The RFB Protocol

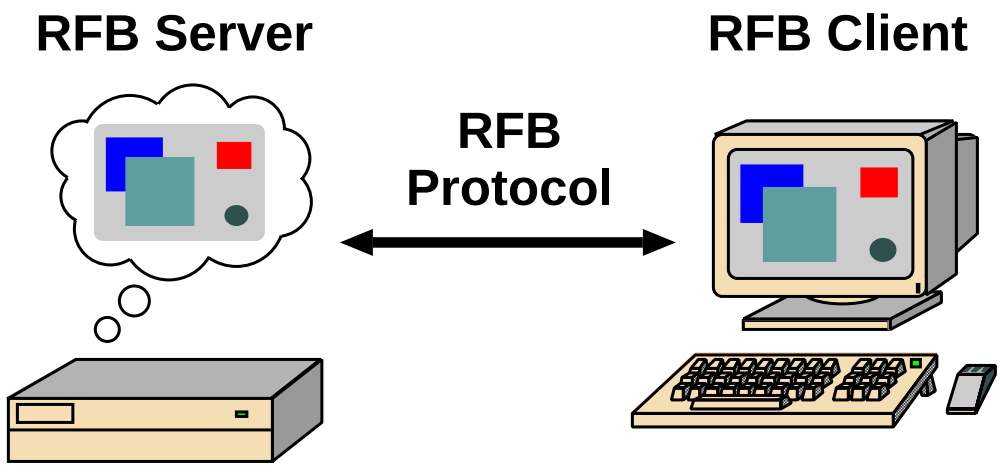
Tristan Richardson
Kenneth R. Wood
ORL
Cambridge

Version 3.3
Jan 1998

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 3.1/95/NT and Macintosh.

The remote endpoint where the user sits (i.e. the display plus keyboard and/or pointer) is called the RFB client. 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 pre-

served. 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.

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.

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.

The protocol can be extended by adding new encoding types. The encoding types defined at present are *raw* encoding, *copy rectangle* encoding, *RRE (rise-and-run-length)* encoding, *CoRRE (Compact RRE)* encoding and *hextile* encoding. In practice we normally use only the *hextile* and *copy rectangle* encodings since they provide the best compression for typical desktops. Other examples of possible encodings include JPEG for still images and MPEG for efficient transmission of moving images.

4.1 Raw encoding

The simplest encoding type is raw pixel data. In this case the data consists of *n* pixel values where *n* is the width times the 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.

4.2 Copy Rectangle encoding

The *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 *copy rectangle* encoding.

4.3 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 are not only compressed to the same degree or better than is possible with run-length encoding

but, more importantly, they arrive at the client in a form which can be rendered immediately and efficiently by the simplest of graphics engines. Thus, RRE is aimed at situations where compression is desired but the RFB client is insufficiently powerful to perform any decompression fast enough to maintain interactive performance.

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 resulting encoding on the wire consists of a background pixel value, V_b (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 $(\neq V_b)$ 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.

4.4 CoRRE encoding

CoRRE is a variant of RRE, where we guarantee that the largest rectangle sent is no more than 255x255 pixels. A server which wants to send a rectangle larger than this simply splits it up and sends several smaller RFB rectangles. Within each of these smaller rectangles, a single byte can then be used to represent the dimensions of the subrectangles. For a typical desktop, this results in better compression than RRE. In fact, the best compression is achieved when we limit the rectangle size even more - current implementations use a maximum of 48x48. This is because rectangles which do not encode well (typically those containing image data) are sent as raw, while the ones which do encode well are sent as CoRRE. The smaller the maximum rectangle size, the finer the granularity of this decision. With RRE, the whole original rectangle must either be sent as RRE, or the whole thing sent as raw. However, since there is a certain overhead incurred by each RFB rectangle, making the maximum rectangle size too small (and thus increasing the number of RFB rectangles), results in worse compression.

4.5 Hextile encoding

Hextile is a variation on the CoRRE 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. Unlike CoRRE, tiles are not top-level RFB rectangles. When splitting the original rectangle into tiles this is done in a predetermined way. This means that the position and size of each tile do not have to be explicitly specified - the encoded contents of the tiles simply follow one another in the predetermined order. The ordering of tiles that we use is starting at the top left going in left-to-right, top-to-bottom 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 - this is specified by a type byte for each tile. Each tile has a background pixel value, as before. However, 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. 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.

