
# PALACE SERVER PROTOCOLS

*Copyright © 1999 Communities.com, All rights reserved.*

*The Palace Server Protocols*

*May, 1999*

*This document and the software described in it are furnished under license and may be used or copied only in accordance with such license. Except as permitted by such license, the contents of this document may not be disclosed to third parties, copied, or duplicated in any form, in whole or in part, without the prior written permission of Communities.com.*

*The contents of this document are for informational use only, and the contents are subject to change without notice. Communities.com assumes no responsibility or liability for any errors or inaccuracies that may appear in this book.*

*Restricted Rights Legend. For defense agencies: Use, reproduction, or disclosure is subject to restrictions set fort in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 52.227-7013, and or similar successor clauses in the FAR, or the DOD or NASA FAR Supplement.*

*Unpublished right reserved under the Copyright Laws of the United States.*

*The Palace Inc., The Palace, PalacePresents, Palace Authoring Wizard, PalaceEvents and PalaceServer are trademarks and The Palace logo is a registered trademark of Communities.com. All rights reserved. All other trademarks are the property of their respective owners.*

*Printed in the USA.*

# Table of Contents

This document describes the current Palace client/server protocol.



©1999 Electric Communities, all rights reserved. Proprietary & confidential. 

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

# 1 Types and Notation Conventions

This section describes the Palace type and notation conventions.

## 1.1 Struct Notation

This document describes messages and other data structures using a variant of the C struct declaration notation. However, Palace message and data structures are not C code. Specifically, array fields are not pointers; rather, the array members themselves are contained by the struct. Thus, the following example is a structure consisting of 40 bytes: a 4-byte integer `first`, a 32-byte array `anArray`, and a 4-byte integer `last`.
:

```
struct Example1 {
    uint32 first;
    char   anArray[32];
    uint32 last;
}
```

When an array is field declared without explicit dimensions, it means that the size of the array is to be known from context, not that the field contains a pointer. Thus:

```
struct Example2 {
    uint32 first;
    char   anArray[];
    uint32 last;
}
```

is a structure consisting of 8 or more bytes: a 4-byte integer `first`, a character array `anArray` of unknown dimension, and a 4-byte integer `last`. It is up to the user/interpreter of the structure to know what the dimension really is.

When an array field is declared with a dimension that is the name of another field, it means that the named field contains the dimension. Thus:

```
struct Example2 {
    uint32 first;
    char   anArray[first];
    uint32 last;
}
```

is a structure consisting of 8 or more bytes: a 4-byte integer `first`, followed by

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

`first` bytes of the array `anArray`, then a 4-byte integer `last`.

## 1.2 Primitive Types

The following primitive types are assumed:

| | |
|---|---|
| `sint8` | signed, 8-bit (1-byte) integer |
| `uint8` | unsigned, 8-bit (1-byte) integer |
| `char` | 1-byte character (sign not relevant) |
| `sint16` | signed, 16-bit (2-byte) integer |
| `uint16` | unsigned, 16-bit (2-byte) integer |
| `sint32` | signed, 32-bit (4-byte) integer |
| `uint32` | unsigned, 32-bit (4-byte) integer |

**do we want to mention this:

## 1.3 Non-primitive Types

The Palace code uses a number of different representations for strings. The two major styles of string are:

Pascal-style string, consisting of a length byte followed by the requisite number of characters:

```
struct PString {
    uint8 length;
    char  chars[length];
}
```

C-style string, consisting of a sequence of characters whose end is delimited by a zero byte:

```
struct CString {
    char chars[];
}
```

The Pascal-style string is limited to a length of 255 characters, whereas the C-style string is not so limited. The C-style string cannot contain embedded nul characters, whereas the Pascal-style string is not so limited..

In addition, the MacOS defines two more string types, each of which simply embeds a Pascal-style string in a fixed-size structure. The only difference between the two is the size of the structure:

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

```
struct Str31 {
    uint8 length;
    char  chars[31];
}

struct Str63 {
    uint8 length;
    char  chars[63];
}
```

(the Mac also defines a `Str255` whose definition is analogous, but it is not used in the Palace protocol). Note that sending these latter two structs in messages over the wire is wasteful of bandwidth, as the extra bytes to pad the structure to its full length server no useful purpose. Moreover, if a client or server does not properly clear these padding bytes when strings are assigned into these structures, it is possible that private information (such as fragments of passwords) may be revealed unintentionally.

Another commonly used type is a `Point`, which represents a screen position or offset:

```
struct Point {
    sint16 v;
    sint16 h;
}
```

The client and server maintain collections of "assets", which are chunks of digital media such as images and sounds. These are used for features such as propsAssets are typed, with the type being specified by yet another 4-character ASCII value stored in a 32 bit integer:

```
typedef sint32 AssetType;
```

Predefined asset types include:

```
RT_PROP          0x50726f70    /* 'Prop' */
RT_USERBASE      0x55736572    /* 'User' */
RT_IPUSERBASE    0x49557372    /* 'IUsr' */
```

`RT_PROP` is for assets which are props. `RT_USERBASE` is for assets which represent users (the server has vestigial support for a user database stored as a collection of assets). `RT_IPUSERBASE` is defined but not used and is an historical artifact.

Each asset is identified by an ID number and characterized by a CRC code which can be used to verify that the asset is what it is supposed to be. These two values frequently travel together in an `AssetSpec` struct:

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

```
struct AssetSpec {
    sint32 id;
    uint32 crc;
}
```

The ID identifies the asset within a namespace defined by the asset type. The CRC is computed from the bits of the asset itself and can be used to attempt to verify that a particular asset is the one it is supposed to be. In addition, the CRC is sometimes used as an asset lookup key in its own right

Finally, certain integer types are used as ID numbers for important entities:

```
typedef sint32 UserID;
typedef sint16 RoomID;
typedef sint16 HotspotID;
```

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

# 2 Client/Server Message Generalities

## 2.1 The Generic Client/Server Message

While the format of the different messages varies from message type to message type, all messages between the client and server share a common outer structure:

```
struct ClientMsg {
    uint32 eventType;   /* 32-bit opcode */
    uint32 length;      /* length of message body */
    sint32 refNum;      /* arbitrary integer operand */
    uint8  msg[length]; /* message body */
}
```

*eventType* is a 4-byte message opcode. It (indicates the operation to be performed or the information being requested or provided. By convention its value is a four-character ASCII mnemonic, though this is important only in one case: the sequence of characters in the *eventType* field of the first message received from a remote machine (a MSG_TIYID message,) is used by the receiver to determine the byte ordering (i.e., endianess) of the sender, so that other data values can be decoded properly.

NOTE: *his value is declared signed but probably should be unsigned for clarity; having it be signed probably does not cause any harm.*

`refnum` is an arbitrary 32-bit integer message argument. The interpretation of this value is specific to the particular message using it. Some messages require only a single, numeric parameter, which they place in this position with no further encoding required. On the other hand, many messages have no parameters at all or require a more complex parameter structure, in which case this value is just wasted space.

`length` and `msg` describe the rest of the message. `length` is simply the number of bytes in `msg`. `msg` contains a struct whose internal form is specific to the particular message being sent (see the individual message descriptions in **3. The Client/Server Messages** below). If a particular message requires no additional parameters, or has just a single parameter that is sent in the `refnum` field, then `length` will be 0 and the `msg` field will not be present, yielding a 12-byte structure.

Note: *`length` is declared signed but probably should be unsigned; as a practical*

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

*matter, having it be signed is probably harmless as no message should be over two gigabytes anyhow.*

## 2.2 Client/Server Message Summary

The following table lists all the client/server messages currently defined.

"Name" is the symbolic name used in the source code and in this document to identify the message.

"Hex value" is the hexadecimal value of the *eventType* field in messages of this type.

"Mnemonic" is the 4-character ASCII string from which the *eventType* value is derived.

"Usage" describes the pattern of use that this message experiences, according to the following legend:

| | |
|---|---|
| server → client | Message gets sent from the server to the client |
| server ← client | Message gets sent from the client to the server |
| server ↔ client | Client and server both send and receive the message |
| server — client | According to source code, message is defined but unused |

*Note: When a message is shown in the below table as unused, it is noted in the table but not appear in the individual message descriptions.*

In some cases a message is defined as being both sent and received by both the client and the server, but the actual message format used is different depending on the direction the message is traveling. This is like having two different messages that can get away with sharing a message type value because the contexts of their usage are disjoint. Details of cases of this nature are given in the individual message descriptions below.

| Name | Hex value | Mnemonic | Usage |
|---|---|---|---|
| MSG_ALTLOGONREPLY | 0x72657032 | 'rep2' | server → client |
| MSG_ASSETNEW | 0x61417374 | 'aAst' | server — client |
| MSG_ASSETQUERY | 0x71417374 | 'qAst' | server ↔ client |
| MSG_ASSETREGI | 0x72417374 | 'rAst' | server ← client |
| MSG_ASSETSEND | 0x73417374 | 'sAst' | server → client |
| MSG_AUTHENTICATE | 0x61757468 | 'auth' | server → client |
| MSG_AUTHRESPONSE | 0x61757472 | 'autr' | server ← client |

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

| | | | |
|---|---|---|---|
| MSG_BLOWTHRU | 0x626c6f77 | 'blow' | server ↔ client |
| MSG_DISPLAYURL | 0x6475726c | 'durl' | server → client |
| MSG_DIYIT | 0x72796974 | 'ryit' | server — client |
| MSG_DOORLOCK | 0x6c6f636b | 'lock' | server ↔ client |
| MSG_DOORUNLOCK | 0x756e6c6f | 'unlo' | server ↔ client |
| MSG_DRAW | 0x64726177 | 'draw' | server ↔ client |
| MSG_EXTENDEDINFO | 0x73496e66 | 'sInf' | server ↔ client |
| MSG_FILENOTFND | 0x666e6665 | 'fnfe' | server → client |
| MSG_FILEQUERY | 0x7146696c | 'qFil' | server ← client |
| MSG_FILESEND | 0x7346696c | 'sFil' | server → client |
| MSG_GMSG | 0x676d7367 | 'gmsg' | server ← client |
| MSG_HTTPSERVER | 0x48545450 | 'HTTP' | server → client |
| MSG_INITCONNECTION | 0x634c6f67 | 'cLog' | server — client |
| MSG_KILLUSER | 0x6b696c6c | 'kill' | server ← client |
| MSG_LISTOFALLROOMS | 0x724c7374 | 'rLst' | server ↔ client |
| MSG_LISTOFALLUSERS | 0x754c7374 | 'uLst' | server ↔ client |
| MSG_LOGOFF | 0x62796520 | 'bye ' | server ↔ client |
| MSG_LOGON | 0x72656769 | 'regi' | server ← client |
| MSG_NAVERROR | 0x73457272 | 'sErr' | server → client |
| MSG_NOOP | 0x4e4f4f50 | 'NOOP' | server — client |
| MSG_PICTDEL | 0x46505371 | 'FPSq' | server — client |
| MSG_PICTMOVE | 0x704c6f63 | 'pLoc' | server ↔ client |
| MSG_PICTNEW | 0x6e506374 | 'nPct' | server — client |
| MSG_PICTSETDESC | 0x73506374 | 'sPct' | server — client |
| MSG_PING | 0x70696e67 | 'ping' | server ↔ client |
| MSG_PONG | 0x706f6e67 | 'pong' | server ↔ client |
| MSG_PROPDEL | 0x64507270 | 'dPrp' | server ↔ client |
| MSG_PROPMOVE | 0x6d507270 | 'mPrp' | server ↔ client |
| MSG_PROPNEW | 0x6e507270 | 'nPrp' | server ↔ client |
| MSG_PROPSETDESC | 0x73507270 | 'sPrp' | server — client |
| MSG_RESPORT | 0x72657370 | 'resp' | server — client |
| MSG_RMSG | 0x726d7367 | 'rmsg' | server ← client |
| MSG_ROOMDESC | 0x726f6f6d | 'room' | server → client |
| MSG_ROOMDESCEND | 0x656e6472 | 'endr' | server → client |
| MSG_ROOMGOTO | 0x6e617652 | 'navR' | server ← client |
| MSG_ROOMNEW | 0x6e526f6d | 'nRom' | server ← client |
| MSG_ROOMSETDESC | 0x73526f6d | 'sRom' | server ↔ client |
| MSG_SERVERDOWN | 0x646f776e | 'down' | server → client |
| MSG_SERVERINFO | 0x73696e66 | 'sinf' | server → client |
| MSG_SERVERUP | 0x696e6974 | 'init' | server — client |
| MSG_SMSG | 0x736d7367 | 'smsg' | server ← client |

