. Every rectangle of pixel data is prefixed by a header giving the X,Y position of the rectangle on the screen, the width and height of the rectangle, and an *encoding type* which specifies the encoding of the pixel data. The data itself then follows using the specified encoding.

5 Protocol Extensions

There are a number of ways in which the protocol can be extended:

New encodings

A new encoding type can be added to the protocol relatively easily whilst maintaining compatibility with existing clients and servers. Existing servers will simply ignore requests for a new encoding which they don't support. Existing clients will never request the new encoding so will never see rectangles encoded that way.

Pseudo encodings

In addition to genuine encodings, a client can request a "pseudo- encoding" to declare to the server that it supports a certain extension to the protocol. A server which does not support the extension will simply ignore the pseudo-encoding. Note that this means the client must assume that the server does not support the extension until it gets some extension-specific confirmation from the server. See Pseudo-encodings for a description of current pseudo-encodings.

New security types

Adding a new security type gives the ultimate flexibility in modifying the behaviour of the protocol without sacrificing compatibility with existing clients and servers. A client and server which agree on a new security type can effectively talk whatever protocol they like after that, it doesn't necessarily have to be anything like the RFB protocol.

Under no circumstances should you use a different protocol version number. If you use a different protocol version number then you are not RFB / VNC compatible.

All three mechanisms for extensions are handled by RealVNC Ltd. To ensure that you stay compatible with the RFB protocol it is important that you contact RealVNC Ltd to make sure that your encoding types and security types do not clash. Please see the RealVNC website at http://www.realvnc.com for details of how to contact them.

6 String Encodings

The encoding used for strings in the protocol has historically often been unspecified, or has changed between versions of the protocol. As a result, there are a lot of implementations which use different, incompatible encodings. Commonly those encodings have been ISO 8859-1 (also known as Latin-1) or Windows code pages.

It is strongly recommended that new implementations use the UTF-8 encoding for these strings. This allows full unicode support, yet retains good compatibility with older RFB implementations.

New protocol additions that do not have a legacy problem should mandate the UTF-8 encoding to provide full character support and to avoid any issues with ambiguity.

All clients and servers should be prepared to receive invalid UTF-8 sequences at all times. These can occur as a result of historical ambiguity or because of bugs. Neither case should result in lost protocol synchronization.

Handling an invalid UTF-8 sequence is largely dependent on the role that string plays. Modifying the string should only be done when the string is only used in the user interface. It should be obvious in that case that the string has been modified, e.g. by appending a notice to the string.

7 Protocol Messages

The RFB protocol can operate over any reliable transport, either byte- stream or message-based. Conventionally it is used over a TCP/IP connection. There are three stages to the protocol. First is the handshaking phase, the purpose of which is to agree upon the protocol version and the type of security to be used. The second stage is an initialisation phase where the client and server exchange *ClientInit* and *ServerInit* messages. The final stage is the normal protocol interaction. The client can send whichever messages it wants, and may receive messages from the server as a result. All these messages begin with a *message-type* byte, followed by any message-specific data.

The following descriptions of protocol messages use the basic types `U8`, `U16`, `U32`, `S8`, `S16`, `S32`. These represent respectively 8, 16 and 32-bit unsigned integers and 8, 16 and 32-bit signed integers. All multiple byte integers (other than pixel values themselves) are in big endian order (most significant byte first).

However, some protocol extensions use protocol messages that have types that may be in little endian order. These endian agnostic types are `EU16`, `EU32`, `ES16`, `ES32`, with some extension specific indicator of the endianness.

The type `PIXEL` is taken to mean a pixel value of *bytesPerPixel* bytes, where 8 * *bytesPerPixel* is the number of *bits-per-pixel* as agreed by the client and server, either in the *ServerInit* message (ServerInit) or a *SetPixelFormat* message (SetPixelFormat).

7.1 Handshaking Messages

7.1.1 ProtocolVersion

Handshaking begins by the server sending the client a *ProtocolVersion* message. This lets the client know which is the highest RFB protocol version number supported by the server. The client then replies with a similar message giving the version number of the protocol which should actually be used (which may be different to that quoted by the server). A client should never request a protocol version higher than that offered by the server. It is intended that both clients and servers may provide some level of backwards compatibility by this mechanism.

The only published protocol versions at this time are 3.3, 3.7, 3.8 (version 3.5 was wrongly reported by some clients, but this should be interpreted by all servers as 3.3). Addition of a new encoding or pseudo-encoding type does not require a change in protocol version, since a server can simply ignore encodings it does not understand.

The *ProtocolVersion* message consists of 12 bytes interpreted as a string of ASCII characters in the format "`RFB xxx.yyy\n`" where `xxx` and `yyy` are the major and minor version numbers, padded with zeros.

No. of bytes	Value
12	"`RFB 003.003\n`" (hex 52 46 42 20 30 30 33 2e 30 30 33 0a)

or

No. of bytes	Value
12	"`RFB 003.007\n`" (hex 52 46 42 20 30 30 33 2e 30 30 37 0a)

or

No. of bytes	Value
12	"`RFB 003.008\n`" (hex 52 46 42 20 30 30 33 2e 30 30 38 0a)

7.1.2 Security

Once the protocol version has been decided, the server and client must agree on the type of security to be used on the connection.

Version 3.7 onwards

The server lists the security types which it supports:

No. of bytes	Type	Description
1	`U8`	*number-of-security-types*
number-of-security-types	`U8` array	*security-types*

If the server listed at least one valid security type supported by the client, the client sends back a single byte indicating which security type is to be used on the connection:

No. of bytes	Type	Description
1	`U8`	*security-type*

If *number-of-security-types* is zero, then for some reason the connection failed (e.g. the server cannot support the desired protocol version). This is followed by a string describing the reason (where a string is specified as a length followed by that many ASCII characters):

No. of bytes	Type	Description
4	`U32`	*reason-length*
reason-length	`U8` array	*reason-string*

The server closes the connection after sending the *reason-string*.

Version 3.3

The server decides the security type and sends a single word:

No. of bytes	Type	Description
4	`U32`	*security-type*

The *security-type* may only take the value 0, 1 or 2. A value of 0 means that the connection has failed and is followed by a string giving the reason, as described above.

The security types defined in this document are:

Number	Name
0	Invalid
1	None
2	VNC Authentication
16	Tight Security Type
19	VeNCrypt

Other registered security types are:

Number	Name
3-4	RealVNC
5	RA2
6	RA2ne
7-15	RealVNC
17	Ultra
18	TLS
20	SASL
21	MD5 hash authentication
22	xvp
23	Secure Tunnel
24	Integrated SSH
30-35	Apple Inc.
128-255	RealVNC

The official, up-to-date list is maintained by IANA [1].

[1]	*(1, 2, 3, 4)* http://www.iana.org/assignments/rfb/rfb.xml

Once the *security-type* has been decided, data specific to that *security-type* follows (see Security Types for details). At the end of the security handshaking phase, the protocol normally continues with the *SecurityResult* message.

Note that after the security handshaking phase, it is possible that further protocol data is over an encrypted or otherwise altered channel.

7.1.3 SecurityResult

The server sends a word to inform the client whether the security handshaking was successful.

No. of bytes	Type	[Value]	Description
4	`U32`		status:
		0	OK
		1	failed
		2	failed, too many attempts [2]

[2]	Only valid if the Tight Security Type is enabled.

If successful, the protocol passes to the initialisation phase (Initialisation Messages).

Version 3.8 onwards

If unsuccessful, the server sends a string describing the reason for the failure, and then closes the connection:

No. of bytes	Type	Description
4	`U32`	*reason-length*
reason-length	`U8` array	*reason-string*

Version 3.3 and 3.7

If unsuccessful, the server closes the connection.

7.2 Security Types

7.2.1 None

No authentication is needed and protocol data is to be sent unencrypted.

Version 3.8 onwards

The protocol continues with the *SecurityResult* message.

Version 3.3 and 3.7

The protocol passes to the initialisation phase (Initialisation Messages).

7.2.2 VNC Authentication

VNC authentication is to be used and protocol data is to be sent unencrypted. The server sends a random 16-byte challenge:

No. of bytes	Type	Description
16	`U8`	*challenge*

The client encrypts the challenge with DES, using a password supplied by the user as the key, and sends the resulting 16-byte response:

No. of bytes	Type	Description
16	`U8`	*response*

The protocol continues with the *SecurityResult* message.

7.2.3 Tight Security Type

The Tight security type is a generic protocol extension that allows for three things:

Tunneling of data

A tunnel can be e.g. encryption, or indeed a no-op tunnel.

Authentication

The Tight security type allows for flexible authentication of the client, which is typically one of the other security types.

Server capabilities

As a last step the Tight security type extends the ServerInit message and enables the server to let the client know about the server capabilities in terms of encodings and supported message types.

The Tight security type is under the control of the TightVNC project, and any new numbers must be registered with that project before they can be added to any of the lists of Tight capabilities. It is strongly recommended that any messages and security types registered with RealVNC are also registered with the TightVNC project (register security types as Tight authentication capabilities) in order to eliminate clashes as much as is possible. Same thing with new encodings, but in that case the problem is not as severe as the TightVNC project are not using any encodings that are not registered with RealVNC. Please see the TightVNC website at http://www.tightvnc.com/ for details on how to contact the project.

After the Tight security type has been selected, the server starts by sending a list of supported tunnels, in order of preference:

No. of bytes	Type	Description
4	`U32`	*number-of-tunnels*

followed by *number-of-tunnels* repetitions of the following:

No. of bytes	Type	Description
16	`CAPABILITY`	*tunnel*

where `CAPABILITY` is

No. of bytes	Type	Description
4	`S32`	*code*
4	`U8` array	*vendor*
8	`U8` array	*signature*

Note that the *code* is not the only thing identifying a capability. The client must ensure that all members of the structure match before using the capability. Also note that *code* is `U32` in the original Tight documentation and implementation, but since *code* is used to hold encoding numbers we have selected `S32` in this document.

The following tunnel capabilities are registered:

Code	Vendor	Signature	Description
0	"`TGHT`"	"`NOTUNNEL`"	No tunneling

If *number-of-tunnels* is non-zero, the client has to request a tunnel from the list with a tunneling method request:

No. of bytes	Type	Description
4	`S32`	*code*

If *number-of-tunnels* is zero, the client must make no such request, instead the server carries on with sending the list of supported authentication types, in order of preference:

No. of bytes	Type	Description
4	`U32`	*number-of-auth-types*

followed by *number-of-auth-types* repetitions of the following:

No. of bytes	Type	Description
16	`CAPABILITY`	*auth-type*

The following authentication capabilities are registered:

Code	Vendor	Signature	Description
1	"`STDV`"	"`NOAUTH__`"	None
2	"`STDV`"	"`VNCAUTH_`"	VNC Authentication
19	"`VENC`"	"`VENCRYPT`"	VeNCrypt Security
20	"`GTKV`"	"`SASL____`"	Simple Authentication and Security Layer (SASL)
129	"`TGHT`"	"`ULGNAUTH`"	Unix Login Authentication
130	"`TGHT`"	"`XTRNAUTH`"	External Authentication

If *number-of-auth-types* is non-zero, the client has to request an authentication type from the list with an authentication scheme request:

No. of bytes	Type	Description
4	`S32`	*code*

For *code* 1, the protocol the proceeds at security type None and for *code* 2 it proceeds at security type VNC Authentication.

If *number-of-auth-types* is zero, the protocol the proceeds directly at security type None.

Note that the ServerInit message is extended when the Tight security type has been activated.

7.2.4 VeNCrypt

The VeNCrypt security type is a generic authentication method which encapsulates multiple authentication subtypes.

After VeNCrypt security type is selected server sends the highest version of VeNCrypt it can support. Although two versions exist, 0.1 and 0.2, this document describes only newer version 0.2.

No. of bytes	Type	Value	Description
1	`U8`	0	Major version number
1	`U8`	2	Minor version number

Then client sends back the highest VeNCrypt version it can support, up to version that it received from the server.

No. of bytes	Type	Description
1	`U8`	Major version number
1	`U8`	Minor version number

After that server sends one byte response which indicates if everything is OK. Non-zero value means failure and connection will be closed. Zero value means success.

No. of bytes	Type	Description
1	`U8`	Ack

Then server sends list of supported VeNCrypt subtypes.