5 Protocol Messages

The RFB protocol can operate over any reliable transport, either byte-stream or message-based. There are two stages to the protocol; an initial handshaking phase followed by the normal protocol interaction.

The initial handshaking consists of *ProtocolVersion*, *Authentication*, *ClientInitialisation* and *ServerInitialisation* messages, as described below. Note that both client and server send a *ProtocolVersion* message.

The protocol proceeds to the normal interaction stage after the *ServerInitialisation* message. At this stage, 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 `CARD8`, `CARD16`, `CARD32`, `INT8`, `INT16`, `INT32`. 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).

5.1 Initial Handshaking Messages

5.1.1 ProtocolVersion

Handshaking begins by the server sending the client a *ProtocolVersion* message. This lets the client know which is the latest 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).

It is intended that both clients and servers may provide some level of backwards compatibility by this mechanism. Servers in particular should attempt to provide backwards compatibility, and even forwards compatibility to some extent. For example if a client demands version 3.1 of the protocol, a 3.0 server can probably assume that by ignoring requests for encoding types it doesn't understand, everything will still work OK. This will probably not be the case for changes in the major version number.

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)

5.1.2 Authentication

Once the protocol version has been decided, the server then sends a word indicating the authentication scheme to be used on the connection:

No. of bytes	Type [Value]	Description
4	CARD32	*authentication-scheme*:
	0	***connection failed***
	1	***no authentication***
	2	***VNC authentication***

This is followed by data specific to the *authentication-scheme*:

- ***connection failed*** - 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 [Value]	Description
4	CARD32	*reason-length*
reason-length	CARD8 array	*reason-string*

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

- ***no authentication*** - no authentication is needed. The protocol continues with the *ClientInitialisation* message.

- ***VNC authentication*** - VNC authentication is to be used. This is followed by a random 16-byte challenge:

No. of bytes	Type [Value]	Description
16	CARD8	*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 [Value]	Description
16	CARD8	*response*

  The server sends a word to inform the client whether authentication was successful. If so, the protocol continues with the *ClientInitialisation* message; if not the server closes the connection:

No. of bytes	Type [Value]	Description
4	CARD32	*status*:
	0	***OK***
	1	***failed***
	2	***too-many***

  If the server decides that ***too-many*** authentication failures have occurred, it should not allow immediate reconnection by the same client.

5.1.3 ClientInitialisation

Once the client and server are sure that they're happy to talk to one another, the client sends an initialisation message:

No. of bytes	Type [Value]	Description
1	CARD8	*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.

5.1.4 ServerInitialisation

After receiving the *ClientInitialisation* message, the server sends a *ServerInitialistion* 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 [Value]	Description
2	CARD16	*framebuffer-width*
2	CARD16	*framebuffer-height*
16	PIXEL_FORMAT	*server-pixel-format*
4	CARD32	*name-length*
name-length	CARD8 array	*name-string*

where PIXEL_FORMAT is

No. of bytes	Type [Value]	Description
1	CARD8	*bits-per-pixel*
1	CARD8	*depth*
1	CARD8	*big-endian-flag*
1	CARD8	*true-colour-flag*
2	CARD16	*red-max*
2	CARD16	*green-max*
2	CARD16	*blue-max*
1	CARD8	*red-shift*
1	CARD8	*green-shift*
1	CARD8	*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 (section 5.2.1).

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).

Currently there is little or no support for colour maps. Some preliminary work was done on this, but is incomplete. It was intended to be something like this:

> 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 can be set either by the client using the *FixColourMapEntries* message (section 5.2.2) or by the server using the *SetColourMapEntries* message (section 5.3.2).

The FixColourMapEntries message is not supported by any currently available servers, and colour maps are only supported at all by the X-based server. In fact, for proper colour map support the client probably needs to be able to specify particular pixel values which the server should not use. This may be added in future versions of the protocol, but don't hold your breath.

5.2 Client to server messages

5.2.1 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 *ServerInitialisation* message (section 5.1.4).