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

| | | | |
|---|---|---|---|
| MSG_SPOTDEL | 0x6f705364 | 'opSd' | server ← client |
| MSG_SPOTMOVE | 0x636f4c73 | 'coLs' | server ↔ client |
| MSG_SPOTNEW | 0x6f70536e | 'opSn' | server ← client |
| MSG_SPOTSETDESC | 0x6f705373 | 'opSs' | server — client |
| MSG_SPOTSTATE | 0x73537461 | 'sSta' | server ↔ client |
| MSG_SUPERUSER | 0x73757372 | 'susr' | server ← client |
| MSG_TALK | 0x74616c6b | 'talk' | server ↔ client |
| MSG_TIMYID | 0x74696d79 | 'timy' | server — client |
| MSG_TIYID | 0x74697972 | 'tiyr' | server ↔ client |
| MSG_TROPSER | 0x70736572 | 'pser' | server — client |
| MSG_USERCOLOR | 0x75737243 | 'usrC' | server ↔ client |
| MSG_USERDESC | 0x75737244 | 'usrD' | server ↔ client |
| MSG_USERENTER | 0x77707273 | 'wprs' | server — client |
| MSG_USEREXIT | 0x65707273 | 'eprs' | server → client |
| MSG_USERFACE | 0x75737246 | 'usrF' | server ↔ client |
| MSG_USERLIST | 0x72707273 | 'rprs' | server → client |
| MSG_USERLOG | 0x6c6f6720 | 'log ' | server → client |
| MSG_USERMOVE | 0x754c6f63 | 'uLoc' | server ↔ client |
| MSG_USERNAME | 0x7573724e | 'usrN' | server ↔ client |
| MSG_USERNEW | 0x6e707273 | 'nprs' | server → client |
| MSG_USERPROP | 0x75737250 | 'usrP' | server ↔ client |
| MSG_USERSTATUS | 0x75537461 | 'uSta' | server → client |
| MSG_VERSION | 0x76657273 | 'vers' | server → client |
| MSG_WHISPER | 0x77686973 | 'whis' | server ↔ client |
| MSG_WMSG | 0x776d7367 | 'wmsg' | server — client |
| MSG_XTALK | 0x78746c6b | 'xtlk' | server ↔ client |
| MSG_XWHISPER | 0x78776973 | 'xwis' | server ↔ client |

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

# 3 The Client/Server Messages

## 3.1 MSG_ALTLOGONREPLY

The server may send this message to a client in response to the `MSG_LOGON` message (assuming the logon is successful, of course). It contains various information describing the situation in which newly-logged-on users find themselves. This message is not always sent, however. Currently only one circumstance triggers this, and that is when the server is running in guests-are-members mode.

The `refnum` field is the user ID of the user logging on.

The `msg` field is an `AuxRegistrationRec`. See **§3.20 `MSG_LOGON`** for details on this struct:

```
struct ClientMsg_altLogonReply {
     AuxRegistrationRec rec;
}
```

## 3.2 MSG_ASSETQUERY

This requests the receiver to send the sender a particular asset. The server uses it to request props from the client, and the client uses it to request arbitrary assets. Assets are identified by type, ID, and a CRC. Asset types are 4-character ASCII codes; see the description of `AssetType` in **1.3 Non-primitive Types**. Asset IDs are arbitrary 4-byte integers. Ordinarily the CRC is used to check the asset for validity, but a CRC value of 0 indicates "don't care".

The only asset the server ever asks the client for is type `RT_PROP`. It is unclear what asset types the client may ask the server for. It is unclear to me what it means for the server to send this message to the client. A server normally responds to a `MSG_ASSETQUERY` with a `MSG_ASSETSEND`, but the server is not set up to receive a `MSG_ASSETSEND` from the client.

The `refnum` field is not used in this message and should be set to 0.

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

The `msg` field is a `ClientMsg_assetQuery` struct:

```
struct ClientMsg_assetQuery {
    AssetType type;
    AssetSpec spec;
}
```

The `type` and `spec` fields together describe the asset desired.

## 3.3 MSG_ASSETREGI and MSG_ASSETSEND

These two messages are used to transmit assets from one machine to another. The two messages are identical except for the message type. `MSG_ASSETREGI` is used when the client is sending assets to the server. `MSG_ASSETSEND` is used when the server is sending assets to the client.

The message format is designed to enable assets to be transmitted in blocks, with each block sent in a separate message.

The `refnum` field is not used in this message and should be set to 0.

The `msg` field is a `ClientMsg_assetSend` struct:

```
struct ClientMsg_assetSend {
    AssetType type;
    AssetSpec spec;
    sint32    blockSize;
    sint32    blockOffset;
    sint16    blockNbr;
    sint16    nbrBlocks;
    AssetDescriptor desc;
    uint8     data[blockSize];
}
```

`type` and `spec` identify the asset being sent.

`blockSize` is the size of the block being sent in this message.

`blockOffset` is the offset from the start of the asset at which the block begins.

`blockNbr` is the block number of the block being sent in this message. Blocks are numbered starting with 0.

`data` contains the actual bytes of the asset itself.

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

`desc` is present only in messages containing the first block of an asset (i.e., `blockNbr` is 0). It is an `AssetDescriptor` struct, giving information about the asset as a whole:

```
struct AssetDescriptor {
    uint32 flags;
    uint32 size;
    Str31  name;
}
```

`flags` are various flag bits characterizing the asset. These are used by the client only.

`size` is the total size (in bytes) of the asset.

`name` is a name for the asset. The name is used by the client only. In particular, the name is used to identify the asset in scripts.

In theory, as mentioned above, this message format allows an asset to be transmitted in pieces. However, the Unix server only understands assets sent whole (i.e., as a single block containing the entire asset). Thus, in practice, the message *always* contains an `AssetDescriptor`, `blockSize` is *always* equal to `size`, `blockOffset` is *always* 0, `blockNbr` is *always* 0, and `nbrBlocks` is *always* 1.

## 3.4 MSG_AUTHENTICATE

The server sends this message to request the client to authenticate its user to the server. The client responds with a `MSG_AUTHRESPONSE` message.

The `refnum` field is not used in this message and should be set to 0.

There are no parameters in this message, so the `length` field should be 0 and the `msg` field should be empty:

```
struct ClientMsg_authenticate {
}
```

## 3.5 MSG_AUTHRESPONSE

The client sends this message in response to a `MSG_AUTHENTICATE` request from the server. It provides a user name and password.

The `refnum` field is not used in this message and should be set to 0.

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

The `msg` field is a PString consisting of the concatenation of the user name, a colon (“`:`”) character, and the password, all “encrypted” with the Palace string encryption:

```
struct ClientMsg_authResponse {
    PString nameAndPassword;
}
```

## 3.6 MSG_BLOWTHRU

This message provides a way for clients to use the server to relay messages to other clients. The interpretation of the messages being relayed is not specified, but must be agreed upon in advance by the clients involved. There is a standard tag on the header of the embedded message which plugin developers are assigned by us. Plugins are expected to place this tag on messages they send; the client will use it to route received blowthru messages to installed plugins. *Note: Currently, there isn't a mechanism to enforce that a plugin be well behaved in this regard.*

There are two forms of the `MSG_BLOWTHRU` message, one which the client sends to the server and one which the server sends to the client. The client-to-server form contains an embedded message and an encoding of who that embedded message should be relayed to. The server-to-client form actually relays the message.

In the client-to-server form:

The `refnum` field is an arbitrary value whose interpretation is between the clients; it gets relayed.

The `msg` field is a `ClientMsg_blowThru_toServer` struct:

```
struct ClientMsg_blowThru_toServer {
    uint32 flags;
    uint32 nbrUsers;
    UserID userIDs[nbrUsers]; /* iff nbrUsers >= 0 */
    uint32 pluginTag;
    uint8  embedded[];
}
```

`flags` are various flag bits *Note:these are unused*.

`nbrUsers` indicates who to relay the blowthru message to: a negative value (canonically -1, though the server does not enforce this) means the message should be relayed to all users on the server; a value of 0 means the message

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

should be relayed to all users in the same room with the sender; a positive value indicates a number of individual users to whom the message should be relayed, in which case `userIDs` is an array of that many `UserID`s identifying the recipients of the relay.

`pluginTag` is the (Communities.com assigned) tag identifying the client plugin to which the message is supposed to be routed.

`embedded` contains the actual message bytes of the message that is to be relayed.

In the server-to-client form:

The `refnum` field contains the value of the `refnum` field from the corresponding client-to-server message that is being relayed.

The `msg` field contains the bytes from the `pluginTag` and `embedded` fields from the corresponding client-to-server message that is being relayed:

```
struct ClientMsg_blowThru_toClient {
    uint32 pluginTag;
    uint8 embedded[];
}
```

## 3.7 MSG_DISPLAYURL

This message directs the client to display a particular URL in association with a particular Palace window pane. This message is used with the Java client (which runs in a browser) and Palace Presents (which uses ActiveX in collusion with Internet Explorer); these clients have the ability to display web pages by commanding the browser.

The `refnum` field contains the pane number of the pane in which the URL is to be displayed. ( *a browser pane in display that uses frames)*

The `msg` field contains a `CString` that is the URL itself:

```
struct ClientMsg_displayURL {
    CString url;
}
```

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

## 3.8 MSG_DOORLOCK and MSG_DOORUNLOCK

These two message are used to lock and unlock doors. The two messages are identical except for the message type. A client sends a `MSG_DOOR(UN)LOCK` message to the server requesting a door to be (un)locked. If the operation is successful, the server sends a matching `MSG_DOOR(UN)LOCK` message to the clients in the room with the door, informing them of the event. In either direction, the message format is the same.

The `refnum` field is not used in this message and should be set to 0.

The `msg` field is a `ClientMsg_doorLock` struct:

```
struct ClientMsg_doorLock {
    RoomID    roomID;
    HotspotID doorID;
}
```

`roomID` is the `RoomID` of the room containing the door to be (un)locked.

`doorID` is the `HotspotID` of hotspot in that room which is the door.

## 3.9 MSG_DRAW

This message is used to instruct the client to add or drawing operations to the collection of such operations describing the appearance of a room. A client sends a `MSG_DRAW` message to the server requesting that a particular drawing command be added to sender's room's set of draw commands. If the operation is successful (i.e., various format, resource limit, and permission checks succeed), the server sends a matching `MSG_DRAW` message to the clients in the room. In both directions the message format is the same.