No. of bytes	Type	Description
1	`U8`	subtypes length
subtypes length	`U32` array	subtypes

Following VeNCrypt subtypes are defined in this document:

Code	Name	Description
256	Plain	Plain authentication (should be never used)
257	TLSNone	TLS encryption with no authentication
258	TLSVnc	TLS encryption with VNC authentication
259	TLSPlain	TLS encryption with Plain authentication
260	X509None	X509 encryption with no authentication
261	X509Vnc	X509 encryption with VNC authentication
262	X509Plain	X509 encryption with Plain authentication
263	TLSSASL	TLS encryption with SASL authentication
264	X509SASL	X509 encryption with SASL authentication

In addition, any of the normal VNC security types (except VeNCrypt) may be sent.

After that client selects one VeNCrypt subtype and sends back the number of that type.

No. of bytes	Type	Description
1	`U32`	Selected VeNCrypt subtype

If client supports none of the VeNCrypt subtypes it terminates connection.

For TLS and X509 subtypes, the server then sends a one byte response which indicates if everything is OK. Non-one value means failure and connection will be closed. One value means success.

No. of bytes	Type	Description
1	`U8`	Ack

When subtype is selected authentication continues as written in particular VeNCrypt subtype description.

7.2.4.1 Subtypes with TLS or X509 prefix

All those subtypes use TLS-encrypted stream and server use anonymous X509 certificate (subtypes with the TLS prefix) or valid X509 certificate (subtypes with the X509 prefix). When session is negotiated, all further traffic is send via this encrypted channel.

After receiving the U32 confirmation of the VeNCrypt subtype, the TLS handshake is performed between the client and server. If the handshake is unsuccessful the connection must be closed and no further RFB protocol messages attempted.

Note about TLS parameters, like algorithm and key length. VeNCrypt doesn't enforce any restriction, setting should be determined by local security policy on client, respective server, side. This also applies for validity of the server certificate, client side can decide if it wants to accept invalid server certificate.

In case TLS handshake is not successful, detailed information of failure can be obtained from underlying TLS stream and both sides must close the connection.

In case TLS handshake is successful and TLS channel is estabilished, VeNCrypt authentication can continue.

7.2.4.2 Subtypes with None suffix

After TLS handshake, authentication is successful and both sides can continue with the SecurityResult message.

7.2.4.3 Subtypes with Vnc suffix

Authentication continues with the VNC Authentication method when TLS handshake is completed.

7.2.4.4 Plain subtype

Client sends the username and password in the following form:

No. of bytes	Type	Description
4	`U32`	*username-length*
4	`U32`	*password-length*
username-length	`U8` array	*username*
password-length	`U8` array	*password*

After that server verifies if supplied credentials are correct and continues with the SecurityResult message.

7.2.4.5 Subtypes with Plain suffix

Authentication continues with the Plain subtype method when TLS handshake is completed.

7.2.4.6 Subtypes with SASL suffix

Authentication continues with the SASL method when TLS handshake is completed.

7.3 Initialisation Messages

Once the client and server are sure that they're happy to talk to one another using the agreed security type, the protocol passes to the initialisation phase. The client sends a *ClientInit* message followed by the server sending a *ServerInit* message.

7.3.1 ClientInit

No. of bytes	Type	Description
1	`U8`	*shared-flag*

Shared-flag is non-zero (true) if the server should try to share the desktop by leaving other clients connected, zero (false) if it should give exclusive access to this client by disconnecting all other clients.

7.3.2 ServerInit

After receiving the *ClientInit* message, the server sends a *ServerInit* message. This tells the client the width and height of the server's framebuffer, its pixel format and the name associated with the desktop:

No. of bytes	Type	Description
2	`U16`	*framebuffer-width*
2	`U16`	*framebuffer-height*
16	`PIXEL_FORMAT`	*server-pixel-format*
4	`U32`	*name-length*
name-length	`U8` array	*name-string*

The text encoding used for *name-string* is historically undefined but it is strongly recommended to use UTF-8 (see String Encodings for more details).

`PIXEL_FORMAT` is defined as:

No. of bytes	Type	Description
1	`U8`	*bits-per-pixel*
1	`U8`	*depth*
1	`U8`	*big-endian-flag*
1	`U8`	*true-colour-flag*
2	`U16`	*red-max*
2	`U16`	*green-max*
2	`U16`	*blue-max*
1	`U8`	*red-shift*
1	`U8`	*green-shift*
1	`U8`	*blue-shift*
3		*padding*

Server-pixel-format specifies the server's natural pixel format. This pixel format will be used unless the client requests a different format using the *SetPixelFormat* message (SetPixelFormat).

Bits-per-pixel is the number of bits used for each pixel value on the wire. This must be greater than or equal to the depth which is the number of useful bits in the pixel value. Currently *bits-per-pixel* must be 8, 16 or 32. Less than 8-bit pixels are not yet supported. *Big-endian-flag* is non-zero (true) if multi-byte pixels are interpreted as big endian. Of course this is meaningless for 8 bits-per-pixel.

If *true-colour-flag* is non-zero (true) then the last six items specify how to extract the red, green and blue intensities from the pixel value. *Red-max* is the maximum red value (= 2^n - 1 where *n* is the number of bits used for red). Note this value is always in big endian order. *Red-shift* is the number of shifts needed to get the red value in a pixel to the least significant bit. *Green-max*, *green-shift* and *blue-max*, *blue-shift* are similar for green and blue. For example, to find the red value (between 0 and *red-max*) from a given pixel, do the following:

- Swap the pixel value according to *big-endian-flag* (e.g. if *big-endian-flag* is zero (false) and host byte order is big endian, then swap).
- Shift right by *red-shift*.
- AND with *red-max* (in host byte order).

If *true-colour-flag* is zero (false) then the server uses pixel values which are not directly composed from the red, green and blue intensities, but which serve as indices into a colour map. Entries in the colour map are set by the server using the *SetColourMapEntries* message (SetColourMapEntries).

If the Tight Security Type is activated, the server init message is extended with an interaction capabilities section:

No. of bytes	Type	[Value]	Description
2	`U16`		*number-of-server-messages*
2	`U16`		*number-of-client-messages*
2	`U16`		*number-of-encodings*
2	`U16`	0	*padding*

followed by *number-of-server-messages* repetitions of the following:

No. of bytes	Type	Description
16	`CAPABILITY`	*server-message*

followed by *number-of-client-messages* repetitions of the following:

No. of bytes	Type	Description
16	`CAPABILITY`	*client-message*

followed by *number-of-encodings* repetitions of the following:

No. of bytes	Type	Description
16	`CAPABILITY`	*encoding*

The following *server-message* capabilities are registered:

Code	Vendor	Signature	Description
130	" `TGHT` "	" `FTS_LSDT` "	File List Data
131	" `TGHT` "	" `FTS_DNDT` "	File Download Data
132	" `TGHT` "	" `FTS_UPCN` "	File Upload Cancel
133	" `TGHT` "	" `FTS_DNFL` "	File Download Failed
150	" `TGHT` "	" `CUS_EOCU` "	End Of Continuous Updates
253	" `GGI_` "	" `GII_SERV` "	gii Server Message

The following *client-message* capabilities are registered:

Code	Vendor	Signature	Description
130	" `TGHT` "	" `FTC_LSRQ` "	File List Request
131	" `TGHT` "	" `FTC_DNRQ` "	File Download Request
132	" `TGHT` "	" `FTC_UPRQ` "	File Upload Request
133	" `TGHT` "	" `FTC_UPDT` "	File Upload Data
134	" `TGHT` "	" `FTC_DNCN` "	File Download Cancel
135	" `TGHT` "	" `FTC_UPFL` "	File Upload Failed
136	" `TGHT` "	" `FTC_FCDR` "	File Create Directory Request
150	" `TGHT` "	" `CUC_ENCU` "	Enable/Disable Continuous Updates
151	" `TGHT` "	" `VRECTSEL` "	Video Rectangle Selection
253	" `GGI_` "	" `GII_CLNT` "	gii Client Message

The following *encoding* capabilities are registered:

Code	Vendor	Signature	Description
0	" `STDV` "	" `RAW_____` "	Raw Encoding
1	" `STDV` "	" `COPYRECT` "	CopyRect Encoding
2	" `STDV` "	" `RRE_____` "	RRE Encoding
4	" `STDV` "	" `CORRE___` "	CoRRE Encoding
5	" `STDV` "	" `HEXTILE_` "	Hextile Encoding
6	" `TRDV` "	" `ZLIB____` "	ZLib Encoding
7	" `TGHT` "	" `TIGHT___` "	Tight Encoding
8	" `TRDV` "	" `ZLIBHEX_` "	ZLibHex Encoding
-32	" `TGHT` "	" `JPEGQLVL` "	JPEG Quality Level Pseudo-encoding
-223	" `TGHT` "	" `NEWFBSIZ` "	DesktopSize Pseudo-encoding (New FB Size)
-224	" `TGHT` "	" `LASTRECT` "	LastRect Pseudo-encoding
-232	" `TGHT` "	" `POINTPOS` "	Pointer Position
-239	" `TGHT` "	" `RCHCURSR` "	Cursor Pseudo-encoding (Rich Cursor)
-240	" `TGHT` "	" `X11CURSR` "	X Cursor Pseudo-encoding
-256	" `TGHT` "	" `COMPRLVL` "	Compression Level Pseudo-encoding
-305	" `GGI_` "	" `GII_____` "	gii Pseudo-encoding
-512	" `TRBO` "	" `FINEQLVL` "	JPEG Fine-Grained Quality Level Pseudo-encoding
-768	" `TRBO` "	" `SSAMPLVL` "	JPEG Subsampling Level Pseudo-encoding

Note that the server need not (but it may) list the " `RAW_____` " capability since it must be supported anyway.

7.4 Client to Server Messages

The client to server message types that all servers must support are:

Number	Name
0	SetPixelFormat
2	SetEncodings
3	FramebufferUpdateRequest
4	KeyEvent
5	PointerEvent
6	ClientCutText

Optional message types are:

Number	Name
7	FileTransfer
8	SetScale
9	SetServerInput
10	SetSW
11	TextChat
12	KeyFrameRequest
13	KeepAlive
14	Possibly used in UltraVNC
15	SetScaleFactor
16-19	Possibly used in UltraVNC
20	RequestSession
21	SetSession
80	NotifyPluginStreaming
127	VMWare
128	Car Connectivity
150	EnableContinuousUpdates
248	ClientFence
249	OLIVE Call Control
250	xvp Client Message
251	SetDesktopSize
252	tight
253	gii Client Message
254	VMWare
255	QEMU Client Message

The official, up-to-date list is maintained by IANA [1].

Note that before sending a message with an optional message type a client must have determined that the server supports the relevant extension by receiving some extension-specific confirmation from the server.

7.4.1 SetPixelFormat

Sets the format in which pixel values should be sent in *FramebufferUpdate* messages. If the client does not send a *SetPixelFormat* message then the server sends pixel values in its natural format as specified in the ServerInit message (ServerInit).

If *true-colour-flag* is zero (false) then this indicates that a "colour map" is to be used. The server can set any of the entries in the colour map using the *SetColourMapEntries* message (SetColourMapEntries). Immediately after the client has sent this message the colour map is empty, even if entries had previously been set by the server.

Note that a client must not have an outstanding *FramebufferUpdateRequest* when it sends *SetPixelFormat* as it would be impossible to determine if the next *FramebufferUpdate* is using the new or the previous pixel format.

No. of bytes	Type	[Value]	Description
1	`U8`	0	*message-type*
3			*padding*
16	`PIXEL_FORMAT`		*pixel-format*

where `PIXEL_FORMAT` is as described in ServerInit:

No. of bytes	Type	Description
1	`U8`	*bits-per-pixel*
1	`U8`	*depth*
1	`U8`	*big-endian-flag*
1	`U8`	*true-colour-flag*
2	`U16`	*red-max*
2	`U16`	*green-max*
2	`U16`	*blue-max*
1	`U8`	*red-shift*
1	`U8`	*green-shift*
1	`U8`	*blue-shift*
3		*padding*

7.4.2 SetEncodings

Sets the encoding types in which pixel data can be sent by the server. The order of the encoding types given in this message is a hint by the client as to its preference (the first encoding specified being most preferred). The server may or may not choose to make use of this hint. Pixel data may always be sent in *raw* encoding even if not specified explicitly here.

In addition to genuine encodings, a client can request "pseudo-encodings" to declare to the server that it supports certain extensions to the protocol. A server which does not support the extension will simply ignore the pseudo-encoding. Note that this means the client must assume that the server does not support the extension until it gets some extension-specific confirmation from the server.

See Encodings for a description of each encoding and Pseudo-encodings for the meaning of pseudo-encodings.