Currently there is little or no support for colour maps. Some preliminary work was done on this, but is incomplete. It was intended to be something like this:

> If *true-colour-flag* is zero (false) then this indicates that a "colour map" is to be used. The client can fix any of the entries in the colour map using the *FixColourMapEntries* message (section 5.2.2). Any entries not fixed by the client may be set dynamically as desired by the server using the *SetColourMapEntries* message (section 5.3.2). Immediately after the client has sent this message the colour map is empty, even if entries had previously been fixed by the client or set by the server.

No. of bytes	Type	[Value]	Description
1	`CARD8`	`0`	*message-type*
3			*padding*
16	`PIXEL_FORMAT`		*pixel-format*

where `PIXEL_FORMAT` is as described in section 5.1.4:

No. of bytes	Type	[Value]	Description
1	`CARD8`		*bits-per-pixel*
1	`CARD8`		*depth*
1	`CARD8`		*big-endian-flag*
1	`CARD8`		*true-colour-flag*
2	`CARD16`		*red-max*
2	`CARD16`		*green-max*
2	`CARD16`		*blue-max*
1	`CARD8`		*red-shift*
1	`CARD8`		*green-shift*
1	`CARD8`		*blue-shift*
3			*padding*

5.2.2 FixColourMapEntries

Currently there is little or no support for colour maps. Some preliminary work was done on this, but is incomplete. It was intended to be something like this:

> When the pixel format uses a "colour map", this message tells the server that the specified pixel values are mapped to the given RGB intensities. This means that the server may not subsequently specify RGB intensities for these pixel values using *SetColourMapEntries* messages (section 5.3.2).
>
> If the client has a fixed colour map it can simply send a *FixColourMapEntries* message describing its entire colour map and the server will then translate all pixel values as appropriate for the client's colour map. In this case the server cannot send any *SetColourMapEntries* messages.
>
> Note that despite the name, the client can, if it desires, send a *FixColourMapEntries* message at any time. This includes messages remapping the same pixel values to different RGB intensities. However, the only way the client can indicate that a pixel value is no longer fixed is by sending another *SetPixelFormat* message, which clears the entire colour map.

The FixColourMapEntries message is not supported by any currently available servers.

No. of bytes	Type	[Value]	Description
1	CARD8	1	*message-type*
1			*padding*
2	CARD16		*first-colour*
2	CARD16		*number-of-colours*

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

No. of bytes	Type	[Value]	Description
2	CARD16		*red*
2	CARD16		*green*
2	CARD16		*blue*

5.2.3 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.

No. of bytes	Type	[Value]	Description
1	CARD8	2	*message-type*
1			*padding*
2	CARD16		*number-of-encodings*

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

No. of bytes	Type	[Value]	Description
4	CARD32		*encoding-type*
		0	***raw*** encoding
		1	***copy rectangle*** encoding
		2	***RRE*** encoding
		4	***CoRRE*** encoding
		5	***hextile*** encoding

5.2.4 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 *FramebufferUpdateRequest*s.

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 *copy rectangle* 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 *FramebufferUpdateRequest*s to avoid hogging the network.

No. of bytes	Type	[Value]	Description
1	CARD8	3	*message-type*
1	CARD8		*incremental*
2	CARD16		*x-position*
2	CARD16		*y-position*
2	CARD16		*width*
2	CARD16		*height*

5.2.5 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. 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.

No. of bytes	Type	[Value]	Description
1	CARD8	4	*message-type*
1	CARD8		*down-flag*
2			*padding*
4	CARD32		*key*

For most ordinary keys, the "keysym" is the same as the corresponding ASCII value. 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

Key name	Keysym value
Up	0xff52
Right	0xff53
Down	0xff54
F1	0xffbe
F2	0xffbf
...	...
F12	0xffc9
Shift	0xffe1
Control	0xffe3
Meta	0xffe7
Alt	0xffe9

5.2.6 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).

No. of bytes	Type	[Value]	Description
1	CARD8	5	*message-type*
1	CARD8		*button-mask*
2	CARD16		*x-position*
2	CARD16		*y-position*

5.2.7 ClientCutText

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

No. of bytes	Type	[Value]	Description
1	CARD8	6	*message-type*
3			*padding*
4	CARD32		*length*
length	CARD8 array		*text*

5.3 Server to client messages

5.3.1 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	CARD8	0	*message-type*
1			*padding*
2	CARD16		*number-of-rectangles*

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

No. of bytes	Type	[Value]	Description
2	CARD16		*x-position*
2	CARD16		*y-position*
2	CARD16		*width*
2	CARD16		*height*
4	CARD32		*encoding-type*:
		0	***raw*** encoding
		1	***copy rectangle*** encoding
		2	***RRE*** encoding
		4	***CoRRE*** encoding
		5	***hextile*** encoding

followed by the pixel data in the specified encoding.