The `refnum` field is not used in this message and should be set to 0.

The `msg` field contains a `DrawRecord` struct:

```
struct ClientMsg_draw {
    DrawRecord command;
}
```

where a `DrawRecord` consists of:

```
struct DrawRecord {
```

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

```
        LLRec  link;
        sint16 drawCmd;
        uint16 cmdLength;
        sint16 dataOfst;
        uint8  cmdData[cmdLength];
    }
```

`link` is an `LLRec` struct:

```
    struct LLRec {
        sint16 nextOfst;
        sint16 reserved;
    }
```

However, the `LLRec` struct is only used internally and its contents are ignored in the message

`drawCmd` is the draw operation opcode, taken from the set:

```
    DC_Path      0
    DC_Shape     1
    DC_Text      2
    DC_Detonate  3
    DC_Delete    4
    DC_Ellipse   5
```

`cmdLength` is the length of the operand data for the draw operation.

`dataOfst` is another field only used internally.

`cmdData` is an array `cmdLength` bytes of operand data for the draw operation.
What goes in this field depends on what `drawCmd` is.

The `DC_Delete` draw command deletes the most recent draw command from the room's
set. It has no operand bytes.

The `DC_Detonate` draw command deletes all draw commands from the rooms's set. It
has no operand bytes.

## *3.10MSG_EXTENDEDINFO*

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

This message has two forms: a client-to-server (“request”) form, in which the client requests various pieces of information from the server as well as providing various authenticating information of its own, and a server-to-client (“response”) form in which the server delivers the requested information or delivers an error indication that it could not provide what was asked.

In the request form, the `refnum` field is unused *(and presumably should be set to 0)*.

In the request form, the `msg` field is a `ClientMsg_extendedInfo_request` struct:

```
struct ClientMsg_extendedInfo_request {
    uint32 flags;
    ExtendedInfo info[];
}
```

`flags` is a set of bit flags indicating the information desired:

| | | |
|---|---|---|
| `SI_AVATAR_URL` | `0x00000001` | default avatar URL |
| `SI_SERVER_VERSION` | `0x00000002` | server version string |
| `SI_SERVER_TYPE` | `0x00000004` | server type |
| `SI_SERVER_FLAGS` | `0x00000008` | server flags |
| `SI_NUM_USERS` | `0x00000010` | number of users |
| `SI_SERVER_NAME` | `0x00000020` | server name |
| `SI_HTTP_URL` | `0x00000040` | HTTP (picture) URL |

`info` is a sequence of `ExtendedInfo` structs (described below).

In the response form, the `refnum` field is the `UserID` of the user to whom the message is sent (*the client receiving this will always see its own UserID)*.

In the response form, the `msg` field is just a sequence of `ExtendedInfo` structs:

```
struct ClientMsg_extendedInfo_response {
    ExtendedInfo info[];
}
```

Note that the request form and the response form are similar. The difference is that the request form has a `flags` field, while the response form does not.

In either form, information is passed in a sequence of `ExtendedInfo` structs, which have the form:

```
struct ExtendedInfo {
    sint32 id;
```

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

```
    sint32 length;
    uint8  buf[length];
}
```

`id` is a 4-character ASCII code identifying the piece of information being delivered. These codes (and associated information) fall into three families:

Extended information codes, in the request form:

| | | | |
|---|---|---|---|
| SI_EXT_NAME | 0x4E414D45 | 'NAME' | user name |
| SI_EXT_PASS | 0x50415353 | 'PASS' | password |
| SI_EXT_TYPE | 0x54595045 | 'TYPE' | client type |

Error response codes, in the response form when something goes wrong:

| | | | |
|---|---|---|---|
| SI_ERR_AUTH | 0x41555448 | 'AUTH' | authentication needed |
| SI_ERR_UNKN | 0x554E4B4E | 'UNKN' | unknown information requested |

Information response codes, in the response form providing requested info:

| | | | |
|---|---|---|---|
| SI_INF_AURL | 0x4155524C | 'AURL' | default avatar URL |
| SI_INF_VERS | 0x56455253 | 'VERS' | server version string |
| SI_INF_TYPE | 0x54595045 | 'TYPE' | server type |
| SI_INF_FLAG | 0x464C4147 | 'FLAG' | server flags |
| SI_INF_NUM_USERS | 0x4E555352 | 'NUSR' | number of users |
| SI_INF_NAME | 0x4E414D45 | 'NAME' | server name |
| SI_INF_HURL | 0x4855524C | 'HURL' | HTTP (picture) URL |

`length` is the length (in bytes) of the information itself, which is sent in `buf`. Each of the kinds of information has its own particular format:

`SI_EXT_NAME`: Provides the username of the user issuing the request.

```
struct ExtendedInfo_name {
    PString username;
}
```

`SI_EXT_PASS`: Provides the password of the user issuing the request.

```
struct ExtendedInfo_pass {
    PString password;
}
```

`SI_EXT_TYPE`: Identifies the type of client issuing the request. Currently the (Unix) server recognizes the following client type strings:

```
"68k Mac"
```

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

```
"PPC Mac"
"16Bit Windows"
"32Bit Windows"
"Java"
"Unknown"

struct ExtendedInfo_type {
    PString clientType;
}
```

SI_ERR_AUTH: Response when user was not authorized to request the information that they requested.

```
struct ExtendedInfo_errAuth {
}
```

SI_ERR_UNKN: Response when one of the flags in the request message was not one of the recognized flags.

```
struct ExtendedInfo_errUnknown {
    uint32 flags;
}
```

SI_INF_AURL: Response to request with the SI_AVATAR_URL flag bit set. Provides the default avatar URL from which the (Java) client can (and should) download avatar graphics.

```
struct ExtendedInfo_avatarURL {
    CString avatarURL;
}
```

SI_INF_VERS: Response to request with the SI_SERVER_VERSION flag bit set. Provides the server's current version ID string.

```
struct ExtendedInfo_serverVersion {
    CString versionString;
}
```

SI_INF_TYPE: Response to request with the SI_SERVER_TYPE flag bit set. Identifies the type of server. Currently there exist the following server type strings:

```
"Unix"
"Macintosh"
"Windows"
"Unknown"

struct ExtendedInfo_serverType {
```

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

```
        CString serverType;
    }
```

SI_INF_FLAG: Response to request with the SI_SERVER_FLAGS flag bit set. Provides various information about the server.

```
    struct ExtendedInfo_flags {
        uint32 flags;
        sint32 capacity;
        uint8  platform;
    }
```

`flags` are various flag bits describing the server configuration. Currently, these flags are defined:

```
    FF_DirectPlay           0x0001
    FF_ClosedServer         0x0002
    FF_GuestsAreMembers     0x0004
    FF_Unused1              0x0008
    FF_InstantPalace        0x0010
    FF_PalacePresents       0x0020
```

`capacity` is the maximum number of users the server can support.

`platform` encodes the server platform:

```
    PLAT_Macintosh     0
    PLAT_Windows95     1
    PLAT_WindwosNT     2
    PLAT_Unix          3
```

`SI_INF_NUM_USERS`: Response to request with the `SI_NUM_USERS` flag bit sit. Provides the number of users currently on the server.

```
    struct ExtendedInfo_numUsers {
        sint32 numUsers;
    }
```

`SI_INF_HURL`: Response to request with the `SI_HTTP_URL` flag bit set. Provides the HTTP URL for the client to obtain pictures and sounds instead of fetching them from the server using file transfers (`MSG_FILEQUERY` messages et al).

```
    struct ExtendedInfo_HTTPURL(string httpUrl) {
        CString(httpUrl);
    }
```

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

## 3.11MSG_FILENOTFND

This message is sent from the server to the client in response to a `MSG_FILEQUERY` request in which the requested file was unavailable. The message name implies “file not found”, but in reality this message is sent on failure regardless of the failure mode.

The `refnum` field is not used in this message and should be set to 0.

The `msg` field is a `PString` containing the name of the file that was not found or otherwise not obtainable:

```
struct ClientMsg_fileNotFnd {
    PString filename;
}
```

## 3.12MSG_FILEQUERY

This message is sent from the client to the server to request that the server send a file to the client. If the file can be delivered to the client, it will be sent in a `MSG_FILESEND` message; otherwise an error will be indicated via a `MSG_FILENOTFND` message.

The `refnum` field is not used in this message and should be set to 0.

The `msg` field is a `PString` containing the name of the file that is desired:

```
struct ClientMsg_fileQuery {
    PString filename;
}
```

## 3.13MSG_FILESEND

This message is sent from the server to the client in response to a `MSG_FILEQUERY` request, in order to deliver the file requested. The message format is designed to enable files to be transmitted in blocks, with each block sent in a separate message. Unlike the similar `MSG_ASSETSEND` message, this mechanism for sending files in pieces is actually implemented and supported.

The `refnum` field is not used in this message and should be set to 0.

The `msg` field is a `ClientMsg_fileSend` struct:

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

```
struct ClientMsg_fileSend {
    sint32 transactionID;
    sint32 blockSize;
    sint16 blockNbr;
    FileDescriptor desc;
    uint8  data[blockSize];
}
```

`transactionID` is simply a unique identifier, assigned by the server, that allows `MSG_FILESEND` messages for multiple files to be interleaved. All the messages associated with the response to a particular `MSG_FILEQUERY` request will share a common `transactionID`.

`blockSize` is the size of the file data block being sent in this message.

`blockNbr` is the block number of the block being sent in this message. Blocks are numbered starting with 0

`data` contains the actual bytes of the file block itself.

`desc` is present only in messages containing the first block of a file (i.e., when `blockNbr` is 0). It is a `FileDescriptor` struct, giving information about the file as a whole:

```
struct FileDescriptor {
    uint16 nbrBlocks;
    sint32 size;
    Str63  name;
}
```

`nbrBlocks` is the number of blocks in which the file will be transmitted.

`size` is the total size of the file, in bytes.

`name` is the name of the file.

## *3.14 MSG_GMSG*

The client sends this message to the server to utter a global world balloon, i.e., a speech utterance that is delivered to all the users on the server. A client sends the `MSG_GMSG` message to the server with the text to be spoken. If the server is happy with this (e.g., permission checks succeed -- this is a privileged operation), it sends corresponding `MSG_TALK` messages to all the users on the server.

The `refnum` field is not used in this message and should be set to 0.

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

The `msg` field is a `CString` containing the text of the utterance to be spoken:

```
struct ClientMsg_gMsg {
    CString text;
}
```

Although it is a `CString`, the text is limited to a maximum of 255 characters.

*(Note that the speaker can't be identified by the receiving client.)*

## 3.15MSG_HTTPSERVER

This message is sent from the server to the client to direct the client to use a particular HTTP server for obtaining images and sounds in preference to fetching them from the Palace server via file transfers.

The `refnum` field is the `UserID` of the user to whom the message is sent, or.

The `msg` field is the URL of the HTTP server which the client should use for HTTP requests:

```
struct ClientMsg_HTTPServer {
    CString url;
}
```

## 3.16MSG_KILLUSER

This message is sent from the client to the server, requesting the server to kill (i.e., forcibly logoff) a particular user. Obviously, this only works if the sender has sufficient authority.

The `refnum` field is not used in this message and should be set to 0.

The `msg` field identifies the user who is to be bumped off:

```
struct ClientMsg_killUser {
    UserID targetID;
}
```

## 3.17MSG_LISTOFALLROOMS

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

This message has two forms: a client-to-server (“request”) form, in which the client request the server to send it information about all the rooms on the server, and a server-to-client (“response”) form, in which the server delivers this information.