No. of bytes	Type	[Value]	Description
1	`U8`	2	*message-type*
1			*padding*
2	`U16`		*number-of-encodings*

followed by *number-of-encodings* repetitions of the following:

No. of bytes	Type	Description
4	`S32`	*encoding-type*

7.4.3 FramebufferUpdateRequest

Notifies the server that the client is interested in the area of the framebuffer specified by *x-position*, *y-position*, *width* and *height*. The server usually responds to a *FramebufferUpdateRequest* by sending a *FramebufferUpdate*. Note however that a single *FramebufferUpdate* may be sent in reply to several *FramebufferUpdateRequests*.

The server assumes that the client keeps a copy of all parts of the framebuffer in which it is interested. This means that normally the server only needs to send incremental updates to the client.

However, if for some reason the client has lost the contents of a particular area which it needs, then the client sends a *FramebufferUpdateRequest* with *incremental* set to zero (false). This requests that the server send the entire contents of the specified area as soon as possible. The area will not be updated using the *CopyRect* encoding.

If the client has not lost any contents of the area in which it is interested, then it sends a *FramebufferUpdateRequest* with *incremental* set to non-zero (true). If and when there are changes to the specified area of the framebuffer, the server will send a *FramebufferUpdate*. Note that there may be an indefinite period between the *FramebufferUpdateRequest* and the *FramebufferUpdate*.

In the case of a fast client, the client may want to regulate the rate at which it sends incremental *FramebufferUpdateRequests* to avoid hogging the network.

No. of bytes	Type	[Value]	Description
1	`U8`	3	*message-type*
1	`U8`		*incremental*
2	`U16`		*x-position*
2	`U16`		*y-position*
2	`U16`		*width*
2	`U16`		*height*

A request for an area that partly falls outside the current framebuffer must be cropped so that it fits within the framebuffer dimensions.

Note that an empty area can still solicit a *FramebufferUpdate* even though that update will only contain pseudo-encodings.

7.4.4 KeyEvent

A key press or release. *Down-flag* is non-zero (true) if the key is now pressed, zero (false) if it is now released. The key itself is specified using the "keysym" values defined by the X Window System.

No. of bytes	Type	[Value]	Description
1	`U8`	4	*message-type*
1	`U8`		*down-flag*
2			*padding*
4	`U32`		*key*

Auto repeating of keys when a key is held down should be handled on the client. The rationale being that high latency on the network can make it seem like a key is being held for a very long time, yet the problem is that the *KeyEvent* message releasing the button has been delayed.

The client should send only repeated "down" *KeyEvent* messages, no "up" messages, when a key is automatically repeated. This allows the server to tell the difference between automatic repeat and actual repeated entry by the user.

For most ordinary keys, the "keysym" is the same as the corresponding ASCII value. For full details, see The Xlib Reference Manual, published by O'Reilly & Associates, or see the header file `<X11/keysymdef.h>` from any X Window System installation. Some other common keys are:

Key name	Keysym value
BackSpace	0xff08
Tab	0xff09
Return or Enter	0xff0d
Escape	0xff1b
Insert	0xff63
Delete	0xffff
Home	0xff50
End	0xff57
Page Up	0xff55
Page Down	0xff56
Left	0xff51
Up	0xff52
Right	0xff53
Down	0xff54
F1	0xffbe
F2	0xffbf
F3	0xffc0
F4	0xffc1
...	...
F12	0xffc9
Shift (left)	0xffe1
Shift (right)	0xffe2
Control (left)	0xffe3
Control (right)	0xffe4
Meta (left)	0xffe7
Meta (right)	0xffe8
Alt (left)	0xffe9
Alt (right)	0xffea

The interpretation of keysyms is a complex area. In order to be as widely interoperable as possible the following guidelines should be used:

- The "shift state" (i.e. whether either of the Shift keysyms are down) should only be used as a hint when interpreting a keysym. For example, on a US keyboard the '#' character is shifted, but on a UK keyboard it is not. A server with a US keyboard receiving a '#' character from a client with a UK keyboard will not have been sent any shift presses. In this case, it is likely that the server will internally need to "fake" a shift press on its local system, in order to get a '#' character and not, for example, a '3'.
- The difference between upper and lower case keysyms is significant. This is unlike some of the keyboard processing in the X Window System which treats them as the same. For example, a server receiving an uppercase 'A' keysym without any shift presses should interpret it as an uppercase 'A'. Again this may involve an internal "fake" shift press.
- Servers should ignore "lock" keysyms such as CapsLock and NumLock where possible. Instead they should interpret each character-based keysym according to its case.

- Unlike Shift, the state of modifier keys such as Control and Alt should be taken as modifying the interpretation of other keysyms. Note that there are no keysyms for ASCII control characters such as ctrl-a; these should be generated by viewers sending a Control press followed by an 'a' press.
- On a viewer where modifiers like Control and Alt can also be used to generate character-based keysyms, the viewer may need to send extra "release" events in order that the keysym is interpreted correctly. For example, on a German PC keyboard, ctrl-alt-q generates the '@' character. In this case, the viewer needs to send "fake" release events for Control and Alt in order that the '@' character is interpreted correctly (ctrl-alt-@ is likely to mean something completely different to the server).
- There is no universal standard for "backward tab" in the X Window System. On some systems shift+tab gives the keysym "ISO Left Tab", on others it gives a private "BackTab" keysym and on others it gives "Tab" and applications tell from the shift state that it means backward-tab rather than forward-tab. In the RFB protocol the latter approach is preferred. Viewers should generate a shifted Tab rather than ISO Left Tab. However, to be backwards-compatible with existing viewers, servers should also recognise ISO Left Tab as meaning a shifted Tab.

7.4.5 PointerEvent

Indicates either pointer movement or a pointer button press or release. The pointer is now at (*x-position*, *y-position*), and the current state of buttons 1 to 8 are represented by bits 0 to 7 of *button-mask* respectively, 0 meaning up, 1 meaning down (pressed).

On a conventional mouse, buttons 1, 2 and 3 correspond to the left, middle and right buttons on the mouse. On a wheel mouse, each step of the wheel is represented by a press and release of a certain button. Button 4 means up, button 5 means down, button 6 means left and button 7 means right.

No. of bytes	Type	[Value]	Description
1	`U8`	5	*message-type*
1	`U8`		*button-mask*
2	`U16`		*x-position*
2	`U16`		*y-position*

The QEMU Pointer Motion Change Pseudo-encoding allows for the negotiation of an alternative interpretation for the *x-position* and *y-position* fields, as relative deltas.

7.4.6 ClientCutText

The client has new ISO 8859-1 (Latin-1) text in its cut buffer. Ends of lines are represented by the linefeed / newline character (value 10) alone. No carriage-return (value 13) is needed.

No. of bytes	Type	[Value]	Description
1	`U8`	6	*message-type*
3			*padding*
4	`U32`		*length*
length	`U8` array		*text*

See also Extended Clipboard Pseudo-Encoding which modifies the behaviour of this message.

7.4.7 EnableContinuousUpdates

This message informs the server to switch between only sending FramebufferUpdate messages as a result of a FramebufferUpdateRequest message, or sending `FramebufferUpdate` messages continuously.

Note that there is currently no way to determine if the server supports this message except for using the Tight Security Type authentication.

No. of bytes	Type	[Value]	Description
1	`U8`	150	*message-type*
1	`U8`		*enable-flag*
2	`U16`		*x-position*
2	`U16`		*y-position*
2	`U16`		*width*
2	`U16`		*height*

If *enable-flag* is non-zero, then the server can start sending `FramebufferUpdate` messages as needed for the area specified by *x-position*, *y-position*, *width*, and *height*. If continuous updates are already active, then they must remain active active and the coordinates must be replaced with the last message seen.

If *enable-flag* is zero, then the server must only send `FramebufferUpdate` messages as a result of receiving `FramebufferUpdateRequest` messages. The server must also immediately send out a EndOfContinuousUpdates message. This message must be sent out even if continuous updates were already disabled.

The server must ignore all incremental update requests (`FramebufferUpdateRequest` with *incremental* set to non-zero) as long as continuous updates are active. Non-incremental update requests must however be honored, even if the area in such a request does not overlap the area specified for continuous updates.

7.4.8 ClientFence

A client supporting the *Fence* extension sends this to request a synchronisation of the data stream.

No. of bytes	Type	[Value]	Description
1	`U8`	248	*message-type*
3			*padding*
4	`U32`		*flags*
1	`U8`		*length*
length	`U8` array		*payload*

The *flags* byte informs the server if this is a new request, or a response to a server request sent earlier, as well as what kind of synchronisation that is desired. The server should not delay the response more than necessary, even if the synchronisation requirements would allow it.

Bit	Description
0	**BlockBefore**
1	**BlockAfter**
2	**SyncNext**
3-30	Currently unused
31	**Request**

The server should respond with a ServerFence with the **Request** bit cleared, as well as clearing any bits it does not understand. The remaining bits should remain set in the response. This allows the client to determine which flags the server supports when new ones are defined in the future.

BlockBefore

All messages preceding this one must have finished processing and taken effect before the response is sent.

Messages following this one are unaffected and may be processed in any order the protocol permits, even before the response is sent.

BlockAfter

All messages following this one must not start processing until the response is sent.

Messages preceding this one are unaffected and may be processed in any order the protocol permits, even being delayed until after the response is sent.

SyncNext

The message following this one must be executed in an atomic manner so that anything preceding the fence response **must not** be affected by the message, and anything following the fence response **must** be affected by the message.

Anything unaffected by the following message can be sent at any time the protocol permits.

The primary purpose of this synchronisation is to allow safe usage of stream altering commands such as SetPixelFormat, which would impose strict ordering on FramebufferUpdate messages even with asynchrounous extensions such as the ContinuousUpdates Pseudo-encoding.

If **BlockAfter** is also set then the interaction between the two flags can be ambiguous. In this case we relax the requirement for **BlockAfter** and allow the following message (the one made atomic by **SyncNext**) to be processed before a response is sent. All messages after that first one are still subjected to the semantics of **BlockAfter** however. The behaviour will be similar to the following series of messages:

> 1. *ClientFence* with **SyncNext**
> 2. *message made atomic*
> 3. *ClientFence* with **BlockAfter**

Request

Indicates that this is a new request and that a response is expected. If this bit is cleared then this message is a response to an earlier request.

The client can also include a chunk of data to differentiate between responses and to avoid keeping state. This data is specified using *length* and *payload*. The size of this data is limited to 64 bytes in order to minimise the disturbance to highly parallel clients and servers.

7.4.9 xvp Client Message

A client supporting the *xvp* extension sends this to request that the server initiate a clean shutdown, clean reboot or abrupt reset of the system whose framebuffer the client is displaying.

No. of bytes	Type	[Value]	Description
1	U8	250	*message-type*
1			*padding*
1	U8	1	*xvp-extension-version*
1	U8		*xvp-message-code*

The possible values for *xvp-message-code* are: 2 - XVP_SHUTDOWN, 3 - XVP_REBOOT, and 4 - XVP_RESET. The client must have already established that the server supports this extension, by requesting the xvp Pseudo-encoding.

7.4.10 SetDesktopSize

Requests a change of desktop size. This message is an extension and may only be sent if the client has previously received an *ExtendedDesktopSize* rectangle.

The server must send an *ExtendedDesktopSize* rectangle for every *SetDesktopSize* message received. Several rectangles may be sent in a single *FramebufferUpdate* message, but the rectangles must not be merged or reordered in any way. Note that rectangles sent for other reasons may be interleaved with the ones generated as a result of *SetDesktopSize* messages.

Upon a successful request the server must send an *ExtendedDesktopSize* rectangle to the requesting client with the exact same information the client provided in the corresponding *SetDesktopSize* message. *x-position* must be set to 1, indicating a client initiated event, and *y-position* must be set to 0, indicating success.

The server must also send an *ExtendedDesktopSize* rectangle to all other connected clients, but with *x-position* set to 2, indicating a change initiated by another client.

If the server can not or will not satisfy the request, it must send an *ExtendedDesktopSize* rectangle to the requesting client with *x-position* set to 1 and *y-position* set to the relevant error code. All remaining fields are undefined, although the basic structure must still be followed. The server must not send an *ExtendedDesktopSize* rectangle to any other connected clients.

All *ExtendedDesktopSize* rectangles that are sent as a result of a *SetDesktopSize* message should be sent as soon as possible.

