Modern IRC Client Protocol

Jack Allnutt
Kiwi IRC
jack@allnutt.eu

Daniel Oaks
ircdocs
daniel@danieloaks.net

Val Lorentz [Editor]
Limnoria
vlorentz.ircdocs@isometry.eu

This document intends to be a useful overview and reference of the IRC client protocol as it is implemented today. It is a *living specification* which is updated in response to feedback and implementations as they change. This document describes existing behaviour and what I consider best practices for new software.

This **is not a new protocol** – it is the standard IRC protocol, just described in a single document with some already widely-implemented/accepted features and capabilities. Clients written to this spec will work with old and new servers, and servers written this way will service old and new clients.

TL;DR if a new RFC was released today describing how IRC works, this is what I think it would look like.

If something written in here isn't correct for or interoperable with an IRC server / network you know of, please *open an issue* or *contact me*.

NOTE: This is a WORK IN PROGRESS. All major points of the protocol are covered, but some common message and mode types are missing.

You can contribute by sending pull requests to our *GitHub repository*!

1 Introduction

The Internet Relay Chat (IRC) protocol has been designed over a number of years, with multitudes of implementations and use cases appearing. This document describes the IRC Client-Server protocol.

IRC is a text-based chat protocol which has proven itself valuable and useful. It is well-suited to running on many machines in a distributed fashion. A typical setup involves multiple servers connected in a distributed network. Messages are delivered through this network and state is maintained across it for the connected clients and active channels.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in *RFC2119*.

☾

2 Table of Contents

☾

3   IRC Concepts

This section describes concepts behind the implementation and organisation of the IRC protocol, which are useful in understanding how it works.

3.1   Architectural

A typical IRC network consists of servers and clients connected to those servers, with a good mix of IRC operators and channels. This section goes through each of those, what they are and a brief overview of them.

3.1.1   Servers

Servers form the backbone of IRC, providing a point to which clients may connect and talk to each other, and a point for other servers to connect to, forming an IRC network.

The most common network configuration for IRC servers is that of a spanning tree [see the figure below], where each server acts as a central node for the rest of the network it sees. Other topologies are being experimented with, but right now there are none widely used in production.

```
                        [ Server 15 ]   [ Server 13 ] [ Server 14 ]
                           /                    \           /
                          /                      \         /
     [ Server 11 ] ------ [ Server 1 ]          [ Server 12 ]
                        /            \          /
                       /              \        /
           [ Server 2 ]             [ Server 3 ]
            /       \                        \
           /         \                        \
     [ Server 4 ]    [ Server 5 ]         [ Server 6 ]
     /    |    \                             /
    /     |     \                           /
   /      |      \____                     /
  /       |           \                   /
[ Server 7 ] [ Server 8 ] [ Server 9 ]   [ Server 10 ]

                    :
                 [ etc. ]
                    :
```

Figure 1: Format of a typical IRC network.

There have been several terms created over time to describe the roles of different servers on an IRC network. Some of the most common terms are as follows:

- **Hub**: A 'hub' is a server that connects to multiple other servers. For instance, in the figure above, Server 2, Server 3, and Server 4 would be examples of hub servers.

- **Core Hub**: A 'core hub' is typically a hub server that connects fairly major parts of the IRC network together. What is considered a core hub will change depending on the size of a network and what the administrators of the network consider important. For instance, in the figure above, Server 1, Server 2, and Server 3 may be considered core hubs by the network administrators.

- **Leaf**: A 'leaf' is a server that is only connected to a single other server on the network. Typically, leafs are the primary servers that handle client connections. In the figure above, Servers 7, 8, 10, 13, 14, and others would be considered leaf servers.

- **Services**: A 'services' server is a special type of server that extends the capabilities of the server software on the network (ie, they provide *services* to the network). Services are not used on all networks, and the capabilities typically provided by them may be built-into server software itself rather than being provided by a separate software package. Features usually handled by services include client account registration (as are typically used for *SASL authentication*), channel registration (allowing client accounts to 'own' channels), and further modifications and extensions to the IRC protocol. 'Services' themselves are **not** specified in any way by the protocol, they are different from the *services* defined by the RFCs. What they provide depends entirely on the software packages being run.

A trend these days is to hide the real structure of a network from regular users. Networks that implement this may restrict or modify commands like *MAP* so that regular users see every other server on the network as linked directly to the current server. When this is done, servers that do not handle client connections may also be hidden from users (hubs hidden in this way can be called 'hidden hubs'). Generally, IRC operators can always see the true structure of a network.

These terms are not generally used in IRC protocol documentation, but may be used by the administrators of a network in order to differentiate the servers they run and their roles.

Servers SHOULD pick a name which contains a dot character (`"."`, `0x2E`). This can help clients disambiguate between server names and nicknames in a message source.

3.1.2 Clients

A client is anything connecting to a server that is not another server. Each client is distinguished from other clients by a unique nickname. In addition to the nickname, all servers must have the following information about all clients: the real name/address of the host that the client is connecting from, the username of the client on that host, and the server to which the client is connected.

Nicknames are non-empty strings with the following restrictions:

- They MUST NOT contain any of the following characters: space (`' '`, `0x20`), comma (`','`, `0x2C`), asterisk (`'*'`, `0x2A`), question mark (`'?'`, `0x3F`),

exclamation mark (`'!'`, `0x21`), at sign (`'@'`, `0x40`).

- They MUST NOT start with any of the following characters: dollar (`'$'`, `0x24`), colon (`':'`, `0x3A`).

- They MUST NOT start with a character listed as a *channel type*, *channel membership prefix*, or prefix listed in the IRCv3 `multi-prefix Extension`.

- They SHOULD NOT contain any dot character (`'.'`, `0x2E`).

Servers MAY have additional implementation-specific nickname restrictions and SHOULD avoid the use of nicknames which are ambiguous with commands or command parameters where this could lead to confusion or error.

3.1.3   Services

Services were a different kind of clients than users, defined in the *RFC2812*. They were to provide or collect information about the IRC network. They are no longer used now. As such the service-related messages (SERVICE, SERVLIST and SQUERY) are also deprecated.

Operators

To allow a reasonable amount of order to be kept within the IRC network, a special class of clients (operators) are allowed to perform general maintenance functions on the network. Although the powers granted to an operator can be considered as 'dangerous', they are nonetheless required.

The tasks operators can perform vary with different server software and the specific privileges granted to each operator. Some can perform network maintenance tasks, such as disconnecting and reconnecting servers as needed to prevent long-term use of bad network routing. Some operators can also remove a user from their server or the IRC network by 'force', i.e. the operator is able to close the connection between a client and server.

The justification for operators being able to remove users from the network is delicate since its abuse is both destructive and annoying. However, IRC network policies and administrators handle operators who abuse their privileges, and what is considered abuse by that network.

3.1.4   Channels

A channel is a named group of one or more clients. All clients in the channel will receive all messages addressed to that channel. The channel is created implicitly when the first client joins it, and the channel ceases to exist when the last client leaves it. While the channel exists, any client can reference the channel using the name of the channel. Networks that support the concept of 'channel ownership' may persist specific channels in some way while no clients are connected to them.

Channel names are strings (beginning with specified prefix characters). Apart from the requirement of the first character being a valid *channel type* prefix character; the only restriction on a channel name is that it may not contain any spaces (`' '`, `0x20`), a control G / BELL (`'^G'`, `0x07`), or a comma (`','`, `0x2C`) (which is used as a list item separator by the protocol).

There are several types of channels used in the IRC protocol. The first standard type of channel is a *regular channel*, which is known to all servers that are connected to the network. The prefix character for this type of channel is (`'#'`, `0x23`). The second type are server-specific or *local channels*, where the clients connected can only see and talk to other clients on the same server. The prefix character for this type of channel is (`'&'`, `0x26`). Other types of channels are described in the *Channel Types* section.

Along with various channel types, there are also channel modes that can alter the characteristics and behaviour of individual channels. See the *Channel Modes* section for more information on these.

To create a new channel or become part of an existing channel, a user is required to join the channel using the *JOIN* command. If the channel doesn't exist prior to joining, the channel is created and the creating user becomes a channel operator. If the channel already exists, whether or not the client successfully joins that channel depends on the modes currently set on the channel. For example, if the channel is set to `invite-only` mode (`+i`), the client only joins the channel if they have been invited by another user or they have been exempted from requiring an invite by the channel operators.

Channels also contain a *topic*. The topic is a line shown to all users when they join the channel, and all users in the channel are notified when the topic of a channel is changed. Channel topics commonly state channel rules, links, quotes from channel members, a general description of the channel, or whatever the *channel operators* want to share with the clients in their channel.