In the request form, the `refnum` field is not used and should be set to 0, and there are no parameters, so the `length` field should be 0 and the `msg` field should be empty:

```
struct ClientMsg_listOfAllRooms_request {
}
```

In the response from, the `refnum` field contains the number of rooms, and the `msg` field contains a sequence of `RoomListRec` structs, one for each room:

```
struct ClientMsg_listOfAllRooms_response {
    RoomListRec rooms[];
}
```

Each `RoomListRec` describes one room:

```
struct RoomListRec {
    sint32  roomID;
    sint16  flags;
    sint16  nbrUsers;
    PString name; /* padded to align length */
}
```

`roomID` is the ID number of the room. Note that even though a room ID is a 16-bit quantity, this message stores it in a 32-bit field.

`flags` are various flag bits providing information about the room:

| | | |
|---|---|---|
| `RF_AuthorLocked` | `0x0001` | non-owner can’t change (member created room) |
| `RF_Private` | `0x0002` | not in room list; in user list as “private” |
| `RF_NoPainting` | `0x0004` | disables drawing commands in room |
| `RF_Closed` | `0x0008` | no entry permitted (those in can stay) |
| `RF_CyborgFreeZone` | `0x0010` | client disables cyborg.ipt scripts in room |
| `RF_Hidden` | `0x0020` | doesn’t show up in goto list |
| `RF_NoGuests` | `0x0040` | guest users not permitted in room |
| `RF_WizardsOnly` | `0x0080` | only wizards permitted in room |
| `RF_DropZone` | `0x0100` | one of the rooms in which new users arrive |

`nbrUsers` is the number of users currently in the room.

`name` is the name of the room. If necessary, this `PString` will have extra padding bytes added onto the end to ensure that its total length (including the

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

length byte) is a multiple of

## 3.18MSG_LISTOFALLUSERS

This message has two forms: a client-to-server (“request”) form, in which the client request the server to send it information about all the users on the server, and a server-to-client (“response”) form, in which the server delivers this information.

In the request form, the `refnum` field is not used and should be set to 0, and there are no parameters, so the `length` field should be 0 and the `msg` field should be empty:

```
struct ClientMsg_listOfAllUsers_request {
}
```

In the response from, the `refnum` field contains the number of users, and the `msg` field contains a sequence of UserListRec structs, one for each user:

```
struct ClientMsg_listOfAllUsers_response {
    UserListRec users[];
}
```

Each `UserListRec` describes one user:

```
struct UserListRec {
    UserID userID;
    sint16 flags;
    RoomID roomID;
    PString name; /* padded to align length */
}
```

`userID` is the `UserID` of the user.

`flags` are various flags bits providing information about the user:

| | | |
|---|---|---|
| `U_SuperUser` | `0x0001` | wizard |
| `U_God` | `0x0002` | total wizard |
| `U_Kill` | `0x0004` | server should drop user at first opportunity |
| `U_Guest` | `0x0008` | user is a guest (i.e., no registration code) |
| `U_Banished` | `0x0010` | redundant with `U_Kill`, shouldn’t be used |
| `U_Penalized` | `0x0020` | historical artifact, shouldn’t be used |
| `U_CommError` | `0x0040` | comm error, drop at first opportunity |
| `U_Gag` | `0x0080` | not allowed to speak |
| `U_Pin` | `0x0100` | stuck in corner and not allowed to move |

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

| | | |
|---|---|---|
| U_Hide | 0x0200 | doesn't appear on user list |
| U_RejectESP | 0x0400 | not accepting whisper from outside room |
| U_RejectPrivate | 0x0800 | not accepting whisper from inside room |
| U_PropGag | 0x1000 | not allowed to have props |

roomID is the ID number of the room the user is currently in.

name is the user's name. If necessary, this PString will have extra padding bytes added onto the end to ensure that its total length (including the length byte) is a multiple of

## 3.19MSG_LOGOFF

This message has two forms: a client-to-server ("request") form, in which the client requests the server to log it off, and a server-to-client ("notify") form, in which the server informs (other) clients that the user has logged off.

In the request form, the refnum field is not used and should be set to 0, and there are no parameters, so the length field should be 0 and the msg field should be empty:

```
struct ClientMsg_logOff_request {
}
```

In the notify form, the refnum field contains the UserID of the user who logged off and the msg field indicates the revised number of users on the server:

```
struct ClientMsg_logoff_notify {
    sint32 nbrUsers;
}
```

Normally, this message is sent to everyone who was in the same room with the user who just logged off. However, it will be sent to every user on the server if the total number of users is small enough or if a long enough time has passed since the last such global notification.

## 3.20MSG_LOGON

The client sends this message to the server to initiate a session.

The refnum field is not used in this message and should be set to 0.

The msg field is an AuxRegistrationRec:

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

```
struct ClientMsg_logon {
    AuxRegistrationRec rec;
}
```

An `AuxRegistrationRec` contains lots of information about the user:

```
struct AuxRegistrationRec {
    uint32 crc;
    uint32 counter;
    Str31  userName;
    Str31  wizPassword;
    sint32 auxFlags;
    uint32 puidCtr;
    uint32 puidCRC;
    uint32 demoElapsed;
    uint32 totalElapsed;
    uint32 demoLimit;
    sint16 desiredRoom;
    char   reserved[6];
    uint32 ulRequestedProtocolVersion;
    uint32 ulUploadCaps;
    uint32 ulDownloadCaps;
    uint32 ul2DEngineCaps;
    uint32 ul2DGraphicsCaps;
    uint32 ul3DEngineCaps;
}
```

`crc` and `counter` together form the registration code. `counter` is the registration number; `crc` is a hash that should validate it. `puidCtr` and `puidCRC` are a client generated pseudo-random user ID. Once again, `puidCtr` is the ID number and `puidCRC` is a validation hash.

`userName` is the user's name string.

`wizPassword` is the user's wizard password, if the user is (or is claiming to be) a wizard.

`auxFlags` indicate various attributes of the user's machine:

```
LI_AUXFLAGS_UnknownMach    0
LI_AUXFLAGS_Mac68k         1
LI_AUXFLAGS_MacPPC         2
LI_AUXFLAGS_Win16          3
LI_AUXFLAGS_Win32          4
LI_AUXFLAGS_Java           5
LI_AUXFLAGS_OSMask         0x0000000F
LI_AUXFLAGS_Authenticate   0x80000000
```

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

`demoElapsed`, `totalElapsed` and `demoLimit` are historical artifacts. They once had something to do with restricting trial use of the client, but are not longer used (and would be trivially spoofable if still used).

`desiredRoom` indicates the room ID of the room the user would like to enter initially.

`reserved` are simply empty bytes that should not have anything put in them.

`ulRequestedProtocolVersion` is supposed to encode the version number of the protocol that the client is running, with the major version number in the high 16 bits and the minor version number in the low 16 bits. However, the server ignores this.

`ulUploadCaps` indicates the client's capabilities with respect to uploading assets and files:

```
LI_ULCAPS_ASSETS_PALACE     0x00000001
LI_ULCAPS_ASSETS_FTP        0x00000002
LI_ULCAPS_ASSETS_HTTP       0x00000004
LI_ULCAPS_ASSETS_OTHER      0x00000008
LI_ULCAPS_FILES_PALACE      0x00000010
LI_ULCAPS_FILES_FTP         0x00000020
LI_ULCAPS_FILES_HTTP        0x00000040
LI_ULCAPS_FILES_OTHER       0x00000080
LI_ULCAPS_EXTEND_PKT        0x00000100
```

However, it is completely unused on the server.

`ulDownloadCaps` indicates the client's capabilities with respect to downloading assets and files:

```
LI_DLCAPS_ASSETS_PALACE     0x00000001
LI_DLCAPS_ASSETS_FTP        0x00000002
LI_DLCAPS_ASSETS_HTTP       0x00000004
LI_DLCAPS_ASSETS_OTHER      0x00000008
LI_DLCAPS_FILES_PALACE      0x00000010
LI_DLCAPS_FILES_FTP         0x00000020
LI_DLCAPS_FILES_HTTP        0x00000040
LI_DLCAPS_FILES_OTHER       0x00000080
LI_DLCAPS_FILES_HTTPSrvr    0x00000100
LI_DLCAPS_EXTEND_PKT        0x00000200
```

The only bit which the Unix server examines is `LI_DLCAPS_FILES_HTTPSrvr`.

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

`ul2DEngineCaps` allegedly indicates the client's 2-D display engine:

```
LI_2DENGINECAP_PALACE        0x00000001
LI_2DENGINECAP_DOUBLEBYTE    0x00000002
```

However, it (and the flag bit values) are completely unused on the server

`ul2DGraphicsCaps` allegedly indicates the client's 2-D graphics capabilities:

```
LI_2DGRAPHCAP_GIF87          0x00000001
LI_2DGRAPHCAP_GIF89a         0x00000002
LI_2DGRAPHCAP_JPG            0x00000004
LI_2DGRAPHCAP_TIFF           0x00000008
LI_2DGRAPHCAP_TARGA          0x00000010
LI_2DGRAPHCAP_BMP            0x00000020
LI_2DGRAPHCAP_PCT            0x00000040
```

However, it (and the flag bit values) are completely unused on the server.

`ul3DEngineCaps` allegedly indicates the client's 3-D graphics capabilities:

```
LI_3DENGINECAP_VRML1         0x00000001
LI_3DENGINECAP_VRML2         0x00000002
```

However, it (and the flag bit values) are completely unused on the server.

## *3.21MSG_NAVERROR*

The server sends this message to the client to inform the client about a failure in moving the user's avatar from one room to another.

The `refnum` field contains an error code indicating the nature of the problem:

```
SE_InternalError  0
SE_RoomUnknown    1
SE_RoomFull       2
SE_RoomClosed     3
SE_CantAuthor     4
SE_PalaceFull     5
```

There are no other parameters in this message, so the `length` field should be 0 and the `msg` field should be empty:

```
struct ClientMsg_navError {
```

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

```
}
```

## 3.22MSG_NOOP

This message carries no information and has no effect. Presumably it is used for testing network communications, but it is not actually used in the current Unix server.

The `refnum` field is not used in this message and should be set to 0.

There are no parameters in this message, so the `length` field should be 0 and the `msg` field should be empty:

```
struct ClientMsg_noOp {
}
```

## 3.23MSG_PICTMOVE

This message is used to modify the screen location of a picture. A client sends a `MSG_PICTMOVE` message to the server requesting a picture to be moved. If the operation is successful (for example, all the permission checks succeed), the server sends a matching `MSG_PICTMOVE` message to the clients in the room with the picture, informing them of the event. In both directions the message is the same.

The `refnum` field is not used in this message and should be set to 0.

The `msg` field is a `ClientMsg_pictMove` struct:

```
struct ClientMsg_pictMove {
    RoomID roomID;
    HotspotID spotID;
    Point  pos;
}
```

`roomID` is the `RoomID` of the room containing the picture to be moved.

`spotID` is the `HotspotID` of the picture itself, in that room.

`pos` is the new position to which the picture is to be relocated.

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

## 3.24MSG_PING

This message enables one machine to ping another to see if the connection is still alive. It is also sent periodically by the server to each client in the absence of any other message traffic, in order to hold the connection open. The proper behavior in response to a `MSG_PING` message is a `MSG_PONG` message.

The `refnum` field can carry an arbitrary value. It is not used by the server directly but will be echoed in the corresponding `MSG_PONG` message.