No. of bytes	Type	[Value]	Description
1	`U8`	251	*message-type*
1			*padding*
2	`U16`		*width*
2	`U16`		*height*
1	`U8`		*number-of-screens*
1			*padding*
number-of-screens * 16	`SCREEN` array		*screens*

The *width* and *height* indicates the framebuffer size requested. This structure is followed by *number-of-screens* number of `SCREEN` structures, which is defined in ExtendedDesktopSize Pseudo-encoding:

No. of bytes	Type	Description
4	`U32`	*id*
2	`U16`	*x-position*
2	`U16`	*y-position*
2	`U16`	*width*
2	`U16`	*height*
4	`U32`	*flags*

The *id* field must be preserved upon modification as it determines the difference between a moved screen and a newly created one. The client should make every effort to preserve the fields it does not wish to modify, including any unknown *flags* bits.

7.4.11 gii Client Message

This message is an extension and may only be sent if the client has previously received a gii Server Message confirming that the server supports the General Input Interface extension.

7.4.11.1 Version

The client response to a *gii* Version message from the server is the following response:

No. of bytes	Type	[Value]	Description
1	`U8`	253	*message-type*
1	`U8`	1 or 129	*endian-and-sub-type*
2	`EU16`	2	*length*
2	`EU16`	1	*version*

endian-and-sub-type is a bit-field with the leftmost bit indicating big endian if set, and little endian if cleared. The rest of the bits are the actual message sub type.

version is set by the client and ultimately decides the version of *gii* protocol extension to use. It should be in the range given by the server in the *gii* Version message. If the server doesn't support any version that the client supports, the client should instead stop using the *gii* extension at this point.

7.4.11.2 Device Creation

After establishing the *gii* protocol extension version, the client proceeds by requesting creation of one or more devices.

No. of bytes	Type	[Value]	Description
1	`U8`	253	*message-type*
1	`U8`	2 or 130	*endian-and-sub-type*
2	`EU16`	56 + *num-valuators* * 116	*length*
31	`U8` array		*device-name*
1	`U8`	0	*nul-terminator*
4	`EU32`		*vendor-id*
4	`EU32`		*product-id*
4	`EVENT_MASK`		*can-generate*
4	`EU32`		*num-registers*
4	`EU32`		*num-valuators*
4	`EU32`		*num-buttons*
num-valuators * 116	`VALUATOR`		

endian-and-sub-type is a bit-field with the leftmost bit indicating big endian if set, and little endian if cleared. The rest of the bits are the actual message sub type.

`EVENT_MASK` is a bit-field indicating which events the device can generate.

Value	Bit name
0x00000020	Key press
0x00000040	Key release
0x00000080	Key repeat
0x00000100	Pointer relative
0x00000200	Pointer absolute
0x00000400	Pointer button press
0x00000800	Pointer button release
0x00001000	Valuator relative
0x00002000	Valuator absolute

and `VALUATOR` is

No. of bytes	Type	[Value]	Description
4	`EU32`		*index*
74	`U8` array		*long-name*
1	`U8`	0	*nul-terminator*
4	`U8` array		*short-name*
1	`U8`	0	*nul-terminator*
4	`ES32`		*range-min*
4	`ES32`		*range-center*
4	`ES32`		*range-max*
4	`EU32`		*SI-unit*
4	`ES32`		*SI-add*
4	`ES32`		*SI-mul*
4	`ES32`		*SI-div*
4	`ES32`		*SI-shift*

The *SI-unit* field is defined as:

Number	SI-unit	Description
0		unknown
1	s	time
2	1/s	frequency
3	m	length
4	m/s	velocity
5	m/s^2	acceleration
6	rad	angle
7	rad/s	angular velocity
8	rad/s^2	angular acceleration
9	m^2	area
10	m^3	volume
11	kg	mass
12	N (kg*m/s^2)	force
13	N/m^2 (Pa)	pressure
14	Nm	torque
15	Nm, VAs, J	energy
16	Nm/s, VA, W	power
17	K	temperature
18	A	current
19	V (kg*m^2/(As^3))	voltage
20	V/A (Ohm)	resistance
21	As/V	capacity
22	Vs/A	inductivity

The *SI-add*, *SI-mul*, *SI-div* and *SI-shift* fields of the `VALUATOR` indicate how the raw value should be translated to the SI-unit using the below formula.

> float SI = (float) (SI_add + value[n]) * (float) SI_mul / (float) SI_div * pow(2.0, SI_shift);

Setting *SI-mul* to zero indicates that the valuator is non-linear or that the factor is unknown.

7.4.11.3 Device Destruction

The client can destroy a device with a device destruct message.

No. of bytes	Type	[Value]	Description
1	`U8`	253	*message-type*
1	`U8`	3 or 131	*endian-and-sub-type*
2	`EU16`	4	*length*
4	`EU32`		*device-origin*

endian-and-sub-type is a bit-field with the leftmost bit indicating big endian if set, and little endian if cleared. The rest of the bits are the actual message sub type.

device-origin is the handle retrieved with a prior device creation request.

7.4.11.4 Injecting Events

No. of bytes	Type	[Value]	Description
1	`U8`	253	*message-type*
1	`U8`	0 or 128	*endian-and-sub-type*
2	`EU16`		*length*

followed by *length* bytes of `EVENT` entries

endian-and-sub-type is a bit-field with the leftmost bit indicating big endian if set, and little endian if cleared. The rest of the bits are the actual message sub type.

`EVENT` is one of `KEY_EVENT`, `PTR_MOVE_EVENT`, `PTR_BUTTON_EVENT` and `VALUATOR_EVENT`.

`KEY_EVENT` is:

No. of bytes	Type	[Value]	Description
1	`U8`	24	*event-size*
1	`U8`	5, 6 or 7	*event-type*
2	`EU16`		*padding*
4	`EU32`		*device-origin*
4	`EU32`		*modifiers*
4	`EU32`		*symbol*
4	`EU32`		*label*
4	`EU32`		*button*

The possible values for *event-type* are: 5 - key pressed, 6 - key released and 7 - key repeat. XXX describe *modifiers*, *symbol*, *label* and *button*. Meanwhile, see http://www.ggi-project.org/documentation/libgii/current/gii_key_event.3.html for details.

device-origin is the handle retrieved with a prior device creation request.

`PTR_MOVE_EVENT` is:

No. of bytes	Type	[Value]	Description
1	`U8`	24	*event-size*
1	`U8`	8 or 9	*event-type*
2	`EU16`		*padding*
4	`EU32`		*device-origin*
4	`ES32`		*x*
4	`ES32`		*y*
4	`ES32`		*z*
4	`ES32`		*wheel*

The possible values for *event-type* are: 8 - pointer relative and 9 - pointer absolute.

device-origin is the handle retrieved with a prior device creation request.

`PTR_BUTTON_EVENT` is:

No. of bytes	Type	[Value]	Description
1	`U8`	12	*event-size*
1	`U8`	10 or 11	*event-type*
2	`EU16`		*padding*
4	`EU32`		*device-origin*
4	`EU32`		*button-number*

The possible values for *event-type* are: 10 - pointer button press and 11 - pointer button release.

device-origin is the handle retrieved with a prior device creation request.

button-number 1 is the primary or left button, *button-number* 2 is the secondary or right button and *button-number* 3 is the tertiary or middle button. Other values for *button-number* are also valid.

`VALUATOR_EVENT` is:

No. of bytes	Type	[Value]	Description
1	`U8`	16 + 4 * *count*	*event-size*
1	`U8`	12 or 13	*event-type*
2	`EU16`		*padding*
4	`EU32`		*device-origin*
4	`EU32`		*first*
4	`EU32`		*count*
4 * *count*	`ES32` array		*value*

The possible values for *event-type* are: 12 - relative valuator and 13 - absolute valuator.

device-origin is the handle retrieved with a prior device creation request.

The event reports *count* valuators starting with *first*.

7.4.12 QEMU Client Message

This message may only be sent if the client has previously received a *FrameBufferUpdate* that confirms support for the intended *submessage-type*. Every `QEMU Client Message` begins with a standard header

No. of bytes	Type	[Value]	Description
1	`U8`	255	*message-type*
1	`U8`		*submessage-type*

This header is then followed by arbitrary data whose format is determined by the *submessage-type*. Possible values for *submessage-type* and their associated pseudo encodings are

Submessage Type	Pseudo Encoding	Description
0	-258	Extended key events
1	-259	Audio

7.4.12.1 QEMU Extended Key Event Message

This submessage allows the client to send an extended key event containing a keycode, in addition to a keysym. The advantage of providing the keycode is that it enables the server to interpret the key event independently of the clients' locale specific keymap. This can be important for virtual desktops whose key input device requires scancodes, for example, virtual machines emulating a PS/2 keycode. Prior to this extension, RFB servers for such virtualization software would have to be configured with a keymap matching the client. With this extension it is sufficient for the guest operating system to be configured with the matching keymap. The VNC server is keymap independent.

The full message is:

No. of bytes	Type	[Value]	Description
1	`U8`	255	*message-type*
1	`U8`	0	*submessage-type*
2	`U16`		*down-flag*
4	`U32`		*keysym*
4	`U32`		*keycode*

The *keysym* and *down-flag* fields also take the same values as described for the KeyEvent message. Auto repeating behaviour of keys is also as described for the KeyEvent message.

The *keycode* is the XT keycode that produced the *keysym*. An XT keycode is an XT make scancode sequence encoded to fit in a single `U32` quantity. Single byte XT scancodes with a byte value less than 0x7f are encoded as is. 2-byte XT scancodes whose first byte is 0xe0 and second byte is less than 0x7f are encoded with the high bit of the first byte set. Some example mappings are

XT scancode	X11 keysym	RFB keycode	down-flag
0x1e	XK_A (0x41)	0x1e	1
0x9e	XK_A (0x41)	0x1e	0
0xe0 0x4d	XK_Right (0xff53)	0xcd	1
0xe0 0xcd	XK_Right (0xff53)	0xcd	0

7.4.12.2 QEMU Audio Client Message

This submessage allows the client to control how the audio data stream is received. There are three operations that can be invoked with this submessage, the payload varies according to which operation is requested.

The first operation enables audio capture from the server:

No. of bytes	Type	[Value]	Description
1	`U8`	255	*message-type*
1	`U8`	1	*submessage-type*
2	`U16`	0	*operation*

After invoking this operation, the client will receive a QEMU Audio Server Message when an audio stream begins.

The second operation is the inverse, to disable audio capture on the server:

No. of bytes	Type	[Value]	Description
1	`U8`	255	*message-type*
1	`U8`	1	*submessage-type*
2	`U16`	1	*operation*

Due to inherant race conditions in the protocol, after invoking this operation, the client may still receive further QEMU Audio Server Message messages for a short time.

The third and final operation is to set the audio sample format. This should be set before audio capture is enabled on the server, otherwise the client will not be able to reliably interpret the receiving audio buffers:

No. of bytes	Type	[Value]	Description
1	`U8`	255	*message-type*
1	`U8`	1	*submessage-type*
2	`U16`	2	*operation*
1	`U8`		*sample-format*
1	`U8`		*nchannels*
4	`U32`		*frequency*

The *sample-format* field must take one of the following values, and this describes the number of bytes that each sample will consume:

Value	No. of bytes	Type
0	1	`U8`
1	1	`S8`
2	2	`U16`
3	2	`S16`
4	4	`U32`
5	4	`S32`

The *nchannels* field must be either `1` (mono) or `2` (stereo).

7.5 Server to Client Messages

The server to client message types that all clients must support are:

Number	Name
0	FramebufferUpdate
1	SetColourMapEntries
2	Bell
3	ServerCutText

Optional message types are:

Number	Name
4	ResizeFrameBuffer
5	KeyFrameUpdate
6	Possibly used in UltraVNC
7	FileTransfer
8-10	Possibly used in UltraVNC
11	TextChat
12	Possibly used in UltraVNC
13	KeepAlive
14	Possibly used in UltraVNC
15	ResizeFrameBuffer
127	VMWare
128	Car Connectivity
150	EndOfContinuousUpdates
173	ServerState
248	ServerFence
249	OLIVE Call Control
250	xvp Server Message
252	tight
253	gii Server Message
254	VMWare
255	QEMU Server Message

The official, up-to-date list is maintained by IANA [1].

Note that before sending a message with an optional message type a server must have determined that the client supports the relevant extension by receiving some extension-specific confirmation from the client; usually a request for a given pseudo-encoding.

7.5.1 FramebufferUpdate

A framebuffer update consists of a sequence of rectangles of pixel data which the client should put into its framebuffer. It is sent in response to a *FramebufferUpdateRequest* from the client. Note that there may be an indefinite period between the *FramebufferUpdateRequest* and the *FramebufferUpdate*.