A user may be joined to several channels at once, but a limit may be imposed by the server as to how many channels a client can be in at one time. This limit is specified by the *CHANLIMIT* RPL_ISUPPORT parameter. See the *Feature Advertisement* section for more details on RPL_ISUPPORT.

If the IRC network becomes disjoint because of a split between servers, the channel on either side is composed of only those clients which are connected to servers on the respective sides of the split, possibly ceasing to exist on one side. When the split is healed, the connecting servers ensure the network state is consistent between them.

Channel Operators

Channel operators (or "chanops") on a given channel are considered to 'run' or 'own' that channel. In recognition of this status, channel operators are endowed with certain powers which let them moderate and keep control of their channel.

☾

Most IRC operators do not concern themselves with 'channel politics'. In addition, a large number of networks leave the management of specific channels up to chanops where possible, and try not to interfere themselves. However, this is a matter of network policy, and it's best to consult the *Message of the Day* when looking at channel management.

IRC servers may also define other levels of channel moderation. These can include 'halfop' (half operator), 'protected' (protected user/operator), 'founder' (channel founder), and any other positions the server wishes to define. These moderation levels have varying privileges and can execute, and not execute, various channel management commands based on what the server defines.

The commands which may only be used by channel moderators include:

- *KICK*: Eject a client from the channel

- *MODE*: Change the channel's modes

- *INVITE*: Invite a client to an invite-only channel (mode +i)

- *TOPIC*: Change the channel topic in a mode +t channel

Channel moderators are identified by the channel member prefix ('@' for standard channel operators, '%' for halfops) next to their nickname whenever it is associated with a channel (e.g. replies to the *NAMES*, *WHO*, and *WHOIS* commands).

Specific prefixes and moderation levels are covered in the *Channel Membership Prefixes* section.

3.2   Communication Types

This section describes how current implementations deliver different classes of messages and is not normative.

This section ONLY deals with the spanning-tree topology, shown in the figure below. This is because spanning-tree is the topology specified and used in all IRC software today. Other topologies are being experimented with, but are not yet used in production by networks.

```
                1--\
                    A           D---4
                2--/ \          /
                      B----C
                     /      \
                    3        E

    Servers: A, B, C, D, E        Clients: 1, 2, 3, 4
```
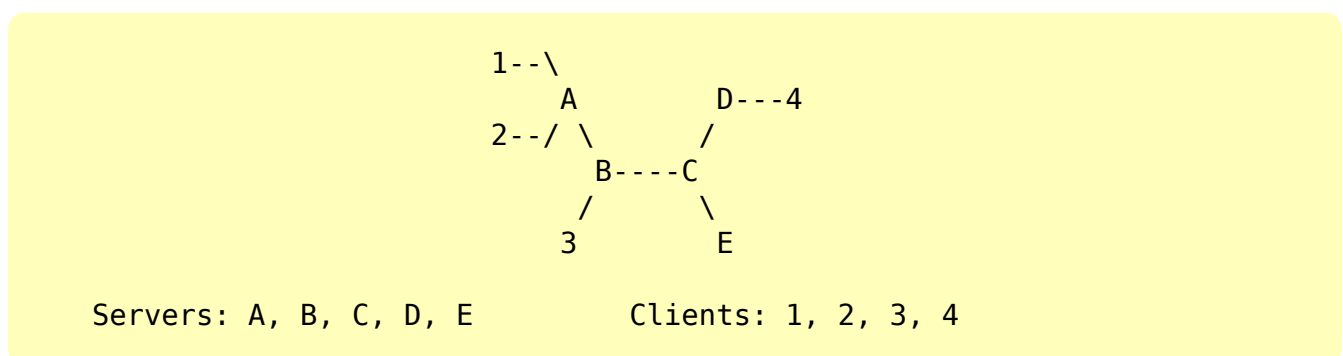
Figure 2: Sample small IRC network.

3.2.1 One-to-one communication

Communication on a one-to-one basis is usually only performed by clients, since most server-server traffic is not a result of servers talking only to each other.

Servers should be able to send a message from any one client to any other. Servers send a message in exactly one direction along the spanning tree to reach any client. Thus the path of a message being delivered is the shortest path between any two points on the spanning tree.

The following examples all refer to the figure above.

1. A message between clients 1 and 2 is only seen by server A, which sends it straight to client 2.

2. A message between clients 1 and 3 is seen by servers A, B, and client 3. No other clients or servers are allowed to see the message.

3. A message between clients 2 and 4 is seen by servers A, B, C, D, and client 4 only.

3.2.2 One-to-many communication

The main goal of IRC is to provide a forum which allows easy and efficient conferencing (one to many conversations). IRC offers several means to achieve this, each serving its own purpose.

To A Channel

In IRC, the channel has a role equivalent to that of the multicast group; their existence is dynamic and the actual conversation carried out on a channel is generally sent only to servers which are supporting users on a given channel, and only once to every local link as each server is responsible for fanning the original message to ensure it will reach all recipients.

The following examples all refer to the above figure:

1. Any channel with a single client in it. Messages to this channel go to the server and then nowhere else.

2. Two clients in a channel. All messages traverse a path as if they were private messages between the two clients outside a channel.

3. Clients 1, 2, and 3 are in a channel. All messages to this channel are sent to all clients and only those servers which must be traversed by the message if it were a private message to a single client. If client 1 sends a message, it goes back to client 2 and then via server B to client 3.

To A Host/Server Mask

To provide with some mechanism to send messages to a large body of related users, host

and server mask messages are available. These messages are sent to users whose host or server information match that of the given mask. The messages are only sent to locations where the users are, in a fashion similar to that of channels.

To A List

The least efficient style of one-to-many conversation is through clients talking to a 'list' of targets (client, channel, ask). How this is done is almost self-explanatory: the client gives a list of destinations to which the message is to be delivered and the server breaks it up and dispatches a separate copy of the message to each given destination.

This is not as efficient as using a channel since the destination list may be broken up and the dispatch sent without checking to make sure duplicates aren't sent down each path.

3.2.3   One-To-All

The one-to-all type of message is better described as a broadcast message, sent to all clients or servers or both. On a large network of users and servers, a single message can result in a lot of traffic being sent over the network in an effort to reach all of the desired destinations.

For some class of messages, there is no option but to broadcast it to all servers to that the state information held by each server is consistent between them.

Client-to-Client

IRC Operators may be able to send a message to every client currently connected to the network. This depends on the specific features and commands implemented in the server software.

Client-to-Server

Most of the commands which result in a change of state information (such as channel membership, channel modes, user status, etc.) MUST be sent to all servers by default, and this distribution SHALL NOT be changed by the client.

Server-to-Server

While most messages between servers are distributed to all 'other' servers, this is only required for any message that affects a user, channel, or server. Since these are the basic items found in IRC, nearly all messages originating from a server are broadcast to all other connected servers.

☾

4   Connection Setup

IRC client-server connections work over TCP/IP. The standard ports for client-server connections are TCP/6667 for plaintext, and TCP/6697 for TLS connections.

☾

5 Server-to-Server Protocol Structure

Both *RFC1459* and *RFC2813* define a Server-to-Server protocol. But in the decades since, implementations have extended this protocol and diverged (see *TS6* and *P10*), and servers have created entirely new protocols (see *InspIRCd*). The days where there was one Server-to-Server Protocol that everyone uses hasn't existed for a long time now.

However, different IRC implementations don't *need* to interact with each other. Networks generally run one server software across their entire network, and use the S2S protocol implemented by that server. The client protocol is important, but how servers on the network talk to each other is considered an implementation detail.

6 Client-to-Server Protocol Structure

While a client is connected to a server, they send a stream of bytes to each other. This stream contains messages separated by CR ('\r', 0x0D) and LF ('\n', 0x0A). These messages may be sent at any time from either side, and may generate zero or more reply messages.

Software SHOULD use the *UTF-8* character encoding to encode and decode messages, with fallbacks as described in the *Character Encodings* implementation considerations appendix.

Names of IRC entities (clients, servers, channels) are casemapped. This prevents, for example, someone having the nickname 'Dan' and someone else having the nickname 'dan', confusing other users. Servers MUST advertise the casemapping they use in the *RPL_ISUPPORT* numeric that's sent when connection registration has completed.

6.1 Message Format

An IRC message is a single line, delimited by a pair of CR ('\r', 0x0D) and LF ('\n', 0x0A) characters.