- For ***raw*** encoding the data consists of $width \times height$ pixel values. The values simply represent each pixel in left-to-right scanline order.
- For ***copy rectangle*** encoding the data consists of:

No. of bytes	Type	[Value]	Description
2	CARD16		*src-x-position*
2	CARD16		*src-y-position*

- For ***RRE*** encoding the data begins with the header:

No. of bytes	Type	[Value]	Description
4	CARD32		*number-of-subrectangles*
n	CARD<8n>		*background-pixel-value*

where $8n$ is the number of *bits-per-pixel* as agreed by the client and server – either in the *ServerInitialisation* message (section 5.1.4) or a *SetPixelFormat* message (section 5.2.1). This is followed by *number-of-subrectangles* instances of the following structure:

No. of bytes	Type [Value]	Description
n	CARD<8n>	*subrect-pixel-value*
2	CARD16	*x-position*
2	CARD16	*y-position*
2	CARD16	*width*
2	CARD16	*height*

- For ***CoRRE*** encoding the data begins with the header:

No. of bytes	Type [Value]	Description
4	CARD32	*number-of-subrectangles*
n	CARD<8n>	*background-pixel-value*

where $8n$ is the number of *bits-per-pixel* as agreed by the client and server – either in the *ServerInitialisation* message (section 5.1.4) or a *SetPixelFormat* message (section 5.2.1). This is followed by *number-of-subrectangles* instances of the following structure:

No. of bytes	Type [Value]	Description
n	CARD<8n>	*subrect-pixel-value*
1	CARD16	*x-position*
1	CARD16	*y-position*
1	CARD16	*width*
1	CARD16	*height*

- For ***hextile*** encoding the rectangle is divided up into *tiles* of 16x16 pixels, starting at the top left going in left-to-right, top-to-bottom order. If the width of the 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 is not an exact multiple of 16 then the height of each tile in the final row will also be smaller.

  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	CARD8	*subencoding-mask*:
	1	***Raw***
	2	***BackgroundSpecified***
	4	***ForegroundSpecified***
	8	***AnySubrects***
	16	***SubrectsColoured***

  If the ***Raw*** bit is set then the other bits are irrelevant; $width \times height$ 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 [Value]	Description
n	CARD<8n>	*background-pixel-value*

where $8n$ is the number of *bits-per-pixel* as agreed by the client and server – either in the *ServerInitialisation* message (section 5.1.4) or a *SetPixelFormat* message (section 5.2.1). 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 [Value]	Description
n	CARD<8n>	*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 [Value]	Description
1	CARD8	*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 [Value]	Description
n	CARD<8n>	*subrect-pixel-value*
1	CARD8	*x-and-y-position*
1	CARD8	*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 [Value]	Description
1	CARD8	*x-and-y-position*
1	CARD8	*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.

5.3.2 SetColourMapEntries

Currently there is little or no support for colour maps. Some preliminary work was done on this, but is incomplete. It was intended to be something like this:

> 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. The server may only specify pixel values for which the client has not already set the RGB intensities using *FixColourMapEntries* (section 5.2.2).

Currently, colour maps are only supported at all by the X-based server.

No. of bytes	Type	[Value]	Description
1	CARD8	1	*message-type*
1			*padding*
2	CARD16		*first-colour*
2	CARD16		*number-of-colours*

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

No. of bytes	Type	[Value]	Description
2	CARD16		*red*
2	CARD16		*green*
2	CARD16		*blue*

5.3.3 Bell

Ring a bell on the client if it has one.

No. of bytes	Type	[Value]	Description
1	CARD8	2	*message-type*

5.3.4 ServerCutText

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

No. of bytes	Type	[Value]	Description
1	CARD8	3	*message-type*
3			*padding*
4	CARD32		*length*
length	CARD8 array		*text*