No. of bytes	Type	[Value]	Description
1	U8	0	*message-type*
1			*padding*
2	U16		*number-of-rectangles*

This is followed by *number-of-rectangles* rectangles of pixel data. Each rectangle consists of:

No. of bytes	Type	Description
2	`U16`	*x-position*
2	`U16`	*y-position*
2	`U16`	*width*
2	`U16`	*height*
4	`S32`	*encoding-type*

followed by the pixel data in the specified encoding. See Encodings for the format of the data for each encoding and Pseudo-encodings for the meaning of pseudo-encodings.

Note that a framebuffer update marks a transition from one valid framebuffer state to another. That means that a single update handles all received *FramebufferUpdateRequest* up to the point where the update is sent out.

However, because there is no strong connection between a *FramebufferUpdateRequest* and a subsequent *FramebufferUpdate*, a client that has more than one *FramebufferUpdateRequest* pending at any given time cannot be sure that it has received all framebuffer updates.

See the LastRect Pseudo-encoding for an extension to this message.

7.5.2 SetColourMapEntries

When the pixel format uses a "colour map", this message tells the client that the specified pixel values should be mapped to the given RGB intensities.

No. of bytes	Type	[Value]	Description
1	`U8`	1	*message-type*
1			*padding*
2	`U16`		*first-colour*
2	`U16`		*number-of-colours*

followed by *number-of-colours* repetitions of the following:

No. of bytes	Type	Description
2	`U16`	*red*
2	`U16`	*green*
2	`U16`	*blue*

7.5.3 Bell

Ring a bell on the client if it has one.

No. of bytes	Type	[Value]	Description
1	`U8`	2	*message-type*

7.5.4 ServerCutText

The server has new ISO 8859-1 (Latin-1) text in its cut buffer. Ends of lines are represented by the linefeed / newline character (value 10) alone. No carriage-return (value 13) is needed.

No. of bytes	Type	[Value]	Description
1	`U8`	3	*message-type*
3			*padding*
4	`U32`		*length*
length	`U8` array		*text*

See also Extended Clipboard Pseudo-Encoding which modifies the behaviour of this message.

7.5.5 EndOfContinuousUpdates

This message is sent whenever the server sees a EnableContinuousUpdates message with *enable* set to a non-zero value. It indicates that the server has stopped sending continuous updates and is now only reacting to FramebufferUpdateRequest messages.

No. of bytes	Type	[Value]	Description
1	`U8`	150	*message-type*

7.5.6 ServerFence

A server supporting the *Fence* extension sends this to request a synchronisation of the data stream.

No. of bytes	Type	[Value]	Description
1	`U8`	248	*message-type*
3			*padding*
4	`U32`		*flags*
1	`U8`		*length*
length	`U8` array		*payload*

The format and semantics is identical to ClientFence, but with the roles of the client and server reversed.

7.5.7 xvp Server Message

This has the following format:

No. of bytes	Type	[Value]	Description
1	`U8`	250	*message-type*
1			*padding*
1	`U8`	1	*xvp-extension-version*
1	`U8`		*xvp-message-code*

The possible values for *xvp-message-code* are: 0 - XVP_FAIL and 1 - XVP_INIT.

A server which supports the *xvp* extension declares this by sending a message with an XVP_INIT *xvp-message-code* when it receives a request from the client to use the xvp Pseudo-encoding. The server must specify in this message the highest *xvp-extension-version* it supports: the client may assume that the server supports all versions from 1 up to this value. The client is then free to use any supported version. Currently, only version 1 is defined.

A server which subsequently receives an xvp Client Message requesting an operation which it is unable to perform, informs the client of this by sending a message with an XVP_FAIL *xvp-message-code*, and the same *xvp-extension-version* as included in the client's operation request.

7.5.8 gii Server Message

This message is an extension and may only be sent if the server has previously received a SetEncodings message confirming that the client supports the General Input Interface extension via the gii Pseudo-encoding.

7.5.8.1 Version

The server response from a server with *gii* capabilities to a client declaring *gii* capabilities is a *gii* version message:

No. of bytes	Type	[Value]	Description
1	`U8`	253	*message-type*
1	`SUB_TYPE`	1 or 129	*endian-and-sub-type*
2	`EU16`	4	*length*
2	`EU16`	1	*maximum-version*
2	`EU16`	1	*minimum-version*

endian-and-sub-type is a bit-field with the leftmost bit indicating big endian if set, and little endian if cleared. The rest of the bits are the actual message sub type.

7.5.8.2 Device Creation Response

The server response to a *gii* Device Creation request from the client is the following response:

No. of bytes	Type	[Value]	Description
1	`U8`	253	*message-type*
1	`SUB_TYPE`	2 or 130	*endian-and-sub-type*
2	`EU16`	4	*length*
4	`EU32`		*device-origin*

endian-and-sub-type is a bit-field with the leftmost bit indicating big endian if set, and little endian if cleared. The rest of the bits are the actual message sub type.

device-origin is used as a handle to the device in subsequent communications. A *device-origin* of zero indicates device creation failure.

7.5.9 QEMU Server Message

This message may only be sent if the client has previously received a *FrameBufferUpdate* that confirms support for the intended *submessage-type*. Every `QEMU Server Message` begins with a standard header

No. of bytes	Type	[Value]	Description
1	`U8`	255	*message-type*
1	`U8`		*submessage-type*

This header is then followed by arbitrary data whose format is determined by the *submessage-type*. Possible values for *submessage-type* and their associated pseudo encodings are

Submessage Type	Pseudo Encoding	Description
1	-259	Audio

Submessage type 0 is unused, since the QEMU Extended Key Event Pseudo-encoding does not require any server messages.

7.5.9.1 QEMU Audio Server Message

This submessage allows the server to send an audio data stream to the client. There are three operations that can be invoked with this submessage, the payload varies according to which operation is requested.

The first operation informs the client that an audio stream is about to start

No. of bytes	Type	[Value]	Description
1	`U8`	255	*message-type*
1	`U8`	1	*submessage-type*
2	`U16`	1	*operation*

The second operation informs the client that an audio stream has now finished:

No. of bytes	Type	[Value]	Description
1	`U8`	255	*message-type*
1	`U8`	1	*submessage-type*
2	`U16`	0	*operation*

The third and final operation is to provide audio data.

No. of bytes	Type	[Value]	Description
1	`U8`	255	*message-type*
1	`U8`	1	*submessage-type*
2	`U16`	2	*operation*
4	`U32`		*data-length*
data-length	`U8` array		*data*

The *data-length* will be a multiple of (*sample-format* * *nchannels*) as requested by the client in an earlier QEMU Audio Client Message.

7.6 Encodings

The encodings defined in this document are:

Number	Name
0	Raw Encoding
1	CopyRect Encoding
2	RRE Encoding
4	CoRRE Encoding
5	Hextile Encoding
6	zlib Encoding
7	Tight Encoding
8	zlibhex Encoding
16	ZRLE Encoding
-23 to -32	JPEG Quality Level Pseudo-encoding
-223	DesktopSize Pseudo-encoding
-224	LastRect Pseudo-encoding
-239	Cursor Pseudo-encoding
-240	X Cursor Pseudo-encoding
-247 to -256	Compression Level Pseudo-encoding
-257	QEMU Pointer Motion Change Pseudo-encoding
-258	QEMU Extended Key Event Pseudo-encoding
-259	QEMU Audio Pseudo-encoding
-261	LED State Pseudo-encoding
-305	gii Pseudo-encoding
-307	DesktopName Pseudo-encoding
-308	ExtendedDesktopSize Pseudo-encoding
-309	xvp Pseudo-encoding
-312	Fence Pseudo-encoding
-313	ContinuousUpdates Pseudo-encoding
-412 to -512	JPEG Fine-Grained Quality Level Pseudo-encoding
-763 to -768	JPEG Subsampling Level Pseudo-encoding
0xc0a1e5ce	Extended Clipboard Pseudo-encoding

Other registered encodings are:

Number	Name
9	Ultra
10	Ultra2
15	TRLE
17	Hitachi ZYWRLE
20	H.264
21	JPEG
22	JRLE

Number	Name
1000 to 1002	Apple Inc.
1011	Apple Inc.
1024 to 1099	RealVNC
1100 to 1105	Apple Inc.
-1 to -22	Tight options
-33 to -218	Tight options
-219 to -222	Historical libVNCServer use
-225	PointerPos
-226 to -238	Tight options
-241 to -246	Tight options
-260	Tight PNG
-262 to -272	QEMU
-273 to -304	VMWare
-306	popa
-310	OLIVE Call Control
-311	ClientRedirect
-314	CursorWithAlpha
-523 to -528	Car Connectivity
0x48323634	VA H.264
0x574d5600 to 0x574d56ff	VMWare
0xc0a1e5cf	PluginStreaming
0xfffe0000	KeyboardLedState
0xfffe0001	SupportedMessages
0xfffe0002	SupportedEncodings
0xfffe0003	ServerIdentity
0xfffe0004 to 0xfffe00ff	libVNSServer
0xffff0000	Cache
0xffff0001	CacheEnable
0xffff0002	XOR zlib
0xffff0003	XORMonoRect zlib
0xffff0004	XORMultiColor zlib
0xffff0005	SolidColor
0xffff0006	XOREnable
0xffff0007	CacheZip
0xffff0008	SolMonoZip
0xffff0009	UltraZip
0xffff8000	ServerState
0xffff8001	EnableKeepAlive

Number	Name
0xffff8002	FTProtocolVersion
0xffff8003	Session

The official, up-to-date list is maintained by IANA [1].

7.6.1 Raw Encoding

The simplest encoding type is raw pixel data. In this case the data consists of *width* * *height* pixel values (where *width* and *height* are the width and height of the rectangle). The values simply represent each pixel in left-to-right scanline order. All RFB clients must be able to cope with pixel data in this raw encoding, and RFB servers should only produce raw encoding unless the client specifically asks for some other encoding type.

No. of bytes	Type	Description
width * *height* * *bytesPerPixel*	`PIXEL` array	*pixels*

7.6.2 CopyRect Encoding

The *CopyRect* (copy rectangle) encoding is a very simple and efficient encoding which can be used when the client already has the same pixel data elsewhere in its framebuffer. The encoding on the wire simply consists of an X,Y coordinate. This gives a position in the framebuffer from which the client can copy the rectangle of pixel data. This can be used in a variety of situations, the most obvious of which are when the user moves a window across the screen, and when the contents of a window are scrolled. A less obvious use is for optimising drawing of text or other repeating patterns. An intelligent server may be able to send a pattern explicitly only once, and knowing the previous position of the pattern in the framebuffer, send subsequent occurrences of the same pattern using the *CopyRect* encoding.

No. of bytes	Type	Description
2	`U16`	*src-x-position*
2	`U16`	*src-y-position*

7.6.3 RRE Encoding

RRE stands for *rise-and-run-length encoding* and as its name implies, it is essentially a two-dimensional analogue of run-length encoding. RRE-encoded rectangles arrive at the client in a form which can be rendered immediately and efficiently by the simplest of graphics engines. RRE is not appropriate for complex desktops, but can be useful in some situations.

The basic idea behind RRE is the partitioning of a rectangle of pixel data into rectangular subregions (subrectangles) each of which consists of pixels of a single value and the union of which comprises the original rectangular region. The near-optimal partition of a given rectangle into such subrectangles is relatively easy to compute.

The encoding consists of a background pixel value, *Vb* (typically the most prevalent pixel value in the rectangle) and a count *N*, followed by a list of *N* subrectangles, each of which consists of a tuple <*v*, *x*, *y*, *w*, *h*> where *v* (!= *Vb*) is the pixel value, (*x*, *y*) are the coordinates of the subrectangle relative to the top-left corner of the rectangle, and (*w*, *h*) are the width and height of the subrectangle. The client can render the original rectangle by drawing a filled rectangle of the background pixel value and then drawing a filled rectangle corresponding to each subrectangle.

On the wire, the data begins with the header:

No. of bytes	Type	Description
4	`U32`	*number-of-subrectangles*
bytesPerPixel	`PIXEL`	*background-pixel-value*

This is followed by *number-of-subrectangles* instances of the following structure:

No. of bytes	Type	Description
bytesPerPixel	`PIXEL`	*subrect-pixel-value*
2	`U16`	*x-position*
2	`U16`	*y-position*
2	`U16`	*width*
2	`U16`	*height*

7.6.4 CoRRE Encoding

CoRRE stands for *compressed rise-and-run-length encoding* and as its name implies, it is a variant of the above RRE Encoding and as such essentially a two-dimensional analogue of run-length encoding.

The only difference between CoRRE and RRE is that the position, width and height of the subrectangles are limited to a maximum of 255 pixels. Because of this, the server needs to produce several rectangles in order to cover a larger area. The Hextile Encoding is probably a better choice in the majority of cases.