- When reading messages from a stream, read the incoming data into a buffer. Only parse and process a message once you encounter the \r\n at the end of it. If you encounter an empty message, silently ignore it.

- When sending messages, ensure that a pair of \r\n characters follows every single message your software sends out.

Messages have this format, as rough ABNF:

```
message         ::= ['@' <tags> SPACE] [':' <source> SPACE] <command> <pa
SPACE           ::= %x20 *( %x20 )    ; space character(s)
crlf            ::= %x0D %x0A         ; "carriage return" "linefeed"
```

The specific parts of an IRC message are:

- **tags**: Optional metadata on a message, starting with ('@', 0x40).

- **source**: Optional note of where the message came from, starting with (':', 0x3A).

- **command**: The specific command this message represents.

- **parameters**: If it exists, data relevant to this specific command.

These message parts, and parameters themselves, are separated by one or more ASCII SPACE characters (' ', 0x20).

==Most IRC servers limit messages to 512 bytes in length==, including the trailing CR-LF characters. Implementations which include *message tags* need to allow additional bytes for the **tags** section of a message; clients must allow 8191 additional bytes and servers must allow 4096 additional bytes.

The following sections describe how to process each part, but here are a few complete ==example messages==:

```
:irc.example.com CAP LS * :multi-prefix extended-join sasl

@id=234AB :dan!d@localhost PRIVMSG #chan :Hey what's up!

CAP REQ :sasl
```

6.1.1 Tags

This is the format of the **tags** part:

```
<tags>          ::= <tag> [';' <tag>]*
<tag>           ::= <key> ['=' <escaped value>]
<key>           ::= [ <client_prefix> ] [ <vendor> '/' ] <sequence of let
<client_prefix> ::= '+'
<escaped value> ::= <sequence of any characters except NUL, CR, LF, semic
<vendor>        ::= <host>
```

Basically, a series of `<key>[=<value>]` segments, separated by (`';'`, `0x3B`).

==The **tags** part is optional, and MUST NOT be sent unless explicitly enabled by *a capability*==. This message part starts with a leading (`'@'`, `0x40`) character, which MUST be the first character of the message itself. The leading (`'@'`, `0x40`) is stripped from the value before it gets processed further.

Here are some examples of tags sections and how they could be represented as *JSON* objects:

```
@id=123AB;rose         ->  {"id": "123AB", "rose": ""}

@url=;netsplit=tur,ty  ->  {"url": "", "netsplit": "tur,ty"}
```

For more information on processing tags – including the naming and registration of them, and how to escape values – see the IRCv3 *Message Tags specification*.

6.1.2 Source

☾

```
source        ::=  <servername> / ( <nickname> [ "!" <user> ] [ "@" <ho
nick          ::=  <any characters except NUL, CR, LF, chantype charact
user          ::=  <sequence of any characters except NUL, CR, LF, and
```

The **source** (formerly known as **prefix**) is optional and starts with a (`':'`, `0x3A`) character (which is stripped from the value), and if there are no tags it MUST be the first character of the message itself.

The source indicates the true origin of a message. If the source is missing from a message, it's is assumed to have originated from the client/server on the other end of the connection the message was received on.

Clients MUST NOT include a source when sending a message.

Servers MAY include a source on any message, and MAY leave a source off of any message. Clients MUST be able to process any given message the same way whether it contains a source or does not contain one.

6.1.3 Command

```
command       ::=  letter* / 3digit
```

The **command** must either be a valid IRC command or a numeric (a three-digit number represented as text).

Information on specific commands / numerics can be found in the *Client Messages* and *Numerics* sections, respectively.

6.1.4 Parameters

This is the format of the **parameters** part:

```
parameters    ::=  *( SPACE middle ) [ SPACE ":" trailing ]
nospcrlfcl    ::=  <sequence of any characters except NUL, CR, LF, colo
middle        ::=  nospcrlfcl *( ":" / nospcrlfcl )
trailing      ::=  *( ":" / " " / nospcrlfcl )
```

Parameters (or 'params') are extra pieces of information added to the end of a message. These parameters generally make up the 'data' portion of the message. What specific parameters mean changes for every single message.

Parameters are a series of values separated by one or more ASCII SPACE characters (`' ', 0x20`). However, this syntax is insufficient in two cases: a parameter that contains one or more spaces, and an empty parameter. To permit such parameters, the final parameter can be prepended with a (`':', 0x3A`) character, in which case that character is stripped and the rest of the message is treated as the final parameter,

including any spaces it contains. Parameters that contain spaces, are empty, or begin with a ':' character MUST be sent with a preceding ':'; in other cases the use of a preceding ':' on the final parameter is OPTIONAL.

Software SHOULD AVOID sending more than 15 parameters, as older client protocol documents specified this was the maximum and some clients may have trouble reading more than this. However, clients MUST parse incoming messages with any number of them.

Here are some examples of messages and how the parameters would be represented as *JSON* lists:

```
:irc.example.com CAP * LIST :        ->  ["*", "LIST", ""]

CAP * LS :multi-prefix sasl          ->  ["*", "LS", "multi-prefix sasl"

CAP REQ :sasl message-tags foo       ->  ["REQ", "sasl message-tags foo"

:dan!d@localhost PRIVMSG #chan :Hey!  ->  ["#chan", "Hey!"]

:dan!d@localhost PRIVMSG #chan Hey!   ->  ["#chan", "Hey!"]

:dan!d@localhost PRIVMSG #chan ::-)   ->  ["#chan", ":-)"]
```

As these examples show, a trailing parameter (a final parameter with a preceding ':') has the same semantics as any other parameter, and MUST NOT be treated specially or stored separately once the ':' is stripped.

6.1.5 Compatibility with incorrect software

Servers SHOULD handle single \n character, and MAY handle a single \r character, as if it was a \r\n pair, to support existing clients that might send this. However, clients and servers alike MUST NOT send single \r or \n characters.

Servers and clients SHOULD ignore empty lines.

Servers SHOULD gracefully handle messages over the 512-bytes limit. They may:

- Send an error numeric back, preferably *ERR_INPUTTOOLONG* (417)

- Truncate on the 510th byte (and add \r\n at the end) or, preferably, on the last UTF-8 character or grapheme that fits.

- Ignore the message or close the connection – but this may be confusing to users of buggy clients.

Finally, clients and servers SHOULD NOT use more than one space (\x20) character as SPACE as defined in the grammar above.

🌙

6.2   Numeric Replies

Most messages sent from a client to a server generates a reply of some sort. The most common form of reply is the numeric reply, used for both errors and normal replies. Distinct from a normal message, a numeric reply MUST contain a `<source>` and use a three-digit numeric as the command. A numeric reply SHOULD contain the target of the reply as the first parameter of the message. A numeric reply is not allowed to originate from a client.

In all other respects, a numeric reply is just like a normal message. A list of numeric replies is supplied in the *Numerics* section.

6.3   Wildcard Expressions

When wildcards are allowed in a string, it is referred to as a "mask".

For string matching purposes, the protocol allows the use of two special characters: (`'?'`, `0x3F`) to match one and only one character, and (`'*'`, `0x2A`) to match any number of any characters. These two characters can be escaped using the (`'\'`, `0x5C`) character.

The ABNF syntax for this is:

```
mask        =  *( nowild / noesc wildone / noesc wildmany )
wildone     =  %x3F
wildmany    =  %x2A
nowild      =  %x01-29 / %x2B-3E / %x40-FF
                 ; any octet except NUL, "*", "?"
noesc       =  %x01-5B / %x5D-FF
                 ; any octet except NUL and "\"

matchone    =  %x01-FF
                 ; matches wildone
matchmany   =  *matchone
                 ; matches wildmany
```

Examples:

```
a?c         ; Matches any string of 3 characters in length starting
              with "a" and ending with "c"

a*c         ; Matches any string of 2 or more characters in length
              starting with "a" and ending with "c"
```

7 Connection Registration

Immediately upon establishing a connection the client must attempt registration, without waiting for any banner message from the server.

Until registration is complete, only a limited subset of commands SHOULD be accepted by the server. This is because it makes sense to require a registered (fully connected) client connection before allowing commands such as *JOIN*, *PRIVMSG* and others.

The recommended order of commands during registration is as follows:

1. `CAP LS 302`

2. PASS

3. NICK and USER

4. *Capability Negotiation*

5. SASL (if negotiated)

6. `CAP END`

The commands specified in steps 1-3 should be sent on connection. If the server supports *capability negotiation* then registration will be suspended and the client can negotiate client capabilities (steps 4-6). If the server does not support capability negotiation then registration will continue immediately without steps 4-6.