There are no other parameters in this message, so the `length` field should be 0 and the `msg` field should be empty:

```
struct ClientMsg_ping {
}
```

## 3.25MSG_PONG

This message is the normal response to a `MSG_PING` message. On the server it functions as a no-op, but has the side effect of keeping the client-server communications channel active.

The `refnum` field should echo the contents of the `refnum` field of the `MSG_PING` message to which this is a response.

There are no other parameters in this message, so the `length` field should be 0 and the `msg` field should be empty:

```
struct ClientMsg_pong {
}
```

## 3.26MSG_PROPDEL

This message is used to delete a prop from the palace. A client sends a `MSG_PROPDEL` message to the server requesting a prop to be deleted. If the operation is successful (for example, all the permission checks succeed), the server sends a matching `MSG_PROPDEL` message to the clients in the room with the prop, informing them of the event. In both directions the message is the same.

The `refnum` field is not used in this message and should be set to 0.

The `msg` field is a `ClientMsg_propDel` struct:

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

```
struct ClientMsg_propDel {
    sint32 propNum;
}
```

`propNum` identifies the prop to be deleted (in the same room with the user whose client initiated the operation -- props in a room are numbered from 0 in the order they are added to room). A `propNum` value of -1 indicates that all props in the room should be deleted. Note that although the value is transmitted in the message as 32-bit quantity, the actual number is limited to a 16-bit quantity.

## 3.27MSG_PROPMOVE

This message is used to change the screen location of a prop. A client sends a `MSG_PROPMOVE` message to the server requesting a prop to be moved. If the operation is successful (for example, all the permissions check succeed), the server sends a matching `MSG_PROPMOVE` message to the clients in the room with the prop, informing them of the event. In both directions the message is the same.

The `refnum` field is not used in this message and should be set to 0.

The `msg` field is a `ClientMsg_propMove` struct:

```
struct ClientMsg_propMove {
    sint32 propNum;
    Point  pos;
}
```

`propNum` identifies the prop to be moved (in the same room with the user whose client initiated the operation -- props in a room are numbered from 0 in the order they are added to room). Note that although the value is transmitted in the message as 32-bit quantity, the actual number is limited to a 16-bit quantity.

`pos` is the new screen position for the prop.

## 3.28MSG_PROPNEW

This message is used to add a prop to a room. A client sends a `MSG_PROPNEW` message to the server requesting that a prop be added. If the operation is successful (for example, all the permissions check succeed), the server sends a matching `MSG_PROPNEW` message to the clients in the room, informing them of the event. In both directions the message is the same.

The `refnum` field is not used in this message and should be set to 0.

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

The msg field is a ClientMsg_propNew struct:

```
struct ClientMsg_propNew {
    AssetSpec propSpec;
    Point  pos;
}
```

propSpec is an AssetSpec identifying the asset to be used for the prop (the asset type is assumed to be RT_PROP).

pos is the initial screen position for the new prop.

## 3.29MSG_RMSG

This message is essentially identical to the MSG_TALK message. It is sent from the client to the server, and relayed to the other users in the sender's room as a MSG_TALK message. However, it also sends an additional, special MSG_TALK message to any superusers (wizards and gods) in the room, flagging the utterance for special attention.

The refnum field is not used in this message and should be set to 0.

The msg field is a CString containing the text of the utterance to be spoken:

```
struct ClientMsg_rMsg {
    CString text;
}
```

Although it is a CString, the text is limited to a maximum of 255 characters.

## 3.30MSG_ROOMDESC

The server sends this message to the client to describe a room upon entry to that room.

The refnum field is not used in this message and should be set to 0.

The msg field is a RoomRec struct:

```
struct ClientMsg_roomDesc {
    RoomRec rec;
```

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

```
    }
```

A `RoomRec` contains lots of information about a room:

```
struct RoomRec {
    sint32 roomFlags;
    sint32 facesID;
    sint16 roomID;
    sint16 roomNameOfst;
    sint16 pictNameOfst;
    sint16 artistNameOfst;
    sint16 passwordOfst;
    sint16 nbrHotspots;
    sint16 hotspotOfst;
    sint16 nbrPictures;
    sint16 pictureOfst;
    sint16 nbrDrawCmds;
    sint16 firstDrawCmd;
    sint16 nbrPeople;
    sint16 nbrLProps;
    sint16 firstLProp;
    sint16 reserved;
    sint16 lenVars;
    uint8  varBuf[lenVars];
}
```

`roomFlags` are various bit flags describing attributes of the room. See the description of the `RoomListRec` struct (in **§3.17. `MSG_LISTOFALLROOMS`**) for descriptions of these flags.

`facesID` selects one of a preset number of client-defined avatar appearances that should be displayed for the avatar when it is not showing a prop instead.

`roomID` is the ID number of the room on the server.

`roomNameOfst` is the index into `varBuf` of a `PString` that is the name of the room.

`pictNameOfst` is the index into `varBuf` of a `PString` that is the filename of a picture to use as the room background.

`artistNameOfst` is the index into `varBuf` of a `PString` that is the name of the artist who created the room.

`passwordOfst` is the index into `varBuf` of a `PString` that is the password for the room -- member-created rooms may have passwords for controlling entry

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

to the room.

`nbrHotspots` is the number of hotspots in the room.

`hotspotOfst` is the index into `varBuf` of the beginning of an array of `nbrHotspots` Hotspot structs (described below) that describe the hotspots in the room. Note that this array must be aligned on a 4-byte boundary.

`nbrPictures` is the number of pictures in the room

`pictureOfst` is the index into `varBuf` of the beginning of an array of `nbrPictures` `PictureRec` structs (described below) that describe the pictures in the room. Note that this array must be aligned on a 4-byte boundary.

`nbrDrawCmds` is the number of draw commands in the room's display list.

`firstDrawCmd` is the index into `varBuf` of the first of a packed sequence of `nbrDrawCmds` draw commands (`DrawRecord` structs and attached data, see **§3.9. `MSG_DRAW`**). Note that these records must be aligned on 4-byte boundaries.

`nbrPeople` is the number of users currently in the room.

`nbrLProps` is the number of props in the room.

`firstLProp` is the index into `varBuf` of the beginning of an array of `nbrLProps` `LPropRec` structs (described below) that describe the props in the room. Note that this array must be aligned on a 4-byte boundary.

`reserved` is a filler block to maintain field alignment. It should be set to 0 .

`varBuf` is an array of `lenVars` bytes of variable-length data associated with the `RoomRec`, as described above

The `Hotspot` struct describes a hotspot, a clickable piece of screen real estate with a script that runs in response to various events:

```
struct Hotspot {
    sint32 scriptEventMask;
    sint32 flags;
    sint32 secureInfo;
    sint32 refCon;
    Point  loc;
    sint16 id;
    sint16 dest;
    sint16 nbrPts;
    sint16 ptsOfst;
    sint16 type;
```

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

```
        sint16 groupID;
        sint16 nbrScripts;
        sint16 scriptRecOfst;
        sint16 state;
        sint16 nbrStates;
        sint16 stateRecOfst;
        sint16 nameOfst;
        sint16 scriptTextOfst;
        sint16 alignReserved;
    }
```

`scriptEventMask` is a set of bit flags that encode what events this hotspot responds to:

```
PE_Select      0x00000001
PE_Lock        0x00000002
PE_Unlock      0x00000004
PE_Hide        0x00000008
PE_Show        0x00000010
PE_Startup     0x00000020
PE_Alarm       0x00000040
PE_Custom      0x00000080
PE_InChat      0x00000100
PE_PropChange  0x00000200
PE_Enter       0x00000400
PE_Leave       0x00000800
PE_OutChat     0x00001000
PE_SignOn      0x00002000
PE_SignOff     0x00004000
PE_Macro0      0x00008000
PE_Macro1      0x00010000
PE_Macro2      0x00020000
PE_Macro3      0x00040000
PE_Macro4      0x00080000
PE_Macro5      0x00100000
PE_Macro6      0x00200000
PE_Macro7      0x00400000
PE_Macro8      0x00800000
PE_Macro9      0x01000000
```

`flags` are various flag bits characterizing the hotspot *<these are unused on the server>*.

`secureInfo` is a variable whose purpose in unclear. *<It is not used.>*

`refCon` is an arbitrary use variable. *<It is not used*

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

`loc` is the location of the hotspot

`id` is the hotspot's ID number.

`dest` is the `RoomID` of the destination when the hotspot is a door, or the `HotspotID` of a door when the hotspot is a bolt.

`nbrPts` is the number of `Points` which describe the outline of the hotspot's click zone.

`ptsOfst` is the index into the `RoomRec`'s `varBuf` array of an array of `nbrPts` `Points` describing the outline of the hotspot's click zone.

`type` encodes the type of hotspot, for navigation purposes:

| | | |
|---|---|---|
| `HS_Normal` | `0` | just a script holder |
| `HS_Door` | `1` | a door |
| `HS_ShutableDoor` | `2` | a door that can be opened/closed (by clicking) |
| `HS_LockableDoor` | `3` | a door that can be locked |
| `HS_Bolt` | `4` | bolt that locks or unlocks door pointed to by `dest` |
| `HS_NavArea` | `5` | hotspot defines screen area where movement is allowed |

`groupID` is some kind of group identification number. *<It is not used.>*

`nbrScripts` is the number of scripts associated with the hotspot.

`scriptRecOfst` is the index into the `RoomRec`'s `varBuf` array of an array of `nbrScripts` structs that describe the hotspot's scripts.

`state` encodes the hotspot's state. The state selects which of the pictures associated with the hotspot should be displayed. Among other things, it encodes whether a door is locked or unlocked:

```
HS_Unlock  0
HS_Lock    1
```

`nbrStates` is the number of states (with pictures and corresponding `StateRec` structs) associated with the hotspot.

`stateRecOfst` is the index into the `RoomRec`'s `varBuf` array of an array of `nbrStates` `StateRec` structs (described below). There is one entry for each of the hotspot's state pictures.

`nameOfst` is the index into the `RoomRec`'s `varBuf` array of a `PString` that is the hotspot's name.

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

`scriptTextOfst` is the index into the `RoomRec`'s `varBar` array of a `CString` that is the script associated with this hotspot.

`alignReserved` is a filler block to maintain field alignment. It should be set to 0 *<and oughtn't be sent over the wire>*.

The `StateRec` struct describes one of the pictures associated with a hotspot; an array of them is pointed to by the hotspot's `stateRecOfst` field:

```
struct StateRec {
    sint16 pictID;
    sint16 reserved;
    Point  picLoc; /* offset from hotspot->loc */
}
```

`pictID` is a picture ID number.

`reserved` is presumably a filler block for alignment.

`picLoc` is a screen location, interpreted as an offset value from the `Point` in the hotspot's `loc` field.

The `PictureRec` struct describes a picture:

```
struct PictureRec {
    sint32 refCon;
    sint16 picID;
    sint16 picNameOfst;
    sint16 transColor;
    sint16 reserved;
}
```

`refCon` is an arbitrary use variable. *<It is not used*

`picID` is the ID number for the picture.

`picNameOfst` is the index into the `RoomRec`'s `varBuf` array of a `PString` that is the name of the picture.

`transColor` is the color value that should be interpreted as transparent in the picture.

`reserved` is a filler block to maintain field alignment. It should be set to 0 *<and oughtn't be sent over the wire>*.

The `LPropRec` struct describes a prop:

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

```
struct LPropRec {
    LLRec     link;
    AssetSpec propSpec;
    sint32    flags;
    sint32    refCon;
    Point     loc;
}
```

`link` is an `LLRec` struct, as described above. It is used internally and has no purpose when sent over wire

`propSpec` identifies the asset that should be used to represent the prop on the client's screen.