On the wire, the data begins with the header:

No. of bytes	Type	Description
4	`U32`	*number-of-subrectangles*
bytesPerPixel	`PIXEL`	*background-pixel-value*

This is followed by *number-of-subrectangles* instances of the following structure:

No. of bytes	Type	Description
bytesPerPixel	`PIXEL`	*subrect-pixel-value*
1	`U8`	*x-position*
1	`U8`	*y-position*
1	`U8`	*width*
1	`U8`	*height*

7.6.5 Hextile Encoding

Hextile is a variation on the RRE idea. Rectangles are split up into 16x16 tiles, allowing the dimensions of the subrectangles to be specified in 4 bits each, 16 bits in total. The rectangle is split into tiles starting at the top left going in left-to-right, top-to-bottom order. The encoded contents of the tiles simply follow one another in the predetermined order. If the width of the whole rectangle is not an exact multiple of 16 then the width of the last tile in each row will be correspondingly smaller. Similarly if the height of the whole rectangle is not an exact multiple of 16 then the height of each tile in the final row will also be smaller.

Each tile is either encoded as raw pixel data, or as a variation on RRE. Each tile has a background pixel value, as before. The background pixel value does not need to be explicitly specified for a given tile if it is the same as the background of the previous tile. However the background pixel value may not be carried over if the previous tile was raw. If all of the subrectangles of a tile have the same pixel value, this can be specified once as a foreground pixel value for the whole tile. As with the background, the foreground pixel value can be left unspecified, meaning it is carried over from the previous tile. The foreground pixel value may not be carried over if the previous tile was raw or had the SubrectsColored bit set. It may, however, be carried over from a previous tile with the AnySubrects bit clear, as long as that tile itself carried over a valid foreground from its previous tile.

So the data consists of each tile encoded in order. Each tile begins with a subencoding type byte, which is a mask made up of a number of bits:

No. of bytes	Type	[Value]	Description
1	`U8`		*subencoding-mask*:
		1	**Raw**
		2	**BackgroundSpecified**
		4	**ForegroundSpecified**
		8	**AnySubrects**
		16	**SubrectsColoured**

If the **Raw** bit is set then the other bits are irrelevant; *width* * *height* pixel values follow (where *width* and *height* are the width and height of the tile). Otherwise the other bits in the mask are as follows:

BackgroundSpecified
If set, a pixel value follows which specifies the background colour for this tile:

No. of bytes	Type	Description
bytesPerPixel	`PIXEL`	*background-pixel-value*

The first non-raw tile in a rectangle must have this bit set. If this bit isn't set then the background is the same as the last tile.

ForegroundSpecified
If set, a pixel value follows which specifies the foreground colour to be used for all subrectangles in this tile:

No. of bytes	Type	Description
bytesPerPixel	`PIXEL`	*foreground-pixel-value*

If this bit is set then the **SubrectsColoured** bit must be zero.

AnySubrects
If set, a single byte follows giving the number of subrectangles following:

No. of bytes	Type	Description
1	`U8`	*number-of-subrectangles*

If not set, there are no subrectangles (i.e. the whole tile is just solid background colour).

SubrectsColoured
If set then each subrectangle is preceded by a pixel value giving the colour of that subrectangle, so a subrectangle is:

No. of bytes	Type	Description
bytesPerPixel	`PIXEL`	*subrect-pixel-value*
1	`U8`	*x-and-y-position*
1	`U8`	*width-and-height*

If not set, all subrectangles are the same colour, the foreground colour; if the **ForegroundSpecified** bit wasn't set then the foreground is the same as the last tile. A subrectangle is:

No. of bytes	Type	Description
1	`U8`	*x-and-y-position*
1	`U8`	*width-and-height*

The position and size of each subrectangle is specified in two bytes, *x-and-y-position* and *width-and-height*. The most-significant four bits of *x-and-y-position* specify the X position, the least-significant specify the Y position. The most-significant four bits of *width-and-height* specify the width minus one, the least-significant specify the height minus one.

7.6.6 zlib Encoding

The zlib encoding uses zlib [3] to compress rectangles encoded according to the Raw Encoding. A single zlib "stream" object is used for a given RFB connection, so that zlib rectangles must be encoded and decoded strictly in order.

[3]	*(1, 2, 3, 4)* see http://www.gzip.org/zlib/

No. of bytes	Type	Description
4	`U32`	*length*
length	`U8` array	*zlibData*

The *zlibData*, when uncompressed, represents a rectangle according to the Raw Encoding.

7.6.7 Tight Encoding

Tight encoding provides efficient compression for pixel data. To reduce implementation complexity, the width of any Tight-encoded rectangle cannot exceed 2048 pixels. If a wider rectangle is desired, it must be split into several rectangles and each one should be encoded separately.

The first byte of each Tight-encoded rectangle is a *compression-control* byte:

No. of bytes	Type	Description
1	`U8`	*compression-control*

The least significant four bits of the *compression-control* byte inform the client which zlib compression streams should be reset before decoding the rectangle. Each bit is independent and corresponds to a separate zlib stream that should be reset:

Bit	Description
0	Reset stream 0
1	Reset stream 1
2	Reset stream 2
3	Reset stream 3

One of three possible compression methods are supported in the Tight encoding. These are **BasicCompression**, **FillCompression** and **JpegCompression**. If the bit 7 (the most significant bit) of the *compression-control* byte is 0, then the compression type is **BasicCompression**. In that case, bits 7-4 (the most significant four bits) of *compression-control* should be interpreted as follows:

Bits	Binary value	Description
5-4	00	Use stream 0
	01	Use stream 1
	10	Use stream 2
	11	Use stream 3
6	0	---
	1	*read-filter-id*
7	0	**BasicCompression**

Otherwise, if the bit 7 of *compression-control* is set to 1, then the compression method is either **FillCompression** or **JpegCompression**, depending on other bits of the same byte:

Bits	Binary value	Description
7-4	1000	**FillCompression**
	1001	**JpegCompression**
	any other	Invalid

Note: **JpegCompression** may only be used when *bits-per-pixel* is either 16 or 32 and the client has advertized a quality level using the JPEG Quality Level Pseudo-encoding.

The Tight encoding makes use of a new type `TPIXEL` (Tight pixel). This is the same as a `PIXEL` for the agreed pixel format, except where *true-colour-flag* is non-zero, *bits-per-pixel* is 32, *depth* is 24 and all of the bits making up the red, green and blue intensities are exactly 8 bits wide. In this case a `TPIXEL` is only 3 bytes long, where the first byte is the red component, the second byte is the green component, and the third byte is the blue component of the pixel color value.

The data following the *compression-control* byte depends on the compression method.

FillCompression

If the compression type is **FillCompression**, then the only pixel value follows, in `TPIXEL` format. This value applies to all pixels of the rectangle.

JpegCompression

If the compression type is **JpegCompression**, the following data stream looks like this:

No. of bytes	Type	Description
1-3		*length* in compact representation
length	`U8` array	*jpeg-data*

length is compactly represented in one, two or three bytes, according to the following scheme:

Value	Description
0xxxxxxx	for values 0..127
1xxxxxxx 0yyyyyyy	for values 128..16383
1xxxxxxx 1yyyyyyy zzzzzzzz	for values 16384..4194303

Here each character denotes one bit, xxxxxxx are the least significant 7 bits of the value (bits 0-6), yyyyyyy are bits 7-13, and zzzzzzzz are the most significant 8 bits (bits 14-21). For example, decimal value 10000 should be represented as two bytes: binary 10010000 01001110, or hexadecimal 90 4E.

The *jpeg-data* is a JFIF stream.

BasicCompression

If the compression type is **BasicCompression** and bit 6 (the *read-filter-id* bit) of the *compression-control* byte was set to 1, then the next (second) byte specifies *filter-id* which tells the decoder what filter type was used by the encoder to pre-process pixel data before the compression. The *filter-id* byte can be one of the following:

No. of bytes	Type	[Value]	Description
1	`U8`		*filter-id*
		0	**CopyFilter** (no filter)
		1	**PaletteFilter**
		2	**GradientFilter**

If bit 6 of the *compression-control* byte is set to 0 (no *filter-id* byte), then the **CopyFilter** is used.

CopyFilter

When the **CopyFilter** is active, raw pixel values in `TPIXEL` format will be compressed. See below for details on the compression.

PaletteFilter
The **PaletteFilter** converts true-color pixel data to indexed colors and a palette which can consist of 2..256 colors. If the number of colors is 2, then each pixel is encoded in 1 bit, otherwise 8 bits are used to encode one pixel. 1-bit encoding is performed such way that the most significant bits correspond to the leftmost pixels, and each row of pixels is aligned to the byte boundary. When the **PaletteFilter** is used, the palette is sent before the pixel data. The palette begins with an unsigned byte which value is the number of colors in the palette minus 1 (i.e. 1 means 2 colors, 255 means 256 colors in the palette). Then follows the palette itself which consist of pixel values in `TPIXEL` format.

GradientFilter
The **GradientFilter** pre-processes pixel data with a simple algorithm which converts each color component to a difference between a "predicted" intensity and the actual intensity. Such a technique does not affect uncompressed data size, but helps to compress photo-like images better. Pseudo-code for converting intensities to differences follows:

```
P[i,j] := V[i-1,j] + V[i,j-1] - V[i-1,j-1];
if (P[i,j] < 0) then P[i,j] := 0;
if (P[i,j] > MAX) then P[i,j] := MAX;
D[i,j] := V[i,j] - P[i,j];
```

Here `V[i,j]` is the intensity of a color component for a pixel at coordinates `(i,j)`. For pixels outside the current rectangle, `V[i,j]` is assumed to be zero (which is relevant for `P[i,0]` and `P[0,j]`). MAX is the maximum intensity value for a color component.

Note: The **GradientFilter** may only be used when *bits-per-pixel* is either 16 or 32.

After the pixel data has been filtered with one of the above three filters, it is compressed using the zlib library. But if the data size after applying the filter but before the compression is less then 12, then the data is sent as is, uncompressed. Four separate zlib streams (0..3) can be used and the decoder should read the actual stream id from the *compression-control* byte (see [NOTE1]).

If the compression is not used, then the pixel data is sent as is, otherwise the data stream looks like this:

No. of bytes	Type	Description
1-3		*length* in compact representation
length	`U8` array	*zlibData*

length is compactly represented in one, two or three bytes, just like in the **JpegCompression** method (see above).

[NOTE1]	The decoder must reset the zlib streams before decoding the rectangle, if some of the bits 0, 1, 2 and 3 in the *compression-control* byte are set to 1. Note that the decoder must reset the indicated zlib streams even if the compression type is **FillCompression** or **JpegCompression**.

7.6.8 zlibhex Encoding

The zlibhex encoding uses zlib [3] to optionally compress subrectangles according to the Hextile Encoding. Refer to the hextile encoding for information on how the rectangle is divided into subrectangles and other basic properties of subrectangles. One zlib "stream" object is used for subrectangles encoded according to the **Raw** subencoding and one zlib "stream" object is used for all other subrectangles.

The hextile subencoding bitfield is extended with these bits:

No. of bytes	Type	[Value]	Description
1	`U8`		*subencoding-mask*:
		32	**ZlibRaw**
		64	**Zlib**

If either of the **ZlibRaw** or the **Zlib** bit is set, the subrectangle is compressed using zlib, like this:

No. of bytes	Type	Description
2	`U16`	*length*
length	`U8` array	*zlibData*

Like the **Raw** bit in hextile, the **ZlibRaw** bit in zlibhex cancels all other bits and the subrectangle is encoded using the first zlib "stream" object. The *zlibData*, when uncompressed, should in this case be interpreted as the **Raw** data in the hextile encoding.

If the **Zlib** bit is set, the rectangle is encoded using the second zlib "stream" object. The *zlibData*, when uncompressed, represents a plain hextile rectangle according to the lower 5 bits in the subencoding.

If neither the **ZlibRaw** nor the **Zlib** bit is set, the subrectangle follows the rules described in the Hextile Encoding.

7.6.9 ZRLE Encoding

ZRLE stands for Zlib [3] Run-Length Encoding, and combines zlib compression, tiling, palettisation and run-length encoding. On the wire, the rectangle begins with a 4-byte length field, and is followed by that many bytes of zlib-compressed data. A single zlib "stream" object is used for a given RFB protocol connection, so that ZRLE rectangles must be encoded and decoded strictly in order.