1. If the server supports capability negotiation, the *CAP* command suspends the registration process and immediately starts the *capability negotiation* process. CAP LS 302 means that the client supports *version 302* of client capability negotiation. The registration process is resumed when the client sends CAP END to the server.

2. The *PASS* command is not required for the connection to be registered, but if included it MUST precede the latter of the *NICK* and *USER* commands.

3. The *NICK* and *USER* commands are used to set the user's nickname, username and "real name". Unless the registration is suspended by a *CAP* negotiation, these commands will end the registration process.

4. The client should request advertised capabilities it wishes to enable here.

5. If the client supports *SASL authentication* and wishes to authenticate with the server, it should attempt this after a successful *CAP ACK* of the `sasl` capability is received and while registration is suspended.

6. If the server support capability negotiation, *CAP END* will end the negotiation period and resume the registration.

If the server is waiting to complete a lookup of client information (such as hostname or ident for a username), there may be an arbitrary wait at some point during registration. Servers SHOULD set a reasonable timeout for these lookups.

Additionally, some servers also send a *PING* and require a matching *PONG* from the client before continuing. This exchange may happen immediately on connection and at any time during connection registration, so clients MUST respond correctly to it.

Upon successful completion of the registration process, the server MUST send, in this order:

1. *RPL_WELCOME* (`001`),

2. *RPL_YOURHOST* (`002`),

3. *RPL_CREATED* (`003`),

4. *RPL_MYINFO* (`004`),

5. at least one *RPL_ISUPPORT* (`005`) numeric to the client.

6. The server MAY then send other numerics and messages.

7. The server SHOULD then respond as though the client sent the *LUSERS* command and return the appropriate numerics.

8. The server MUST then respond as though the client sent it the *MOTD* command, i.e. it must send either the successful *Message of the Day* numerics or the *ERR_NOMOTD* (`422`) numeric.

9. If the user has client modes set on them automatically upon joining the network, the server SHOULD send the client the *RPL_UMODEIS* (`221`) reply or a *MODE* message with the client as target, preferably the former.

The first parameter of the *RPL_WELCOME* (`001`) message is the nickname assigned by the network to the client. Since it may differ from the nickname the client requested with the `NICK` command (due to, e.g. length limits or policy restrictions on nicknames), the client SHOULD use this parameter to determine its actual nickname at the time of connection. Subsequent nickname changes, client-initiated or not, will be communicated by the server sending a *NICK* message.

☾

8 Feature Advertisement

IRC servers and networks implement many different IRC features, limits, and protocol options that clients should be aware of. The *RPL_ISUPPORT* `(005)` numeric is designed to advertise these features to clients on connection registration, providing a simple way for clients to change their behaviour based on what is implemented on the server.

Once client registration is complete, the server MUST send at least one `RPL_ISUPPORT` numeric to the client. The server MAY send more than one `RPL_ISUPPORT` numeric and consecutive `RPL_ISUPPORT` numerics SHOULD be sent adjacent to each other.

Clients SHOULD NOT assume a server supports a feature unless it has been advertised in `RPL_ISUPPORT`. For `RPL_ISUPPORT` parameters which specify a 'default' value, clients SHOULD assume the default value for these parameters until the server advertises these parameters itself. This is generally done for compatibility reasons with older versions of the IRC protocol that do not specify the `RPL_ISUPPORT` numeric and servers that do not advertise those specific tokens.

For more information and specific details on tokens, see the *RPL_ISUPPORT* `(005)` reply.

A list of `RPL_ISUPPORT` parameters is available in the *RPL_ISUPPORT Parameters* section.

9 Capability Negotiation

Over the years, various extensions to the IRC protocol have been made by server programmers. Often, these extensions are intended to conserve bandwidth, close loopholes left by the original protocol specification, or add new features for users or for server administrators. Most of these changes are backwards-compatible with the base protocol specifications: A command may be added, a reply may be extended to contain more parameters, etc. However, there are extensions which are designed to change protocol behaviour in a backwards-incompatible way.

Capability Negotiation is a mechanism for the negotiation of protocol extensions, known as **client capabilities**, that makes sure servers implementing backwards-incompatible protocol extensions still interoperate with existing clients, and vice-versa.

Clients implementing capability negotiation will still interoperate with servers that do not implement it; similarly, servers that implement capability negotiation will successfully communicate with clients that do not implement it.

IRC is an asynchronous protocol, which means that clients may issue additional IRC commands while previous commands are being processed. Additionally, there is no guarantee of a specific kind of banner being issued upon connection. Some servers also do not complain about unknown commands during registration, which means that a client cannot reliably do passive implementation discovery at registration time.

The solution to these problems is to allow for active capability negotiation, and to extend the registration process with this negotiation. If the server supports capability negotiation, the registration process will be suspended until negotiation is completed. If the server does not support this, then registration will complete immediately and the client will not use any capabilities.

Capability negotiation is started by the client issuing a `CAP LS 302` command (indicating to the server support for IRCv3.2 capability negotiation). Negotiation is then performed with the `CAP REQ`, `CAP ACK`, and `CAP NAK` commands, and is ended with the `CAP END` command.

If used during initial registration, and the server supports capability negotiation, the `CAP` command will suspend registration. Once capability negotiation has ended the registration process will continue.

Clients and servers should implement capability negotiation and the CAP command based on the *Capability Negotiation specification*. Updates, improvements, and new versions of capability negotiation are managed by the *IRCv3 Working Group*.

☾

10   Client Messages

Messages are client-to-server only unless otherwise specified. If messages may be sent from the server to a connected client, it will be noted in the message's description. For server-to-client messages of this type, the message `<source>` usually indicates the client the message relates to, but this will be noted in the description.

In message descriptions, 'command' refers to the message's behaviour when sent from a client to the server. Similarly, 'Command Examples' represent example messages sent from a client to the server, and 'Message Examples' represent example messages sent from the server to a client. If a command is sent from a client to a server with less parameters than the command requires to be processed, the server will reply with an *ERR_NEEDMOREPARAMS* `(461)` numeric and the command will fail.

In the `"Parameters:"` section, optional parts or parameters are noted with square brackets as such: `"[<param>]"`. Curly braces around a part of parameter indicate that it may be repeated zero or more times, for example: `"<key>{,<key>}"` indicates that there must be at least one `<key>`, and that there may be additional keys separated by the comma (`","`, `0x2C`) character.

10.1   Connection Messages

10.1.1   CAP message

```
    Command: CAP
 Parameters: <subcommand> [:<capabilities>]
```

The CAP command is used for capability negotiation between a server and a client.

The CAP message may be sent from the server to the client.

For the exact semantics of the CAP command and subcommands, please see the *Capability Negotiation specification*.

10.1.2   AUTHENTICATE message

```
    Command: AUTHENTICATE
```

The AUTHENTICATE command is used for SASL authentication between a server and a client. The client must support and successfully negotiate the `"sasl"` client capability (as listed below in the SASL specifications) before using this command.

The AUTHENTICATE message may be sent from the server to the client.

For the exact semantics of the AUTHENTICATE command and negotiating support for the "sasl" client capability, please see the *IRCv3.1* and *IRCv3.2* SASL Authentication specifications.

10.1.3 PASS message

```
    Command: PASS
 Parameters: <password>
```

The PASS command is used to set a 'connection password'. If set, the password must be set before any attempt to register the connection is made. This requires that clients send a PASS command before sending the NICK / USER combination.

The password supplied must match the one defined in the server configuration. It is possible to send multiple PASS commands before registering but only the last one sent is used for verification and it may not be changed once the client has been registered.

If the password supplied does not match the password expected by the server, then the server SHOULD send *ERR_PASSWDMISMATCH* (464) and MAY then close the connection with *ERROR*. Servers MUST send at least one of these two messages.

Servers may also consider requiring *SASL authentication* upon connection as an alternative to this, when more information or an alternate form of identity verification is desired.

Numeric replies:

- *ERR_NEEDMOREPARAMS* (461)

- *ERR_ALREADYREGISTERED* (462)

- *ERR_PASSWDMISMATCH* (464)

Command Example:

```
 PASS secretpasswordhere
```

10.1.4 NICK message

```
    Command: NICK
 Parameters: <nickname>
```

The NICK command is used to give the client a nickname or change the previous one.

If the server receives a NICK command from a client where the desired nickname is already in use on the network, it should issue an ERR_NICKNAMEINUSE numeric and

ignore the NICK command.