`flags` are various flag bits characterizing the prop, used on the client.

`refCon` is an arbitrary use variable, used by the client.

`loc` is the screen location at which the prop should be displayed.

## 3.31MSG_ROOMDESCEND

The server sends this message to the client when it has completed transmission of a room to the client (that is, after it has send the `MSG_ROOMDESC` and `MSG_USERLIST` messages that describe the room).

The `refnum` field is not used in this message and should be set to 0.

There are no parameters in this message, so the `length` field should be 0 and the `msg` field should be empty:

```
struct ClientMsg_roomDescEnd {
}
```

## 3.32MSG_ROOMGOTO

The client sends this message to the server to request that the user move to a different room. If the operation is successful (for example, all the permission checks succeed), the server sends the following:

a `MSG_USEREXIT` message to the other users in the old room informing them of the departure

a `MSG_USERNEW` message to the other users in the new room informing them of the arrival

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

a series of messages to the requestor client describing the new room (MSG_ROOMDESC, then MSG_USERLIST, then MSG_ROOMDESCEND, then possibly one or more MSG_DISPLAURL messages).

If the operation fails, the server responds with a MSG_NAVERROR message.

The refnum field is not used in this message and should be set to 0.

The msg field identifies the desired destination room:

```
struct ClientMsg_roomGoto {
    RoomID dest;
}
```

## 3.33MSG_ROOMNEW

The client sends this message to the server to request that the server create a new (empty) room. If the operation is successful (for example, all the permission checks succeed), the user is moved to the new room as if the client had issued a MSG_ROOMGOTO request. If the operation fails, the server responds with a MSG_NAVERROR message.

The new room will have a standard default arrangement, whose details are outside the scope of this protocol.

The refnum field is not used in this message and should be set to 0.

There are no parameters in this message, so the length field should be 0 and the msg field should be empty:

```
struct ClientMsg_roomNew {
}
```

## 3.34MSG_ROOMSETDESC

This message is used to transmit a revised description of a room. A client sends a MSG_ROOMSETDESC message to the server to request that the room description be replaced. If the operation is successful, the server sends a matching MSG_ROOMSETDESC message to the clients in the room. The server also sends MSG_ROOMSETDESC messages of its own in response to other events which revise or replace the description of a room. In both directions the message format is the same.

The refnum field is not used in this message and should be set to 0.

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

The `msg` field contains a `RoomRec` struct:

```
struct ClientMsg_roomSetDesc {
    RoomRec rec;
}
```

The RoomRec struct is described above in **§3.31 `MSG_ROOMDESC`**.

## *3.35MSG_SERVERDOWN*

The server sends this message to the client to inform the client that the server is dropping the connection. From the message name you would think this is used to inform the user that the server is going down, but it is actually used when users are kicked of the system for some reason.

The `refnum` field contains an encoding of the reason the connection is being dropped:

```
K_Unknown              0
K_LoggedOff            1
K_CommError            2
K_Flood                3
K_KilledByPlayer       4
K_ServerDown           5
K_Unresponsive         6
K_KilledBySysop        7
K_ServerFull           8
K_InvalidSerialNumber  9
K_DuplicateUser       10
K_DeathPenaltyActive  11
K_Banished            12
K_BanishKill          13
K_NoGuests            14
K_DemoExpired         15
K_Verbose             16
```

If the `refnum` value is `K_Verbose`, the `msg` field contains a `CString` with a textual message. Otherwise, the `length` field should be 0 and the `msg` field should be empty:

```
struct ClientMsg_serverDown {
    CString whyMessage;
}
```

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

## 3.36MSG_SERVERINFO

The server sends this message to the client to describe various characteristics of the server to it. It is sent as part of the series of messages that the server sends during logon. It can also be sent as a result of an operator directive (god command).

The `refnum` field is the `UserID` of the user to whom the message is sent.

The `msg` field is a `ClientMsg_serverInfo` struct:

```
struct ClientMsg_serverInfo {
    sint32 serverPermissions;
    Str63  serverName;
    uint32 serverOptions;
    uint32 ulUploadCaps;
    uint32 ulDownloadCaps;
}
```

`serverPermissions` are various bits describing the "rules" that the server currently has in force:

| | | |
|---|---|---|
| `PM_AllowGuests` | `0x0001` | guests may use this server |
| `PM_AllowCyborgs` | `0x0002` | clients can use cyborg.ipt scripts |
| `PM_AllowPainting` | `0x0004` | clients may issue draw commands |
| `PM_AllowCustomProps` | `0x0008` | clients may select custom props |
| `PM_AllowWizards` | `0x0010` | wizards can use this server |
| `PM_WizardsMayKill` | `0x0020` | wizards can kick off users |
| `PM_WizardsMayAuthor` | `0x0040` | wizards can create rooms |
| `PM_PlayersMayKill` | `0x0080` | normal users can kick each other off |
| `PM_CyborgsMayKill` | `0x0100` | scripts can kick off users |
| `PM_DeathPenalty` | `0x0200` | |
| `PM_PurgeInactiveProps` | `0x0400` | server discards unused props |
| `PM_KillFlooders` | `0x0800` | users dropped if they do too much too fast |
| `PM_NoSpoofing` | `0x1000` | command to speak as another is disabled |
| `PM_MemberCreatedRooms` | `0x2000` | users can create rooms |

`serverName` is the name of the server.

`serverOptions` are various bits describing the configuration of the server:

| | | |
|---|---|---|
| `SO_SaveSessionKeys` | `0x00000001` | server logs regcodes of users (obsolete) |
| `SO_PasswordSecurity` | `0x00000002` | you need a password to use this server |
| `SO_ChatLog` | `0x00000004` | server logs all chat |
| `SO_NoWhisper` | `0x00000008` | whisper command disabled |
| `SO_AllowDemoMembers` | `0x00000010` | obsolete |

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

```
SO_Authenticate     0x00000020
SO_PoundProtect     0x00000040   server employs heuristics to evade hackers
SO_SortOptions      0x00000080
SO_AuthTrackLogoff  0x00000100   server logs logoffs
SO_JavaSecure       0x00000200   server supports Java client's auth. scheme
```

`ulUploadCaps` indicates the server's capabilities with respect to uploading assets and files. See the description of the identically named field of the `AuxRegistrationRec` struct above in **§4.20. `MSG_LOGON`** for information on the values that go in this field.

`ulDownloadCaps` indicates the server's capabilities with respect to downloading assets and files. See the description of the identically named field of the `AuxRegistrationRec` struct above in **§4.20. `MSG_LOGON`** for information on the values that go in this field.

## *3.37MSG_SMSG*

The client sends this message to communicate with the superuser(s) (gods and wizards), i.e., a speech utterance is delivered to all the superusers on the system, together with the identity and room name of the sender. The server relays the text to the superusers using `MSG_TALK` messages.

The `refnum` field is not used in this message and should be set to 0.

The `msg` field is a `CString` containing the text of the utterance to be spoken:

```
struct ClientMsg_sMsg {
    CString text;
}
```

Although it is a `CString`, the text is limited to a maximum of 255 characters. Moreover, the superusers will only see the first 168 characters due to the space needed for identification text which the server inserts.

## *3.38MSG_SPOTDEL*

The client sends this message to the server to request that a hotspot be deleted from the current room. If the operation is successful (for example, all the permission checks succeed), the server replaces the room with a new room that lacks the indicated hotspot.

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

The `refnum` field is not used in this message and should be set to 0.

The `msg` field identifies hotspot whose demise is desired:

```
struct ClientMsg_spotDel {
    HotspotID spotID;
}
```

## 3.39MSG_SPOTMOVE

This message is used to modify the screen location of a hotspot. A client sends a `MSG_SPOTMOVE` message to the server requesting a hotspot be moved. If the operation is successful (for example, all the permission checks succeed), the server sends a matching `MSG_SPOTMOVE` message to the clients in the room with the hotspot, informing them of the event. In both directions the message is the same.

The `refnum` field is not used in this message and should be set to 0.

The `msg` field is a `ClientMsg_spotMove` struct:

```
struct ClientMsg_spotMove {
    RoomID    roomID;
    HotspotID spotID;
    Point     pos;
}
```

`roomID` is the `RoomID` of the room containing the hotspot to be moved.

`spotID` is the `HotspotID` of the hotspot itself, in that room.

`pos` is the new position to which the hotspot is to be relocated.

## 3.40MSG_SPOTNEW

The client sends this message to the server to request that a hotspot be created in the current room. If the operation is successful (for example, all the permission checks succeed), the server replaces the room with a new room that contains the new hotspot.

The new hotspot will have a standard, default configuration, whose details are outside the scope of this protocol.

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

The `refnum` field is not used in this message and should be set to 0.

There are no parameters in this message, so the `length` field should be 0 and the `msg` field should be empty:

```
struct ClientMsg_spotNew {
}
```

## 3.41MSG_SPOTSTATE

This message is used to modify the `state` field of a hotspot. A client sends a `MSG_SPOTSTATE` message to the server requesting the change. If the oepration is successful, the server sends a matching `MSG_SPOTSTATE` message to the clients in the room with the hotspot, informing them of the event. In both directions the message is the same.

The `refnum` field is not used in this message and should be set to 0.

The `msg` field is a `ClientMsg_spotState` struct:

```
struct ClientMsg_spotState {
    RoomID    roomID;
    HotspotID spotID;
    sint16    state;
}
```

`roomID` is the `RoomID` of the room containing the hotspot whose state is to be changed.

`spotID` is the `HotspotID` of the hotspot itself, in that room.

`state` is the new value for the hotspot's `state` field.

## 3.42MSG_SUPERUSER

The client sends this message to the server in order to enter superuser (wizard or god) mode, providing the password to do so. If the password matches (and the server distinguishes between wizard and god mode by which password it matches), the server notifies the client of success with a `MSG_USERSTATUS` message.

The `refnum` field is not used in this message and should be set to 0.

The `msg` field is a `PString` containing the password for the superuser mode

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

desired:

```
struct ClientMsg_superuser {
    PString password;
}
```

## 3.43MSG_TALK

This message is used to utter word balloon speech in the Palace. A client sends a `MSG_TALK` message to the server with text to be spoken. If the server is happy with the text, it sends a matching `MSG_TALK` message to the clients in the room with the speaker. In both directions the message format is the same. The `MSG_TALK` message is also used by the server to issue spoken text of a variety of different kinds in a variety of different circumstances.

The `refnum` field can carry an arbitrary value. It is not used by the server directly but will be echoed in the relayed `MSG_TALK` message. `MSG_TALK` messages generated by the server endogenously will set this field to 0. Clients which receive `MSG_TALK` message interpret the `refnum` field as the `UserID` of the person who is talking.

The `msg` field is a `CString` containing the text of the utterance to be spoken:

```
struct ClientMsg_talk {
    CString text;
}
```

Although it is a `CString`, the text is limited to a maximum of 255 characters.

## 3.44MSG_TIYID

When a client connects to the server, the first thing the server does is send a `MSG_TIYID` message. The message informs the client as to its `UserID` (“TIYID” stands for “This Is Your ID”). The byte ordering of the `eventType` field also informs the client as to the server’s native endianess, so that it can adapt appropriately -- the client is responsible for all the work of coping with client/server endianess mismatch.

The `refnum` field contains the `UserID` that the client should use.