No. of bytes	Type	Description
4	`U32`	*length*
length	`U8` array	*zlibData*

The *zlibData* when uncompressed represents tiles of 64x64 pixels in left-to-right, top-to-bottom order, similar to hextile. If the width of the rectangle is not an exact multiple of 64 then the width of the last tile in each row is smaller, and if the height of the rectangle is not an exact multiple of 64 then the height of each tile in the final row is smaller.

ZRLE makes use of a new type `CPIXEL` (compressed pixel). This is the same as a `PIXEL` for the agreed pixel format, except where *true-colour-flag* is non-zero, *bits-per-pixel* is 32, *depth* is 24 or less and all of the bits making up the red, green and blue intensities fit in either the least significant 3 bytes or the most significant 3 bytes. In this case a `CPIXEL` is only 3 bytes long, and contains the least significant or the most significant 3 bytes as appropriate. *bytesPerCPixel* is the number of bytes in a `CPIXEL`.

Note that for the corner case where *bits-per-pixel* is 32 and *depth* is 16 or less (this is a corner case, since the client is **much** better off using 16 or even 8 *bits-per-pixels*) a `CPIXEL` is still 3 bytes long. By convention, the three least significant bytes are used when both the three least and the three most significant bytes would cover the used bits.

Each tile begins with a *subencoding* type byte. The top bit of this byte is set if the tile has been run-length encoded, clear otherwise. The bottom seven bits indicate the size of the palette used: zero means no palette, one means that the tile is of a single colour, 2 to 127 indicate a palette of that size. The possible values of *subencoding* are:

0

Raw pixel data. *width* * *height* pixel values follow (where width and height are the width and height of the tile):

No. of bytes	Type	Description
*width * height * bytesPerCPixel*	`CPIXEL` array	*pixels*

1

A solid tile consisting of a single colour. The pixel value follows:

No. of bytes	Type	Description
bytesPerCPixel	`CPIXEL`	*pixelValue*

2 to 16

Packed palette types. Followed by the palette, consisting of *paletteSize* (=*subencoding*) pixel values. Then the packed pixels follow, each pixel represented as a bit field yielding an index into the palette (0 meaning the first palette entry). For *paletteSize* 2, a 1-bit field is used, for *paletteSize* 3 or 4 a 2-bit field is used and for *paletteSize* from 5 to 16 a 4-bit field is used. The bit fields are packed into bytes, the most significant bits representing the leftmost pixel (i.e. big endian). For tiles not a multiple of 8, 4 or 2 pixels wide (as appropriate), padding bits are used to align each row to an exact number of bytes.

No. of bytes	Type	Description
paletteSize * *bytesPerCPixel*	`CPIXEL` array	*palette*
m	`U8` array	*packedPixels*

where *m* is the number of bytes representing the packed pixels. For *paletteSize* of 2 this is floor((*width* + 7) / 8) * *height*, for *paletteSize* of 3 or 4 this is floor((*width* + 3) / 4) * *height*, for *paletteSize* of 5 to 16 this is floor((*width* + 1) / 2) * *height*.

17 to 127
Unused (no advantage over palette RLE).

128
Plain RLE. Consists of a number of runs, repeated until the tile is done. Runs may continue from the end of one row to the beginning of the next. Each run is a represented by a single pixel value followed by the length of the run. The length is represented as one or more bytes. The length is calculated as one more than the sum of all the bytes representing the length. Any byte value other than 255 indicates the final byte. So for example length 1 is represented as [0], 255 as [254], 256 as [255,0], 257 as [255,1], 510 as [255,254], 511 as [255,255,0] and so on.

No. of bytes	Type	[Value]	Description
bytesPerCPixel	`CPIXEL`		*pixelValue*
r	`U8` array	255	
1	`U8`		(*runLength* - 1) % 255

Where *r* is floor((*runLength* - 1) / 255).

129
Unused.

130 to 255
Palette RLE. Followed by the palette, consisting of *paletteSize* = (*subencoding* - 128) pixel values:

No. of bytes	Type	Description
paletteSize * *bytesPerCPixel*	`CPIXEL` array	*palette*

Then as with plain RLE, consists of a number of runs, repeated until the tile is done. A run of length one is represented simply by a palette index:

No. of bytes	Type	Description
1	`U8`	*paletteIndex*

A run of length more than one is represented by a palette index with the top bit set, followed by the length of the run as for plain RLE.

No. of bytes	Type	[Value]	Description
1	`U8`		*paletteIndex* + 128
r	`U8` array	255	
1	`U8`		(*runLength* - 1) % 255

Where *r* is floor((*runLength* - 1) / 255).

7.7 Pseudo-encodings

7.7.1 JPEG Quality Level Pseudo-encoding

Specifies the desired quality from the JPEG encoder. Encoding number -23 implies high JPEG quality and -32 implies low JPEG quality. Low quality can be useful in low bandwidth situations. If the JPEG quality level is not specified, **JpegCompression** is not used in the Tight Encoding.

The quality level concerns lossy compression and hence the setting is a tradeoff between image quality and bandwidth. The specification defines neither what bandwidth is required at a certain quality level nor what image quality you can expect. The quality level is also just a hint to the server.

7.7.2 Cursor Pseudo-encoding

A client which requests the *Cursor* pseudo-encoding is declaring that it is capable of drawing a mouse cursor locally. This can significantly improve perceived performance over slow links. The server sets the cursor shape by sending a pseudo-rectangle with the *Cursor* pseudo-encoding as part of an update. The pseudo-rectangle's *x-position* and *y-position* indicate the hotspot of the cursor, and *width* and *height* indicate the width and height of the cursor in pixels. The data consists of *width* * *height* pixel values followed by a bitmask. The bitmask consists of left-to-right, top-to-bottom scanlines, where each scanline is padded to a whole number of bytes floor((*width* + 7) / 8). Within each byte the most significant bit represents the leftmost pixel, with a 1-bit meaning the corresponding pixel in the cursor is valid.

No. of bytes	Type	Description
width * *height* * *bytesPerPixel*	`PIXEL` array	*cursor-pixels*
floor((*width* + 7) / 8) * *height*	`U8` array	*bitmask*

7.7.3 X Cursor Pseudo-encoding

A client which requests the *X Cursor* pseudo-encoding is declaring that it is capable of drawing a mouse cursor locally. This can significantly improve perceived performance over slow links. The server sets the cursor shape by sending a pseudo-rectangle with the *X Cursor* pseudo-encoding as part of an update.

The pseudo-rectangle's *x-position* and *y-position* indicate the hotspot of the cursor, and *width* and *height* indicate the width and height of the cursor in pixels.

The data consists of the primary and secondary colours for the cursor, followed by one bitmap for the colour and one bitmask for the transparency. The bitmap and bitmask both consist of left-to-right, top-to-bottom scanlines, where each scanline is padded to a whole number of bytes floor((*width* + 7) / 8). Within each byte the most significant bit represents the leftmost pixel, with a 1-bit meaning the corresponding pixel should use the primary colour, or that the pixel is valid.

No. of bytes	Type	Description
1	`U8`	*primary-r*
1	`U8`	*primary-g*
1	`U8`	*primary-b*
1	`U8`	*secondary-r*
1	`U8`	*secondary-g*
1	`U8`	*secondary-b*
floor((*width* + 7) / 8) * *height*	`U8` array	*bitmap*
floor((*width* + 7) / 8) * *height*	`U8` array	*bitmask*

7.7.4 DesktopSize Pseudo-encoding

A client which requests the *DesktopSize* pseudo-encoding is declaring that it is capable of coping with a change in the framebuffer width and/or height.

The server changes the desktop size by sending a pseudo-rectangle with the *DesktopSize* pseudo-encoding. The pseudo-rectangle's *x-position* and *y-position* are ignored, and *width* and *height* indicate the new width and height of the framebuffer. There is no further data associated with the pseudo-rectangle.

The semantics of the *DesktopSize* pseudo-encoding were originally not clearly defined and as a results there are multiple differing implementations in the wild. Both the client and server need to take special steps to ensure maximum compatibility.

In the initial implementation the *DesktopSize* pseudo-rectangle was sent in its own update without any modifications to the framebuffer data. The client would discard the framebuffer contents upon receiving this pseudo-rectangle and the server would consider the entire framebuffer to be modified.

A later implementation sent the *DesktopSize* pseudo-rectangle together with modifications to the framebuffer data. It also expected the client to retain the framebuffer contents as those modifications could be from after the framebuffer resize had occurred on the server.

The semantics defined here retain compatibility with both of two older implementations.

7.7.4.1 Server Semantics

The update containing the pseudo-rectangle should not contain any rectangles that change the framebuffer data as that will most likely be discarded by the client and will have to be resent later.

The server should assume that the client discards the framebuffer data when receiving a *DesktopSize* pseudo-rectangle. It should therefore not use any encoding that relies on the previous contents of the framebuffer. The server should also consider the entire framebuffer to be modified.

Some early client implementations require the *DesktopSize* pseudo-rectangle to be the very last rectangle in an update. Servers should make every effort to support these.

The server should only send a *DesktopSize* pseudo-rectangle when an actual change of the framebuffer dimensions has occurred. Some clients respond to a *DesktopSize* pseudo-rectangle in a way that could send the system into an infinite loop if the server sent out the pseudo-rectangle for anything other than an actual change.

7.7.4.2 Client Semantics

The client should assume that the server expects the framebuffer data to be retained when the framebuffer dimensions change. This requirement can be satisfied either by actually retaining the framebuffer data, or by making sure that *incremental* is set to zero (false) in the next *FramebufferUpdateRequest*.

The principle of one framebuffer update being a transition from one valid state to another does not hold for updates with the *DesktopSize* pseudo-rectangle as the framebuffer contents can temporarily be partially or completely undefined. Clients should try to handle this gracefully, e.g. by showing a black framebuffer or delay the screen update until a proper update of the framebuffer contents has been received.

7.7.5 LastRect Pseudo-encoding

A client which requests the *LastRect* pseudo-encoding is declaring that it does not need the exact number of rectangles in a *FramebufferUpdate* message. Instead, it will stop parsing when it reaches a *LastRect* rectangle. A server may thus start transmitting the *FramebufferUpdate* message before it knows exactly how many rectangles it is going to transmit, and the server typically advertises this situation by saying that it is going to send 65535 rectangles, but it then stops with a *LastRect* instead of sending all of them. There is no further data associated with the pseudo-rectangle.

7.7.6 Compression Level Pseudo-encoding

Specifies the desired compression level. Encoding number -247 implies high compression level, -255 implies low compression level. Low compression level can be useful to get low latency in medium to high bandwidth situations and high compression level can be useful in low bandwidth situations.

The compression level concerns lossless compression, and hence the setting is a tradoff between CPU time and bandwidth. It is therefore probably difficult to define exact cut-off points for which compression levels should be used for any given bandwidth. The compression level is just a hint for the server, and there is no specification for what a specific compression level means.

7.7.7 QEMU Pointer Motion Change Pseudo-encoding

A client that supports this encoding declares that is able to send pointer motion events either as absolute coordinates, or relative deltas. The server can switch between different pointer motion modes by sending a rectangle with this encoding. If the *x-position* in the update is 1, the server is requesting absolute coordinates, which is the RFB protocol default when this encoding is not supported. If the *x-position* in the update is 0, the server is requesting relative deltas.

When relative delta mode is active, the semantics of the PointerEvent message are changed. The *x-position* and *y-position* fields are to be treated as `S16` quantities, denoting the delta from the last position. A client can compute the signed deltas with the logic:

```
uint16 dx = x + 0x7FFF - last_x
uint16 dy = y + 0x7FFF - last_y
```

If the client needs to send an updated *button-mask* without any associated motion, it should use the value 0x7FFF in the *x-position* and *y-position* fields of the PointerEvent

Servers are advised to implement this pseudo-encoding if the virtual desktop is associated a input device that expects relative coordinates, for example, a virtual machine with a PS/2 mouse. Prior to this extension, a server with such a input device would have to perform the absolute to relative delta conversion itself. This can result in the client pointer hitting an "invisible wall".

Clients are advised that when generating events in relative pointer mode, they should grab and hide the local pointer. When the local pointer hits any edge of the client window, it should be warped back by 100 pixels. This ensures that continued movement of the user's input device will continue to generate relative deltas and thus avoid the "invisible wall" problem.

7.7.8 QEMU Extended Key Event Pseudo-encoding

A client that supports this encoding is indicating that it is able to provide raw keycodes as an alternative to keysyms. If a server wishes to receive raw keycodes it will send an empty pseudo-rectangle with the matching pseudo-encoding. After receiving this notification, clients may optionally use the QEMU Extended Key Event Message to send key events, in preference to the traditional KeyEvent message.