If the server does not accept the new nickname supplied by the client as valid (for instance, due to containing invalid characters), it should issue an ERR_ERRONEUSNICKNAME numeric and ignore the NICK command. Servers MUST allow at least all alphanumerical characters, square and curly brackets ([]{}), backslashes (\), and pipe (|) characters in nicknames, and MAY disallow digits as the first character. Servers MAY allow extra characters, as long as they do not introduce ambiguity in other commands, including:

- no leading # character or other character advertized in *CHANTYPES*

- no leading colon (:)

- no ASCII space

If the server does not receive the `<nickname>` parameter with the NICK command, it should issue an ERR_NONICKNAMEGIVEN numeric and ignore the NICK command.

The NICK message may be sent from the server to clients to acknowledge their NICK command was successful, and to inform other clients about the change of nickname. In these cases, the `<source>` of the message will be the old `nickname [ [ "!" user ] "@" host ]` of the user who is changing their nickname.

Numeric Replies:

- *ERR_NONICKNAMEGIVEN* (431)

- *ERR_ERRONEUSNICKNAME* (432)

- *ERR_NICKNAMEINUSE* (433)

- *ERR_NICKCOLLISION* (436)

Command Example:

```
    NICK Wiz                    ; Requesting the new nick "Wiz".
```

Message Examples:

```
    :WiZ NICK Kilroy            ; WiZ changed his nickname to Kilroy.

    :dan-!d@localhost NICK Mamoped
                                ; dan- changed his nickname to Mamoped.
```

10.1.5 USER message

```
   Command: USER
Parameters: <username> 0 * <realname>
```

The USER command is used at the beginning of a connection to specify the username and realname of a new user.

It must be noted that `<realname>` must be the last parameter because it may contain SPACE (`' '`, `0x20`) characters, and should be prefixed with a colon (`:`) if required.

Servers MAY use the *Ident Protocol* to look up the 'real username' of clients. If username lookups are enabled and a client does not have an Identity Server enabled, the username provided by the client SHOULD be prefixed by a tilde (`'~'`, `0x7E`) to show that this value is user-set.

The maximum length of `<username>` may be specified by the *USERLEN* `RPL_ISUPPORT` parameter. If this length is advertised, the username MUST be silently truncated to the given length before being used. The minimum length of `<username>` is 1, ie. it MUST NOT be empty. If it is empty, the server SHOULD reject the command with *ERR_NEEDMOREPARAMS* (even if an empty parameter is provided); otherwise it MUST use a default value instead.

The second and third parameters of this command SHOULD be sent as one zero (`'0'`, `0x30`) and one asterisk character (`'*'`, `0x2A`) by the client, as the meaning of these two parameters varies between different versions of the IRC protocol.

Clients SHOULD use the nickname as a fallback value for `<username>` and `<realname>` when they don't have a meaningful value to use.

If a client tries to send the USER command after they have already completed registration with the server, the ERR_ALREADYREGISTERED reply should be sent and the attempt should fail.

If the client sends a USER command after the server has successfully received a username using the Ident Protocol, the `<username>` parameter from this command should be ignored in favour of the one received from the identity server.

Numeric Replies:

- *ERR_NEEDMOREPARAMS* (461)

- *ERR_ALREADYREGISTERED* (462)

Command Examples:

☾

```
    USER guest 0 * :Ronnie Reagan
                            ; No ident server
                            ; User gets registered with username
                            "~guest" and real name "Ronnie Reagan"

    USER guest 0 * :Ronnie Reagan
                            ; Ident server gets contacted and
                            returns the name "danp"
                            ; User gets registered with username
                            "danp" and real name "Ronnie Reagan"
```

10.1.6   PING message

```
      Command: PING
   Parameters: <token>
```

The PING command is sent by either clients or servers to check the other side of the connection is still connected and/or to check for connection latency, at the application layer.

The <token> may be any non-empty string.

When receiving a PING message, clients or servers must reply to it with a *PONG* message with the same <token> value. This allows either to match PONG with the PING they reply to, for example to compute latency.

Clients should not send PING during connection registration, though servers may accept it. Servers may send PING during connection registration and clients must reply to them.

Older versions of the protocol gave specific semantics to the <token> and allowed an extra parameter; but these features are not consistently implemented and should not be relied on. Instead, the <token> should be treated as an opaque value by the receiver.

Numeric Replies:

- *ERR_NEEDMOREPARAMS* (461)

- *ERR_NOORIGIN* (409)

Deprecated Numeric Reply:

- *ERR_NOSUCHSERVER* (402)

10.1.7   PONG message

```
      Command: PONG
   Parameters: [<server>] <token>
```

The PONG command is used as a reply to *PING* commands, by both clients and servers. The `<token>` should be the same as the one in the `PING` message that triggered this `PONG`.

Servers MUST send a `<server>` parameter, and clients SHOULD ignore it. It exists for historical reasons, and indicates the name of the server sending the PONG. Clients MUST NOT send a `<server>` parameter.

Numeric Replies:

- None

10.1.8 OPER message

```
    Command: OPER
  Parameters: <name> <password>
```

The `OPER` command is used by a normal user to obtain IRC operator privileges. Both parameters are required for the command to be successful.

If the client does not send the correct password for the given name, the server replies with an `ERR_PASSWDMISMATCH` message and the request is not successful.

If the client is not connecting from a valid host for the given name, the server replies with an `ERR_NOOPERHOST` message and the request is not successful.

If the supplied name and password are both correct, and the user is connecting from a valid host, the `RPL_YOUREOPER` message is sent to the user. The user will also receive a *MODE* message indicating their new user modes, and other messages may be sent.

The `<name>` specified by this command is separate to the accounts specified by SASL authentication, and is generally stored in the IRCd configuration.

Numeric Replies:

- *ERR_NEEDMOREPARAMS* (461)

- *ERR_PASSWDMISMATCH* (464)

- *ERR_NOOPERHOST* (491)

- *RPL_YOUREOPER* (381)

Command Example:

```
  OPER foo bar              ; Attempt to register as an operator
                              using a name of "foo" and the password "bar".
```

☾

10.1.9   QUIT message

```
    Command: QUIT
 Parameters: [<reason>]
```

The QUIT command is used to terminate a client's connection to the server. The server acknowledges this by replying with an *ERROR* message and closing the connection to the client.

This message may also be sent from the server to a client to show that a client has exited from the network. This is typically only dispatched to clients that share a channel with the exiting user. When the QUIT message is sent to clients, <source> represents the client that has exited the network.

When connections are terminated by a client-sent QUIT command, servers SHOULD prepend <reason> with the ASCII string "Quit: " when sending QUIT messages to other clients, to represent that this user terminated the connection themselves. This applies even if <reason> is empty, in which case the reason sent to other clients SHOULD be just this "Quit: " string. However, clients SHOULD NOT change behaviour based on the prefix of QUIT message reasons, as this is not required behaviour from servers.

When a netsplit (the disconnecting of two servers) occurs, a QUIT message is generated for each client that has exited the network, distributed in the same way as ordinary QUIT messages. The <reason> on these QUIT messages SHOULD be composed of the names of the two servers involved, separated by a SPACE (' ', 0x20). The first name is that of the server which is still connected and the second name is that of the server which has become disconnected. If servers wish to hide or obscure the names of the servers involved, the <reason> on these messages MAY also be the literal ASCII string "*.net *.split" (i.e. the two server names are replaced with "*.net" and "*.split"). Software that implements the IRCv3 *batch Extension* should also look at the *netsplit and netjoin* batch types.

If a client connection is closed without the client issuing a QUIT command to the server, the server MUST distribute a QUIT message to other clients informing them of this, distributed in the same was an ordinary QUIT message. Servers MUST fill <reason> with a message reflecting the nature of the event which caused it to happen. For instance, "Ping timeout: 120 seconds", "Excess Flood", and "Too many connections from this IP" are examples of relevant reasons for closing or for a connection with a client to have been closed.

Numeric Replies:

- None

Command Example:

```
    QUIT :Gone to have lunch          ; Client exiting from the network
```

Message Example:

```
    :dan-!d@localhost QUIT :Quit: Bye for now!
                                      ; dan- is exiting the network with
                                      the message: "Quit: Bye for now!"
```

10.1.10   ERROR message

```
      Command: ERROR
   Parameters: <reason>
```

This message is sent from a server to a client to report a fatal error, before terminating the client's connection.

This MUST only be used to report fatal errors. Regular errors should use the appropriate numerics or the IRCv3 *standard replies* framework.