There are no other parameters in this message, so the `length` field should be 0 and the `msg` field should be empty:

```
struct ClientMsg_tiyid {
```

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

```
}
```

## 3.45MSG_USERCOLOR

This message is used to change the color of a user's avatar. A client sends a `MSG_USERCOLOR` message to the server requesting the change. If the operation is successful, the server sends a matching `MSG_USERCOLOR` message to the other clients in the room, informing them of the event. In both directions the message format is the same.

The `refnum` field contains the `UserID` of the user whose color is being changed (this is ignored by the server but meaningful when received by the client).

The `msg` field indicates what the new color is supposed to be:

```
struct ClientMsg_userColor {
    sint16 colorNbr;
}
```

`colorNbr` is a number in the range 0-15.

## 3.46MSG_USERDESC

This message is used to change a number of aspects of a user's appearance all at once. A client sends a `MSG_USERDESC` message to the server requesting the changes. If the operation is successful, the server sends a matching `MSG_USERDESC` message to the other clients in the room, informing them of the event. In both directions the message format is the same.

The `refnum` field contains the `UserID` of the user whose appearance is being changed (this is ignored by the server but meaningful when received by the client).

The `msg` field is a `ClientMsg_userDesc` struct:

```
struct ClientMsg_userDesc {
    sint16 faceNbr;
    sint16 colorNbr;
    sint32 nbrProps;
    AssetSpec props[nbrProps];
}
```

`faceNbr` is a number in the range 0-15 that indicates which face the user wishes to display (see **3.48 MSG_USERFACE**).

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

`colorNbr` is a number in the range 0-15 that indicates the user's color (see **`MSG_USERCOLOR`**).

`nbrProps` is a number in the range 0-9 that indicates how many props the user has.

`props` is an array of `nbrProps` `AssetSpecs` which indicates the assets (`RT_PROP`s, in this case) to use for the props.

## 3.47 MSG_USEREXIT

This message is sent from the server to the client to inform the client that a (different) user has left the room that the client is in.

The `refnum` field contains the `UserID` of the user who left the room.

There are no other parameters in this message, so the `length` field should be 0 and the `msg` field should be empty:

```
struct ClientMsg_userExit {
}
```

## 3.48 MSG_USERFACE

This message is used to change the face that a user's avatar is displaying. A client sends a `MSG_USERFACE` message to the server requesting the change. If the operation is successful, the server sends a matching `MSG_USERFACE` message to the other clients in the room, informing them of the event. In both directions the message format is the same.

The `refnum` field contains the `UserID` of the user whose face is being changed (this is ignored by the server but meaningful when received by the client).

The `msg` field indicates what the new face is supposed to be:

```
struct ClientMsg_userFace {
    sint16 faceNbr;
}
```

`faceNbr` is a number in the range 0-15 selecting the face that is desired.

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

## 3.49MSG_USERLIST

This message is sent from the server to the client as part of the process of entry to a room. It informs the client about the users in a room.

The `refnum` fields contains the number of users in the room.

The `msg` field contains an array of `UserRec` structs, one for each user:

```
struct ClientMsg_userList {
    UserRec users[];
}
```

The `UserRec` struct describes a user in the room:

```
struct UserRec {
    UserID userID;
    Point  roomPos;
    AssetSpec propSpec[9];
    RoomID roomID;
    sint16 faceNbr;
    sint16 colorNbr;
    sint16 awayFlag;
    sint16 openToMsgs;
    sint16 nbrProps;
    Str31  name;
}
```

`userID` is the user's `UserID`.

`roomPos` is the screen location of the user.

`propSpec` is an array of `AssetSpecs` pointing to the props that the user holds *<note that the maximal-length array is sent, even though it needn't be>*.

`roomID` is the `RoomID` of the room

`faceNbr` is a number in the range 0-15 that indicates the user's face.

`colorNbr` is a number in the range 0-15 that indicates the user's color.

`awayFlag` is a flag that is not used. *<not used>*

`openToMsgs` is another flag that is not used.

`nbrProps` is the number of props that the user holds (i.e., the length of the part

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

of `propSpec` that is actually used).

`name` is the user's name.

## 3.50MSG_USERLOG

This message is sent from the server to the client to notify the client that a new user has logged onto the server.

The `refnum` field contains the `UserID` of the user who logged on.

The `msg` field indicates the revised number of users on the server:

```
struct ClientMsg_userLog {
    sint32 nbrUsers;
}
```

Normally, this message is sent to everyone in the room with the new user (including the new user). However, it will be sent to every user on the server if the total number of users is small enough or if a long enough time has passed since the last such global notification.

## 3.51MSG_USERMOVE

This message is used to screen location of a user's avatar. A client sends a `MSG_USERMOVE` message to the server requesting the change. If the operation is successful, the server sends a matching `MSG_USERMOVE` message to the other clients in the room, informing them of the event. In both directions the message format is the same.

The `refnum` field contains the `UserID` of the user who is moving (this is ignored by the server but meaningful when received by the client).

The `msg` field indicates what the new position is supposed to be:

```
struct ClientMsg_userMove {
    Point pos;
}
```

## 3.52MSG_USERNAME

This message is used to change the a user's name. A client sends a

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

MSG_USERNAME message to the server requesting the change. If the operation is successful, the server sends a matching MSG_USERNAME message to the other clients in the room, informing them of the event. If the operation fails, the server sends a MSG_USERNAME back to the requesting client, instructing it to set the user name back to its previous value. In all cases the message format is the same.

The refnum field contains the UserID of the user whose face is being changed (this is ignored by the server but meaningful when received by the client).

The msg field contains a PString with the new name:

```
struct ClientMsg_userName {
    PString name;
}
```

## 3.53MSG_USERNEW

This message is sent from the server to the client to describe a new user who has entered the room.

The refnum field contains the UserID of the new user.

The msg field contains a UserRec struct describing the new user:

```
struct ClientMsg_userNew {
    UserRec newUser;
}
```

See **MSG_USERLIST** for a detailed description of the UserRec struct.

## 3.54MSG_USERPROP

This message is used to change a user's props. A client sends a MSG_USERPROP message to the server requesting the change. If the operation is successful, the server sends a matching MSG_USERPROP message to the other clients in the room, informing them of the event. In both directions the message format is the same.

The refnum field contains the UserID of the user whose props are being changed (this is ignored by the server but meaningful when received by the client).

The msg field indicates what the new props are supposed to be:

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

```
struct ClientMsg_userProp {
    sint32 nbrProps;
    AssetSpec props[nbrProps];
}
```

`nbrProps` is a number in the range 0-9 that indicates how many props the user now has.

`props` is an array of `nbrProps` `AssetSpecs` which indicates the assets (`RT_PROP`s, in this case) to use for these props.

## 3.55MSG_USERSTATUS

This message is sent from the server to the client to update the client as to the user's status.

The `refnum` field contains the `UserID` of the user to whom the message is sent.

The `msg` field indicates the new status information:

```
struct ClientMsg_userStatus {
    sint16 flags;
}
```

`flags` is a set of bit flags providing the status information. See the description of the `flags` field of the `UserListRec` struct in **`MSG_LISTOFALLUSERS`** for a detailed description of the flag bits themselves.

## 3.56MSG_VERSION

This server sends this message to the client to identify the server's version number. It is sent as part of the series of messages that the server sends during logon.

The `refnum` field contains the server version number. It is encoded with the major version number in the high 16 bits and the minor version number in the low 16 bits.

There are no other parameters in this message, so the `length` field should be 0 and the `msg` field should be empty:

```
struct ClientMsg_version {
}
```

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

## 3.57MSG_WHISPER

This message is similar to the MSG_TALK message, except that conversation is between individual users. This message has two forms: a client-to-server (“request”) form, in which the client requests the server to send speech on its behalf to a particular chosen other user, and a server-to-client (“action”) form, in which the server does this. The MSG_WHISPER message is also used by the server to issue private, spoken text of a variety of different kinds in a variety of different circumstances.

In the request form, the refnum field can carry an arbitrary value. It is not used by the server directly but will be echoed in the refnum field of the resulting action form message.

In the request form, the msg field is a ClientMsg_whisper_request struct:

```
struct ClientMsg_whisper_request {
    UserID  target;
    CString text;
}
```

target is the UserID of the user to whom the whisper should be directed.

text is a CString containing the text to be whispered. Although it is a CString, the text is limited to maximum of 255 characters.

In the action form, the refnum field contains the value of the refnum field from the request which generated the action.The receiving client will interpet this as the UserID of the person who is whisperingMSG_WHISPER messages generated by the server endogenously will set this field to 0.

In the action form, the msg field is simply the CString containing the text to be whispered, omitting the target field:

```
struct ClientMsg_whisper_action {
    CString text;
}
```

## 3.58MSG_XTALK

This message is used to utter word balloon speech in the Palace, similar to the MSG_TALK message. The difference is that the text is “encrypted” to inhibit sniffing of the speech over the wire. A client sends a MSG_XTALK message to the

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

server with (encrypted) text to be spoken. If the server is happy with the text, it sends a matching MSG_XTALK message to the clients in the room with the speaker. In both directions the message format is the same.

The refnum field can carry an arbitrary value. It is not used by the server directly but will be echoed in the relayed MSG_XTALK message. Clients which receive MSG_XTALK message interpret the refnum field as the UserID of the person who is talking .

The msg field is a ClientMsg_XTalk struct:

```
struct ClientMsg_xTalk {
    sint16 len;
    char   text[len];
}
```

len is the length of the text to be spoken.

text is the text itself, a CString "encrypted" with the Palace string "encryption" algorithm. Note that although CStrings are self-delimiting, the len field is required because the "encryption" can generated embedded 0 bytes. Although it is a CString, the text is limited to a maximum of 255 characters.

## 3.59 MSG_XWHISPER

This message is similar to the MSG_WHISPER message for private conversation between individual users, except that the text is "encrypted" to inhibit sniffing of the speech over the wire. This message has two forms: a client-to-server ("request") form, in which the client requests the server to send speech on its behalf to a particular chosen other user, and a server-to-client ("action") form, in which the server does this.

In the request form, the refnum field can carry an arbitrary value. It is not used by the server directly but will be echoed in the refnum field of the resulting action form message.

In the request form, the msg field is a ClientMsg_xWhisper_request struct:

```
struct ClientMsg_xWhisper_request {
    UserID target;
    sint16 len;
    char   text[len];
}
```

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

`target` is the `UserID` of the user to whom the whisper should be directed.

`len` is the length of the text to be whispered.

`text` is the text itself, a `CString` "encrypted" with the Palace string "encryption" algorithm. Note that although `CStrings` are self-delimiting, the `len` field is required because the "encryption" can generated embedded 0 bytes. Although it is a `CString`, the text is limited to a maximum of 255 characters.

In the action form, the `refnum` field contains the value of the `refnum` field from the request which generated the action.The receiving client will interpet this as the `UserID` of the person who is whispering.

In the action form, the `msg` field is similar to the request form, but omits the `target` field:

```
struct ClientMsg_xWhisper_action {
    sint16 len;
    char   text[len];
}
```

The meanings of the `len` and `text` fields are the same as in the request form.

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

# 4 Server/Frontend Message Generalities

## 4.1 The Generic Server/Frontend Message

This section describes the messages that travel between the server and the frontend. Typically (though not exclusively) these message wrap client/server messages between the server and some specific client with which the frontend is in communication. As with the client/server messages, all these messages share a common outer structure:

```
struct FrontendMsg {
    sint16 cmd;         /* 8-bit opcode, 8 bits of flags */
    uint16 dport;       /* client port number */
    sint32 dip;         /* client IP address */
    UserID userID;      /* user ID on server */
    sint32 length;      /* length of message body */
    uint8  msg[length]; /* message body */
}
```

`cmd` is a 1-byte message type, encoded in a 16-bit value in which the upper 8 bits are used for flags by some messages.