7.7.9 QEMU Audio Pseudo-encoding

A client that supports this encoding is indicating that it is able to receive an audio data stream. If a server wishes to send audio data it will send an empty pseudo-rectangle with the matching pseudo-encoding. After receiving this notification, clients may optionally use the QEMU Audio Client Message.

7.7.10 LED State Pseudo-encoding

A client that supports this encoding is indicating that it can toggle the state of lock keys on the local keyboard. The server will send a pseudo-rectangle with the following contents when it wishes to update the client's state:

No. of bytes	Type	Description
1	`U8`	*state*

The bits of *state* are defined as:

Bit	Description
0	Scroll Lock
1	Num Lock
2	Caps Lock

The remaining bits are reserved and must be ignored.

An update must be sent whenever the server state changes, but may also be sent at other times to compensate for variance in behaviour between the server and client keyboard handling.

7.7.11 gii Pseudo-encoding

A client that supports the General Input Interface extension starts by requesting the *gii* pseudo-encoding declaring that it is capable of accepting the gii Server Message. The server, in turn, declares that it is capable of accepting the gii Client Message by sending a gii Server Message of subtype *version*.

Requesting the *gii* pseudo-encoding is the first step when a client wants to use the *gii* extension of the RFB protocol. The *gii* extension is used to provide a more powerful input protocol for cases where the standard input model is insufficient. It supports relative mouse movements, mouses with more than 8 buttons and mouses with more than three axes. It even supports joysticks and gamepads.

7.7.12 DesktopName Pseudo-encoding

A client which requests the DesktopName pseudo-encoding is declaring that it is capable of coping with a change of the desktop name. The server changes the desktop name by sending a pseudo-rectangle with the DesktopName pseudo-encoding in an update. The pseudo-rectangle's x-position, y-position, width, and height must be zero. After the rectangle header, a string with the new name follows.

No. of bytes	Type	Description
4	`U32`	*name-length*
name-length	`U8` array	*name-string*

The text encoding used for *name-string* is UTF-8.

7.7.13 ExtendedDesktopSize Pseudo-encoding

A client which requests the *ExtendedDesktopSize* pseudo-encoding is declaring that it is capable of coping with a change in the framebuffer width, height, and/or screen configuration. This encoding is used in conjunction with the *SetDesktopSize* message. If a server supports the *ExtendedDesktopSize* encoding, it must also have basic support for the *SetDesktopSize* message although it may deny all requests to change the screen layout.

The *ExtendedDesktopSize* pseudo-encoding is designed to replace the simpler *DesktopSize* one. Servers and clients should support both for maximum compatibility, but a server must only send the extended version to a client asking for both. The semantics of *DesktopSize* are not as well-defined as for *ExtendedDesktopSize* and handling both at the same time would require needless complexity in the client.

The server must send an *ExtendedDesktopSize* rectangle in response to a *FramebufferUpdateRequest* with *incremental* set to zero, assuming the client has requested the *ExtendedDesktopSize* pseudo-encoding using the *SetEncodings* message. This requirement is needed so that the client has a reliable way of fetching the initial screen configuration, and to determine if the server supports the *SetDesktopSize* message.

A consequence of this is that a client must not respond to an *ExtendedDesktopSize* rectangle by sending a *FramebufferUpdateRequest* with *incremental* set to zero. Doing so would make the system go into an infinite loop.

The server must also send an *ExtendedDesktopSize* rectangle in response to a *SetDesktopSize* message, indicating the result.

For a full description of server behaviour as a result of the *SetDesktopSize* message, see SetDesktopSize.

Rectangles sent as a result of a *SetDesktopSize* message must be sent as soon as possible. Rectangles sent for other reasons may be subjected to delays imposed by the server.

An update containing an *ExtendedDesktopSize* rectangle must not contain any changes to the framebuffer data, neither before nor after the *ExtendedDesktopSize* rectangle.

The pseudo-rectangle's *x-position* indicates the reason for the change:

0

The screen layout was changed via non-RFB means on the server. For example the server may have provided means for server-side applications to manipulate the screen layout. This code is also used when the client sends a non-incremental *FrameBufferUpdateRequest* to learn the server's current state.

1

The client receiving this message requested a change of the screen layout. The change may or may not have happened depending on server policy or available resources. The status code in the *y-position* field must be used to determine which.

2

Another client requested a change of the screen layout and the server approved it. A rectangle with this code is never sent if the server denied the request.

More reasons may be added in the future. Clients should treat an unknown value as a server-side change (i.e. as if *x-position* was set to zero).

The pseudo-rectangle's *y-position* indicates the status code for a change requested by a client:

Code	Description
0	No error
1	Resize is administratively prohibited
2	Out of resources
3	Invalid screen layout

This field shall be set to zero by the server and ignored by clients when not defined. Other error codes may be added in the future and clients must treat them as an unknown failure.

The *width* and *height* indicates the new width and height of the framebuffer.

The encoding data is defined as:

No. of bytes	Type	Description
1	`U8`	*number-of-screens*
3		*padding*
number-of-screens * 16	`SCREEN` array	*screens*

The *number-of-screens* field indicates the number of active screens and allows for multi head configurations. It also indicates how many `SCREEN` structures that follows. These are defined as:

No. of bytes	Type	Description
4	`U32`	*id*
2	`U16`	*x-position*
2	`U16`	*y-position*
2	`U16`	*width*
2	`U16`	*height*
4	`U32`	*flags*

The *id* field contains an arbitrary value that the server and client can use to map RFB screens to physical screens. The value must be unique in the current set of screens and must be preserved for the lifetime of that RFB screen. New ids are assigned by whichever side creates the screen. An *id* may be reused if there has been a subsequent update of the screen layout where the *id* was not used.

The *flags* field is currently unused. Clients and servers must ignore, but preserve, any bits it does not understand. For new screens, those bits must be set to zero.

Note that a simple client which does not support multi head does not need to parse the list of screens and can simply display the entire framebuffer.

7.7.14 xvp Pseudo-encoding

A client which requests the *xvp* pseudo-encoding is declaring that it wishes to use the *xvp* extension. If the server supports this, it replies with a message of type xvp Server Message, using an *xvp-message-code* of *XVP_INIT*. This informs the client that it may then subsequently send messages of type xvp Client Message.

7.7.15 Fence Pseudo-encoding

A client which requests the *Fence* pseudo-encoding is declaring that it supports and/or wishes to use the *Fence* extension. The server should send a ServerFence the first time it sees a `SetEncodings` message with the *Fence* pseudo-encoding, in order to inform the client that this extension is supported. The message can use any flags or payload.

7.7.16 ContinuousUpdates Pseudo-encoding

A client which requests the *ContinuousUpdates* pseudo-encoding is declaring that it wishes to use the EnableContinuousUpdates extension. The server must send a EndOfContinuousUpdates message the first time it sees a `SetEncodings` message with the *ContinuousUpdates* pseudo-encoding, in order to inform the client that the extension is supported.

7.7.17 JPEG Fine-Grained Quality Level Pseudo-encoding

The JPEG Fine-Grained Quality Level pseudo-encoding allows the image quality to be specified on a 0 to 100 scale, with -512 corresponding to image quality 0 and -412 corresponding to image quality 100. This pseudo-encoding was originally intended for use with JPEG-encoded subrectangles, but it could be used with other types of image encoding as well.

7.7.18 JPEG Subsampling Level Pseudo-Encoding

The JPEG Subsampling Level pseudo-encoding allows the level of chrominance subsampling to be specified. When a JPEG image is encoded, the RGB pixels are first converted to YCbCr, a colorspace in which brightness (luminance) is separated from color (chrominance.) Since the human eye is more sensitive to spatial changes in brightness than to spatial changes in color, the chrominance components (Cb, Cr) can be subsampled to save bandwidth without losing much image quality (on smooth images, such as photographs, chrominance subsampling is often not distinguishable by the human eye.) Subsampling can be implemented either by averaging together groups of chrominance components or by simply picking one component from the group and discarding the rest.

The values for this pseudo-encoding are defined as follows:

-768 = 1X chrominance subsampling (no chrominance subsampling).
Chrominance components are sent for every pixel in the source image.

-767 = 4X chrominance subsampling. Chrominance components are sent for every
fourth pixel in the source image. This would typically be implemented using 4:2:0 subsampling (2X subsampling in both X and Y directions), but it could also be implemented using 4:1:1 subsampling (4X subsampling in the X direction.)

-766 = 2X chrominance subsampling. Chrominance components are sent for every
other pixel in the source image. This would typically be implemented using 4:2:2 subsampling (2X subsampling in the X direction.)

-765 = Grayscale. All chrominance components in the source image are
discarded.

-764 = 8X chrominance subsampling. Chrominance components are sent for every
8th pixel in the source image. This would typically be implemented using 4:1:0 subsampling (4X subsampling in the X direction and 2X subsampling in the Y direction.)

-763 = 16X chrominance subsampling. Chrominance components are sent for every
16th pixel in the source image. This would typically be implemented using 4X subsampling in both X and Y directions.

This pseudo-encoding was originally intended for use with JPEG-encoded subrectangles, but it could be used with other types of image encoding as well.

7.7.19 Extended Clipboard Pseudo-Encoding

A client which requests the *Extended Clipboard* pseudo-encoding is declaring that it supports the extended versions of the ClientCutText and ServerCutText messages.

The format of these messages are altered so that *length* is now signed rather than unsigned:

No. of bytes	Type	[Value]	Description
1	`U8`	6/3	*message-type*
3			*padding*
4	`S32`		*length*

A positive value of *length* indicates that the message follows the original format with 8859-1 text data:

No. of bytes	Type	[Value]	Description
length	`U8` array		*text*

A negative value of *length* indicates that the extended message format is used and *abs(length)* is the total number of following bytes.

All messages start with a header:

No. of bytes	Type	[Value]	Description
4	`U32`		*flags*

The bits of *flags* have the following meaning:

Bit	Description
0	*text*
1	*rtf*
2	*html*
3	*dib*
4	*files*
5-15	Reserved for future formats
16-23	Reserved
24	*caps*
25	*request*
26	*peek*
27	*notify*
28	*provide*
29-31	Reserved for future actions

The different formats are:

text
Plain, unformatted text using the UTF-8 encoding. End of line is represented by carriage-return and linefeed / newline pairs (values 13 and 10). The text must be followed by a terminating null even though the length is also explicitly given.

rtf
Microsoft Rich Text Format.

html
Microsoft HTML clipboard fragments.

dib
Microsoft Device Independent Bitmap v5. A file header must not be included.

files
Currently reserved but not defined.

If *caps* is set then the other bits indicate which formats and actions that the sender is willing to receive. Following *flags* is an array indicating the maximing unsolicited size for each format:

No. of bytes	Type	[Value]	Description
formats * 4	`U32` array		*sizes*

The number of entries in *sizes* corresponds to the number of format bits set in *flags* (bit 0-15).

The server must send a ServerCutText message with *caps* set on each SetEncodings message received which includes the *Extended Clipboard* pseudo-encoding.

The client may send a ClientCutText message with *caps* set back to indicate its capabilities. Otherwise the client is assumed to support *text*, *rtf*, *html*, *request*, *notify* and *provide* and a maximum size of 20 MiB for *text* and 0 bytes for the other types.

Note that it is recommended to set all sizes to 0 bytes to force all clipboard updates to be sent in the form of a *notify* action. Failing to do so makes it ambiguous if an incoming message indicates a completely new set of formats, or an update to previous formats caused by size restrictions. It may be possible to also resolve this ambiguity using *peek* actions.

Also be aware that there are some implementations that deviate from the behaviour specified here. The prominent changes are that *dib* is also in the default set of supported formats, that default limits are now 10 MiB for *text*, 2 MiB for *rtf* and *html*, and 0 bytes for *dib*, and that the client ignores the advertised formats and maximum sizes given in the *caps* message.

If *caps* is not set then only one of the other actions (bit 24-31) may be set.

request
The recipient should respond with a *provide* message with the clipboard data for the formats indicated in *flags*. No other data is provided with this message.

peek
The recipient should send a new *notify* message indicating which formats are available. No other bits in *flags* need to be set and no other data is provided with this message.

notify
This message indicates which formats are available on the remote side and should be sent whenever the clipboard changes, or as a response to a *peek* message. The available formats are specified in *flags* and no other data is provided with this message.

provide
This message includes the actual clipboard data and should be sent whenever the clipboard changes and the data for each format is less than the respective specified maximum size, or as a response to a *request* message.

The header is followed by a Zlib [3] stream which contains a pair of *size* and *data* for each format indicated in *caps*:

No. of bytes	Type	[Value]	Description
4	`U32`		*size*
size	`U8` array		*data*