Numeric Replies:

- None

Command Example:

```
    ERROR :Connection timeout         ; Server closing a client connection bec
                                      is unresponsive.
```

10.2   Channel Operations

This group of messages is concerned with manipulating channels, their properties (channel modes), and their contents (typically clients).

These commands may be requests to the server, in which case the server will or will not grant the request. If a 'request' is granted, it will be acknowledged by the server sending a message containing the same information back to the client. This is to tell the user that the request was successful. These sort of 'request' commands will be noted in the message information.

In implementing these messages, race conditions are inevitable when clients at opposing ends of a network send commands which will ultimately clash. Server-to-server protocols should be aware of this and make sure their protocol ensures consistent state across the entire network.

☾

10.2.1   JOIN message

```
    Command: JOIN
 Parameters: <channel>{,<channel>} [<key>{,<key>}]
 Alt Params: 0
```

The `JOIN` command indicates that the client wants to join the given channel(s), each channel using the given key for it. The server receiving the command checks whether or not the client can join the given channel, and processes the request. Servers MUST process the parameters of this command as lists on incoming commands from clients, with the first `<key>` being used for the first `<channel>`, the second `<key>` being used for the second `<channel>`, etc.

While a client is joined to a channel, they receive all relevant information about that channel including the `JOIN`, `PART`, `KICK`, and `MODE` messages affecting the channel. They receive all `PRIVMSG` and `NOTICE` messages sent to the channel, and they also receive `QUIT` messages from other clients joined to the same channel (to let them know those users have left the channel and the network). This allows them to keep track of other channel members and channel modes.

If a client's `JOIN` command to the server is successful, the server MUST send, in this order:

1. A `JOIN` message with the client as the message `<source>` and the channel they have joined as the first parameter of the message.

2. The channel's topic (with *RPL_TOPIC* (`332`) and optionally *RPL_TOPICWHOTIME* (`333`)), and no message if the channel does not have a topic.

3. A list of users currently joined to the channel (with one or more *RPL_NAMREPLY* (`353`) numerics followed by a single *RPL_ENDOFNAMES* (`366`) numeric). These RPL_NAMREPLY messages sent by the server MUST include the requesting client that has just joined the channel.

The *key*, *client limit* , *ban - exception*, *invite-only - exception*, and other (depending on server software) channel modes affect whether or not a given client may join a channel. More information on each of these modes and how they affect the `JOIN` command is available in their respective sections.

Servers MAY restrict the number of channels a client may be joined to at one time. This limit SHOULD be defined in the *CHANLIMIT* RPL_ISUPPORT parameter. If the client cannot join this channel because they would be over their limit, they will receive an *ERR_TOOMANYCHANNELS* (`405`) reply and the command will fail.

Note that this command also accepts the special argument of (`"0"`, `0x30`) instead of any of the usual parameters, which requests that the sending client leave all channels they are currently connected to. The server will process this command as though the

client had sent a *PART* command for each channel they are a member of.

This message may be sent from a server to a client to notify the client that someone has joined a channel. In this case, the message `<source>` will be the client who is joining, and `<channel>` will be the channel which that client has joined. Servers SHOULD NOT send multiple channels in this message to clients, and SHOULD distribute these multiple-channel `JOIN` messages as a series of messages with a single channel name on each.

Numeric Replies:

- *ERR_NEEDMOREPARAMS* (461)

- *ERR_NOSUCHCHANNEL* (403)

- *ERR_TOOMANYCHANNELS* (405)

- *ERR_BADCHANNELKEY* (475)

- *ERR_BANNEDFROMCHAN* (474)

- *ERR_CHANNELISFULL* (471)

- *ERR_INVITEONLYCHAN* (473)

- *ERR_BADCHANMASK* (476)

- *RPL_TOPIC* (332)

- *RPL_TOPICWHOTIME* (333)

- *RPL_NAMREPLY* (353)

- *RPL_ENDOFNAMES* (366)

Command Examples:

```
JOIN #foobar                    ; join channel #foobar.

JOIN &foo fubar                 ; join channel &foo using key "fubar".

JOIN #foo,&bar fubar            ; join channel #foo using key "fubar"
                                and &bar using no key.

JOIN #foo,#bar fubar,foobar     ; join channel #foo using key "fubar".
                                and channel #bar using key "foobar".

JOIN #foo,#bar                  ; join channels #foo and #bar.
```

Message Examples:

☾

```
    :WiZ JOIN #Twilight_zone        ; WiZ is joining the channel
                                    #Twilight_zone

    :dan-!d@localhost JOIN #test    ; dan- is joining the channel #test
```

See also:

- IRCv3 *extended-join Extension*

10.2.2   PART message

```
     Command: PART
  Parameters: <channel>{,<channel>} [<reason>]
```

The PART command removes the client from the given channel(s). On sending a successful PART command, the user will receive a PART message from the server for each channel they have been removed from. `<reason>` is the reason that the client has left the channel(s).

For each channel in the parameter of this command, if the channel exists and the client is not joined to it, they will receive an *ERR_NOTONCHANNEL* (442) reply and that channel will be ignored. If the channel does not exist, the client will receive an *ERR_NOSUCHCHANNEL* (403) reply and that channel will be ignored.

This message may be sent from a server to a client to notify the client that someone has been removed from a channel. In this case, the message `<source>` will be the client who is being removed, and `<channel>` will be the channel which that client has been removed from. Servers SHOULD NOT send multiple channels in this message to clients, and SHOULD distribute these multiple-channel PART messages as a series of messages with a single channel name on each. If a PART message is distributed in this way, `<reason>` (if it exists) should be on each of these messages.

Numeric Replies:

- *ERR_NEEDMOREPARAMS* (461)

- *ERR_NOSUCHCHANNEL* (403)

- *ERR_NOTONCHANNEL* (442)

Command Examples:

```
  PART #twilight_zone             ; leave channel "#twilight_zone"

  PART #oz-ops,&group5            ; leave both channels "&group5" and
                                  "#oz-ops".
```

Message Examples:

```
:dan-!d@localhost PART #test    ; dan- is leaving the channel #test
```

10.2.3 TOPIC message

```
   Command: TOPIC
Parameters: <channel> [<topic>]
```

The TOPIC command is used to change or view the topic of the given channel. If `<topic>` is not given, either RPL_TOPIC or RPL_NOTOPIC is returned specifying the current channel topic or lack of one. If `<topic>` is an empty string, the topic for the channel will be cleared.

If the client sending this command is not joined to the given channel, and tries to view its' topic, the server MAY return the *ERR_NOTONCHANNEL* (442) numeric and have the command fail.

If RPL_TOPIC is returned to the client sending this command, RPL_TOPICWHOTIME SHOULD also be sent to that client.

If the *protected topic* mode is set on a channel, then clients MUST have appropriate channel permissions to modify the topic of that channel. If a client does not have appropriate channel permissions and tries to change the topic, the *ERR_CHANOPRIVSNEEDED* (482) numeric is returned and the command will fail.

If the topic of a channel is changed or cleared, every client in that channel (including the author of the topic change) will receive a TOPIC command with the new topic as argument (or an empty argument if the topic was cleared) alerting them to how the topic has changed. If the `<topic>` param is provided but the same as the previous topic (ie. it is unchanged), servers MAY notify the author and/or other users anyway.

Clients joining the channel in the future will receive a RPL_TOPIC numeric (or lack thereof) accordingly.

Numeric Replies:

- *ERR_NEEDMOREPARAMS* (461)

- *ERR_NOSUCHCHANNEL* (403)

- *ERR_NOTONCHANNEL* (442)

- *ERR_CHANOPRIVSNEEDED* (482)

- *RPL_NOTOPIC* (331)

- *RPL_TOPIC* (332)

- *RPL_TOPICWHOTIME* (333)

Command Examples:

```
   TOPIC #test :New topic          ; Setting the topic on "#test" to
                                   "New topic".

   TOPIC #test :                   ; Clearing the topic on "#test"

   TOPIC #test                     ; Checking the topic for "#test"
```

10.2.4   NAMES message

```
    Command: NAMES
 Parameters: <channel>{,<channel>}
```

The NAMES command is used to view the nicknames joined to a channel and their *channel membership prefixes*. The param of this command is a list of channel names, delimited by a comma (`","`, `0x2C`) character.

The channel names are evaluated one-by-one. For each channel that exists and they are able to see the users in, the server returns one of more RPL_NAMREPLY numerics containing the users joined to the channel and a single RPL_ENDOFNAMES numeric. If the channel name is invalid or the channel does not exist, one RPL_ENDOFNAMES numeric containing the given channel name should be returned. If the given channel has the *secret* channel mode set and the user is not joined to that channel, one RPL_ENDOFNAMES numeric is returned. Users with the *invisible* user mode set are not shown in channel responses unless the requesting client is also joined to that channel.