`dport` is the port number of the client to whom the embedded message is addressed (if the server is the sender) or from whom the embedded message originated (if the server is the receiver).

`dip` is the IP address that goes with `dport`

`userID` is the `UserID` of the user who is the sender or recipient of the embedded message.

`length` and `msg` describe the rest of the message. `length` is simply the number of bytes in `msg`. `msg` contains a struct whose internal form is specific to the particular message being sent (see **§5. The Server/Frontend Messages**). If a particular message requires no additional parameters, or has just a single parameter that is sent in the `refnum` field, then `length` will be 0 and the `msg` field will not be present, yielding a 16-byte structure.

## 4.2 Server/Frontend Message Summary

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

The following table lists all the server/frontend messages currently defined.

"Name" is the symbolic name used in the source code and in this document to identify the message.

"Value" is the value of the lower 8 bits of the `cmd` field in messages of this type.

"Usage" describes the pattern of use that this message experiences, according to the following legend:

| | |
|---|---|
| server → frontend | Message gets sent from the server to the frontend |
| server ← frontend | Message gets sent from the frontend to the server |
| server ↔ frontend | Frontend and server both send and receive the message |

| Name | Value | Usage |
|---|---|---|
| `bi_packet` | `0` | server ↔ frontend |
| `bi_global` | `1` | server → frontend |
| `bi_room` | `2` | server → frontend |
| `bi_serverdown` | `3` | server → frontend |
| `bi_serverfull` | `4` | server → frontend |
| `bi_serveravail` | `5` | server → frontend |
| `bi_begingroup` | `6` | server → frontend |
| `bi_endgroup` | `7` | server → frontend |
| `bi_assoc` | `8` | server → frontend |
| `bi_userflags` | `9` | server → frontend |
| `bi_addaction` | `10` | server → frontend |
| `bi_delaction` | `11` | server → frontend |
| `bi_newuser` | `12` | server ← frontend |
| `bi_kill` | `13` | server ↔ frontend |
| `bi_frontendup` | `14` | server ← frontend |
| `bi_frontenddown` | `15` | server ← frontend |

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

# 5 The Server/Frontend Messages

## 5.1 bi_packet

This message is the normal carrier for an ordinary, point-to-point client/server message (in either direction).

The `dport`, `dip` and `userID` fields identify the particular user to whom the message is being send or from whom the message is being received.

The `msg` field simply contains a `ClientMsg` struct with the message itself:

```
struct FrontendMsg_packet {
    ClientMsg message;
}
```

## 5.2 bi_global

The server sends this message to the frontend, instructing it to broadcast a particular message to all users currently attached to the frontend.

Since the message is not directed to a particular user, the `dport`, `dip` and `userID` fields are unused and all set to 0.

The `msg` field simply contains a `ClientMsg` struct with the message itself:

```
struct FrontendMsg_global {
    ClientMsg message;
}
```

## 5.3 bi_room

The server sends this message to the frontend, instructing it to broadcast a particular message to all users currently attached to the frontend who are in a particular room, or to all such users except for one particular user.

In normal use, the message is not directed to a particular user, so the `dport`, `dip` and `userID` fields are unused and all set to 0. Alternatively, these fields can be set to identify a particular user and the message will be sent to all users in the room *except* for the identified user (think of this as “send to the neighbors of

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

so-and-so”).

The `msg` field is a `FrontendMsg_room` struct:

```
struct FrontendMsg_room {
    RoomID roomID;
    ClientMsg message;
}
```

`roomID` is the `RoomID` of the room whose inhabitants are being sent the message.

`message` is the message they are being sent.

## 5.4 bi_serverdown

The server sends this message to the frontend, informing it that the server is shutting down and will no longer be accepting messages.

Since the message is directed to the frontend itself and not to any user, the `dport`, `dip` and `userID` fields are unused and all set to 0.

There are no other parameters in this message, so the `length` field should be 0 and the `msg` field should be empty:

```
struct FrontendMsg_serverDown {
}
```

## 5.5 bi_serverfull

The server sends this message to the frontend, informing it that the server is full and no longer able to accept new user logons. *<Note: the frontend defines and processes this message, but the Unix server, at least, never sends it.>*

Since the message is directed to the frontend itself and not to any user, the `dport`, `dip` and `userID` fields are unused and all set to 0.

There are no other parameters in this message, so the `length` field should be 0 and the `msg` field should be empty:

```
struct FrontendMsg_serverFull {
}
```

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

## 5.6 bi_serveravail

The server sends this message to the frontend, informing it that the server is able to accept new user logons. This is sent as the last step in the server's processing of a new frontend which has connected to the server, instructing the frontend that it may begin accepting user logons. It is also sent when the condition that resulted in an earlier bi_serverfull message being sent no longer pertains (e.g.., the number of users has dropped to an acceptable level) <*this latter case does not happen in the current Unix server code*>.

Since the message is directed to the frontend itself and not to any user, the `dport`, `dip` and `userID` fields are unused and all set to 0.

There are no other parameters in this message, so the `length` field should be 0 and the `msg` field should be empty:

```
struct FrontendMsg_serverAvail {
}
```

## 5.7 bi_begingroup

The server sends this message to the frontend. It instructs the frontend to begin buffering messages to a particular user, rather than transmitting them. These buffered message will later be transmitted together as the result of a matching `bi_endgroup` message (see **§5.8**below).

The `dport`, `dip` and `userID` fields identify the particular user for whom messages should now be buffered.

There are no other parameters in this message, so the `length` field should be 0 and the `msg` field should be empty:

```
struct FrontendMsg_beginGroup {
}
```

## 5.8 bi_endgroup

The server sends this message to the frontend. It instructs the frontend to cease buffering messages to a particular user (which buffering was started by an earlier `bi_begingroup` message) and to transmit the group of previously buffered messages to the user.

The `dport`, `dip` and `userID` fields identify the particular user whose buffered

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

messages should now be delivered.

There are no other parameters in this message, so the `length` field should be 0 and the `msg` field should be empty:

```
struct FrontendMsg_endGroup {
}
```

## 5.9 bi_assoc

The server sends this message to the frontend, instructing it to associate a particular user with a particular room, i.e., to take note what room the user is currently in. The frontend uses this information to know what to do when sent a `bi_room` message.

The `dport`, `dip` and `userID` fields identify the particular user for whose room location is being described.

The `msg` field simply identifies the room:

```
struct FrontendMsg_assoc {
    RoomID roomID;
}
```

## 5.10 bi_userflags

The server sends this message to the frontend, supplying with a collection of flag bits that provide information about a particular user.

The `dport`, `dip` and `userID` fields identify the particular user to whom this information applies.

The `msg` field contains the flag bits themselves:

```
struct FrontendMsg_userflags {
    sint16 frontendFlags;
    sint16 userFlags;
}
```

`frontendFlags` are flag bits describing information about the user's relationship to the frontend. Currently, only one flag bit is defined:

```
FE_OldClient  0x01
```

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

which indicates that the user has an older client that lacks certain features *<details need to be documented>*.

`userFlags` are flag bits that describe the user's situation in the server. The values that go here are the same as those described above in the explanation of the `flags` field of the `UserListRec` struct in **`MSG_LISTOFALLUSERS`**.

## 5.11 bi_addaction

The server sends this message to the frontend to put in place certain user-to-user relationships that result in various filtering actions being performed on those users' messages. Much of this filtering is low-level and can be performed in the frontend without loading the server.

Since the message is directed to the frontend itself and not to any user, the `dport`, `dip` and `userID` fields are unused and all set to 0.

The `msg` field is a `FrontendMsg_addAction` struct:

```
struct FrontendMsg_addAction {
    UserID cUserID;
    UserID tUserID;
    sint32 action;
}
```

`cUserID` identifies the user on whose behalf this action relationship is being established.

`tUserID` identifies the user who is the target of the action.

`action` identifies the action relationship itself. It is encoded as a series of bit flags (more than one can be specified at a time):

```
UA_HideFrom     0x01
UA_Mute         0x02
UA_Follow       0x04
UA_Kick         0x08
```

`UA_HideFrom` indicates that `cUserID` wishes to hide from `tUserID`. Consequently, `cUserID` will not appear in the list of users that `tUserID` sees. However, `cUserID` will then also not be able to whisper to `tUserID`. Also, you cannot hide from superusers. None if this is really important here, however, as all this takes place in the server, not the frontend.

`UA_Mute` indicates that `cUserID` wishes to not see talk or whisper messages

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

from `tUserID`. This filtering will be done in the frontend at message fan-out time.

`UA_Follow` appears to indicate that `cUserID` wishes to follow `tUserID`, that is, automatically change rooms to stay in the same room. However, neither the Unix server nor frontend seems to contain any support for this.

`UA_Kick` indicates that `tUserID` is not allowed to enter the room that contains `cUserID`. This is done in ther server, not the frontend.

## 5.12bi_delaction

The server sends this message to the frontend to remove an action relationship previously established with a `bi_addaction` message.

Since the message is directed to the frontend itself and not to any user, the `dport`, `dip` and `userID` fields are unused and all set to 0.

The `msg` field is a `FrontendMsg_delAction` struct:

```
struct FrontendMsg_delAction {
    UserID cUserID;
    UserID tUserID;
    sint32 action;
}
```

The fields all have the same meanings as the corresponding fields in the `bi_addaction` message described above.

## 5.13bi_newuser

The frontend sends this message to the server to inform it about the arrival of a new user.

The `dport`, `dip` and `userID` fields identify the user who has connected.

There are no other parameters in this message, so the `length` field should be 0 and the `msg` field should be empty:

```
struct FrontendMsg_newUser {
}
```

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

## 5.14bi_kill

The server sends this message to the frontend to instruct it to drop a particular user. The frontend sends this message to the server to inform it that a particular user has dropped. In either direction, the message format is the same.

The upper 8 bits of the `cmd` field contain an encoding of the reason the connection is being dropped. The values are the same as those found in the `refnum` field of the `MSG_SERVERDOWN` message.

The `dport`, `dip` and `userID` fields identify the user who has or is to be dropped.

If the value of the upper 8 bits of the `cmd` field is `K_Verbose`, the `msg` field contains a `CString` with a textual message. Otherwise, the `length` field should be 0 and the `msg` field should be empty:

```
struct FrontendMsg_kill {
    CString whyMessage;
}
```

## 5.15bi_frontendup

The frontend sends this message to the server, informing it that the frontend is present and accounted for and available for accepting messages (and connections from users).

Since the message is from the frontend itself and not from a user, the `dport`, `dip` and `userID` fields are unused and all set to 0.

There are no other parameters in this message, so the `length` field should be 0 and the `msg` field should be empty:

```
struct FrontendMsg_frontendUp {
}
```

## 5.16bi_frontenddown

The frontend sends this message to the server, informing it that the frontend is shutting down and will no longer be accepting (or sending) messages.

Since the message is from the frontend itself and not from a user, the `dport`, `dip` and `userID` fields are unused and all set to 0.

©1999 Communities.com, all rights reserved. Proprietary & confidential. 

There are no other parameters in this message, so the `length` field should be 0 and the `msg` field should be empty:

```
struct FrontendMsg_frontendDown {
}
```

©1999 Communities.com, all rights reserved. Proprietary & confidential. 