Servers MAY allow more than one target channel. They can advertise the maximum the number of target users per NAMES command via the *TARGMAX* RPL_ISUPPORT parameter.

Numeric Replies:

- *RPL_NAMREPLY* (353)

- *RPL_ENDOFNAMES* (366)

Command Examples:

☾

```
    NAMES #twilight_zone,#42        ; List all visible users on
                                    "#twilight_zone" and "#42".

    NAMES                           ; Attempt to list all visible users on
                                    the network, which SHOULD be responded to
                                    as specified above.
```

See also:

- IRCv3 *multi-prefix Extension*

- IRCv3 *userhost-in-names Extension*

10.2.5   LIST message

```
    Command: LIST
 Parameters: [<channel>{,<channel>}] [<elistcond>{,<elistcond>}]
```

The LIST command is used to get a list of channels along with some information about each channel. Both parameters to this command are optional as they have different syntaxes.

The first possible parameter to this command is a list of channel names, delimited by a comma (",", 0x2C) character. If this parameter is given, the information for only the given channels is returned. If this parameter is not given, the information about all visible channels (those not hidden by the *secret* channel mode rules) is returned.

The second possible parameter to this command is a list of conditions as defined in the *ELIST* RPL_ISUPPORT parameter, delimited by a comma (",", 0x2C) character. Clients MUST NOT submit an ELIST condition unless the server has explicitly defined support for that condition with the ELIST token. If this parameter is supplied, the server filters the returned list of channels with the given conditions as specified in the *ELIST* documentation.

In response to a successful LIST command, the server MAY send one RPL_LISTSTART numeric, MUST send back zero or more RPL_LIST numerics, and MUST send back one RPL_LISTEND numeric.

Numeric Replies:

- *RPL_LISTSTART* (321)

- *RPL_LIST* (322)

- *RPL_LISTEND* (323)

Command Examples:

```
    LIST                           ; Command to list all channels

    LIST #twilight_zone,#42        ; Command to list the channels
                                   "#twilight_zone" and "#42".

    LIST >3                        ; Command to list all channels with
                                   more than three users.

    LIST C>60                      ; Command to list all channels with
                                   created at least 60 minutes ago

    LIST T<60                      ; Command to list all channels with
                                   a topic changed within the last 60 minute
```

10.2.6   INVITE message

```
    Command: INVITE
 Parameters: <nickname> <channel>
```

The INVITE command is used to invite a user to a channel. The parameter <nickname> is the nickname of the person to be invited to the target channel <channel>.

The target channel SHOULD exist (at least one user is on it). Otherwise, the server SHOULD reject the command with the ERR_NOSUCHCHANNEL numeric.

Only members of the channel are allowed to invite other users. Otherwise, the server MUST reject the command with the ERR_NOTONCHANNEL numeric.

Servers MAY reject the command with the ERR_CHANOPRIVSNEEDED numeric. In particular, they SHOULD reject it when the channel has *invite-only* mode set, and the user is not a channel operator.

If the user is already on the target channel, the server MUST reject the command with the ERR_USERONCHANNEL numeric.

When the invite is successful, the server MUST send a RPL_INVITING numeric to the command issuer, and an INVITE message, with the issuer as <source>, to the target user. Other channel members SHOULD NOT be notified.

Numeric Replies:

- *RPL_INVITING* (341)

- *ERR_NEEDMOREPARAMS* (461)

- *ERR_NOSUCHCHANNEL* (403)

- *ERR_NOTONCHANNEL* (442)

- *ERR_CHANOPRIVSNEEDED* (482)

- *ERR_USERONCHANNEL* (443)

Command Examples:

```
INVITE Wiz #foo_bar    ; Invite Wiz to #foo_bar
```

Message Examples:

```
:dan-!d@localhost INVITE Wiz #test    ; dan- has invited Wiz
                                        to the channel #test
```

See also:

- IRCv3 *invite-notify Extension*

Invite list

Servers MAY allow the INVITE with no parameter, and reply with a list of channels the sender is invited to as *RPL_INVITELIST* (336) numerics, ending with a *RPL_ENDOFINVITELIST* (337) numeric.

> Some rare implementations use numerics 346/347 instead of 336/337 as `RPL_INVITELIST`/`RPL_ENDOFINVITELIST`. You should check the server you are using implements them as expected.
>
> 346/347 now generally stands for `RPL_INVEXLIST`/`RPL_ENDOFINVEXLIST`, used for *invite-exception list*.

10.2.7   KICK message

```
  Command: KICK
Parameters: <channel> <user> *( "," <user> ) [<comment>]
```

The KICK command can be used to request the forced removal of a user from a channel. It causes the <user> to be removed from the <channel> by force.

This message may be sent from a server to a client to notify the client that someone has been removed from a channel. In this case, the message <source> will be the client who sent the kick, and <channel> will be the channel which the target client has been removed from.

If no comment is given, the server SHOULD use a default message instead.

Servers MUST NOT send multiple users in this message to clients, and MUST distribute these multiple-user KICK messages as a series of messages with a single user name on each. This is necessary to maintain backward compatibility with existing client software. If a KICK message is distributed in this way, `<comment>` (if it exists) should be on each of these messages.

Servers MAY limit the number of target users per KICK command via the *TARGMAX parameter of RPL_ISUPPORT*, and silently drop targets if the number of targets exceeds the limit.

Numeric Replies:

- *ERR_NEEDMOREPARAMS* (461)

- *ERR_NOSUCHCHANNEL* (403)

- *ERR_CHANOPRIVSNEEDED* (482)

- *ERR_USERNOTINCHANNEL* (441)

- *ERR_NOTONCHANNEL* (442)

Deprecated Numeric Reply:

- *ERR_BADCHANMASK* (476)

Examples:

```
KICK #Finnish Matthew           ; Command to kick Matthew from
                                #Finnish

KICK &Melbourne Matthew         ; Command to kick Matthew from
                                &Melbourne

KICK #Finnish John :Speaking English
                                ; Command to kick John from #Finnish
                                using "Speaking English" as the
                                reason (comment).

:WiZ!jto@tolsun.oulu.fi KICK #Finnish John
                                ; KICK message on channel #Finnish
                                from WiZ to remove John from channel
```

10.3   Server Queries and Commands

10.3.1   MOTD message

🌙

```
    Command: MOTD
 Parameters: [<target>]
```

The MOTD command is used to get the "Message of the Day" of the given server. If <target> is not given, the MOTD of the server the client is connected to should be returned.

If <target> is a server, the MOTD for that server is requested. If <target> is given and a matching server cannot be found, the server will respond with the ERR_NOSUCHSERVER numeric and the command will fail.

If the MOTD can be found, one RPL_MOTDSTART numeric is returned, followed by one or more RPL_MOTD numeric, then one RPL_ENDOFMOTD numeric.

If the MOTD does not exist or could not be found, the ERR_NOMOTD numeric is returned.

Numeric Replies:

- *ERR_NOSUCHSERVER* (402)

- *ERR_NOMOTD* (422)

- *RPL_MOTDSTART* (375)

- *RPL_MOTD* (372)

- *RPL_ENDOFMOTD* (376)

10.3.2 VERSION Message

```
    Command: VERSION
 Parameters: [<target>]
```

The VERSION command is used to query the version of the software and the *RPL_ISUPPORT parameters* of the given server. If <target> is not given, the information for the server the client is connected to should be returned.

If <target> is a server, the information for that server is requested. If <target> is a client, the information for the server that client is connected to is requested. If <target> is given and a matching server cannot be found, the server will respond with the ERR_NOSUCHSERVER numeric and the command will fail.

Wildcards are allowed in the <target> parameter.

Upon receiving a VERSION command, the given server SHOULD respond with one RPL_VERSION reply and one or more RPL_ISUPPORT replies.

Numeric Replies:

- *ERR_NOSUCHSERVER* (402)

- *RPL_ISUPPORT* (005)

- *RPL_VERSION* (351)

Command Examples:

```
:Wiz VERSION *.se              ; message from Wiz to check the
                               version of a server matching "*.se"

VERSION tolsun.oulu.fi         ; check the version of server
                               "tolsun.oulu.fi".
```

10.3.3   ADMIN message

```
    Command: ADMIN
 Parameters: [<target>]
```

The ADMIN command is used to find the name of the administrator of the given server. If `<target>` is not given, the information for the server the client is connected to should be returned.

If `<target>` is a server, the information for that server is requested. If `<target>` is a client, the information for the server that client is connected to is requested. If `<target>` is given and a matching server cannot be found, the server will respond with the ERR_NOSUCHSERVER numeric and the command will fail.

Wildcards are allowed in the `<target>` parameter.

Upon receiving an ADMIN command, the given server SHOULD respond with the RPL_ADMINME, RPL_ADMINLOC1, RPL_ADMINLOC2, and RPL_ADMINEMAIL replies.

Numeric Replies:

- *ERR_NOSUCHSERVER* (402)

- *RPL_ADMINME* (256)

- *RPL_ADMINLOC1* (257)

- *RPL_ADMINLOC2* (258)

- *RPL_ADMINEMAIL* (259)

Command Examples:

☾

```
    ADMIN tolsun.oulu.fi              ; request an ADMIN reply from
                                      tolsun.oulu.fi

    ADMIN syrk                        ; ADMIN request for the server to
                                      which the user syrk is connected
```

10.3.4   CONNECT message

```
   Command: CONNECT
Parameters: <target server> [<port> [<remote server>]]
```

The CONNECT command forces a server to try to establish a new connection to another server. CONNECT is a privileged command and is available only to IRC Operators. If a remote server is given, the connection is attempted by that remote server to <target server> using <port>.

Numeric Replies:

- *ERR_NOSUCHSERVER* (402)

- *ERR_NEEDMOREPARAMS* (461)

- *ERR_NOPRIVILEGES* (481)

- *ERR_NOPRIVS* (723)

Command Examples:

```
CONNECT tolsun.oulu.fi
; Attempt to connect the current server to tololsun.oulu.fi

CONNECT eff.org 12765 csd.bu.edu
; Attempt to connect csu.bu.edu to eff.org on port 12765
```

10.3.5   LUSERS message

```
   Command: LUSERS
Parameters: None
```

Returns statistics about local and global users, as numeric replies.

Servers MUST reply with RPL_LUSERCLIENT and RPL_LUSERME, and SHOULD also include all those defined below.

Clients SHOULD NOT try to parse the free-form text in the trailing parameter, and rely on specific parameters instead.

- *RPL_LUSERCLIENT* (251)

- *RPL_LUSEROP* (252)

- *RPL_LUSERUNKNOWN* (253)

- *RPL_LUSERCHANNELS* (254)

- *RPL_LUSERME* (255)

- *RPL_LOCALUSERS* (265)

- *RPL_GLOBALUSERS* (266)

10.3.6   TIME message

```
    Command: TIME
 Parameters: [<server>]
```

The TIME command is used to query local time from the specified server. If the server parameter is not given, the server handling the command must reply to the query.

Numeric Replies:

- *ERR_NOSUCHSERVER* (402)

- *RPL_TIME* (391)

Command Examples:

```
 TIME tolsun.oulu.fi               ; check the time on the server
                                   "tolson.oulu.fi"

 :Angel TIME *.au                  ; user angel checking the time on a
                                   server matching "*.au"
```

See also:

- IRCv3 *server-time Extension*

10.3.7   STATS message

```
    Command: STATS
 Parameters: <query> [<server>]
```

The STATS command is used to query statistics of a certain server. The specific queries supported by this command depend on the server that replies, although the server must be able to supply information as described by the queries below (or similar).

A query may be given by any single letter which is only checked by the destination server and is otherwise passed on by intermediate servers, ignored and unaltered.

The following queries are those found in current IRC implementations and provide a large portion of the setup and runtime information for that server. All servers should be able to supply a valid reply to a STATS query which is consistent with the reply formats currently used and the purpose of the query.

The currently supported queries are:

- `c` - returns a list of servers which the server may connect to or allow connections from;

- `h` - returns a list of servers which are either forced to be treated as leaves or allowed to act as hubs;

- `i` - returns a list of hosts which the server allows a client to connect from;

- `k` - returns a list of banned username/hostname combinations for that server;

- `l` - returns a list of the server's connections, showing how long each connection has been established and the traffic over that connection in bytes and messages for each direction;

- `m` - returns a list of commands supported by the server and the usage count for each if the usage count is non zero;

- `o` - returns a list of hosts from which normal clients may become operators;

- `u` - returns a string showing how long the server has been up.

- `y` - show Y (Class) lines from server's configuration file;

> Need to give this a good look-over. It's probably quite incorrect.

Numeric Replies:

- *ERR_NOSUCHSERVER* (402)

- *ERR_NEEDMOREPARAMS* (461)

- *ERR_NOPRIVILEGES* (481)

- *ERR_NOPRIVS* (723)

- RPL_STATSCLINE (213)

- RPL_STATSHLINE (244)

- RPL_STATSILINE (215)

- RPL_STATSKLINE (216)

- RPL_STATSLLINE (241)

- RPL_STATSOLINE (243)

- RPL_STATSLINKINFO (211)

- *RPL_STATSUPTIME* (242)

- *RPL_STATSCOMMANDS* (212)

- *RPL_ENDOFSTATS* (219)

Command Examples:

```
    STATS m                            ; check the command usage for the
                                       server you are connected to

    :Wiz STATS c eff.org               ; request by WiZ for C/N line
                                       information from server eff.org
```

10.3.8   HELP message

```
     Command: HELP
  Parameters: [<subject>]
```

The HELP command is used to return documentation about the IRC server and the IRC commands it implements.

When receiving a HELP command, servers MUST either: reply with a single *ERR_HELPNOTFOUND* (524) message; or reply with a single *RPL_HELPSTART* (704) message, then arbitrarily many *RPL_HELPTXT* (705) messages, then a single *RPL_ENDOFHELP* (706). Servers MAY return the *RPL_HELPTXT* (705) form for unknown subjects, especially if their reply would not fit in a single line.

The *RPL_HELPSTART* (704) message SHOULD be some sort of title and the first *RPL_HELPTXT* (705) message SHOULD be empty. This is what most servers do today.

Servers MAY define any `<subject>` they want. Servers typically have documentation for most of the IRC commands they support.

Clients SHOULD gracefully handle older servers that reply to HELP with a set of *NOTICE* messages. On these servers, the client may try sending the HELPOP command (with the same syntax specified here), which may return the numeric-based reply.

Clients SHOULD also gracefully handle servers that reply to HELP with a set of

290/291/292/293/294/295 numerics.

Numerics:

- *ERR_HELPNOTFOUND* (524)

- *RPL_HELPSTART* (704)

- *RPL_HELPTXT* (705)

- *RPL_ENDOFHELP* (706)

Command Examples:

```
HELP                                                ; request generi
:server 704 val * :** Help System **               ; first line
:server 705 val * :
:server 705 val * :Try /HELP <command> for specific help,
:server 705 val * :/HELP USERCMDS to list available
:server 706 val * :commands, or join the #help channel   ; last line

HELP PRIVMSG                                        ; request help o
:server 704 val PRIVMSG :** The PRIVMSG command **
:server 705 val PRIVMSG :
:server 705 val PRIVMSG :The /PRIVMSG command is the main way
:server 706 val PRIVMSG :to send messages to other users.

HELP :unknown subject                               ; request help o
:server 524 val * :I do not know anything about this

HELP :unknown subject
:server 704 val * :** Help System **
:server 705 val * :
:server 705 val * :I do not know anything about this.
:server 705 val * :
:server 705 val * :Try /HELP USERCMDS to list available
:server 706 val * :commands, or join the #help channel
```

10.3.9   INFO message

```
   Command: INFO
Parameters: None
```

The INFO command is used to return information which describes the server. This information usually includes the software name/version and its authors. Some other info that may be returned includes the patch level and compile date of the server, the copyright on the server software, and whatever miscellaneous information the server authors consider relevant.

Upon receiving an INFO command, the server will respond with zero or more RPL_INFO

replies, followed by one RPL_ENDOFINFO numeric.

Numeric Replies:

- *RPL_INFO* (371)

- *RPL_ENDOFINFO* (374)

Command Examples:

```
INFO                             ; request info from the server
```

10.3.10   MODE message

```
    Command: MODE
 Parameters: <target> [<modestring> [<mode arguments>...]]
```

The MODE command is used to set or remove options (or *modes*) from a given target.

User mode

If `<target>` is a nickname that does not exist on the network, the *ERR_NOSUCHNICK* (401) numeric is returned. If `<target>` is a different nick than the user who sent the command, the *ERR_USERSDONTMATCH* (502) numeric is returned.

If `<modestring>` is not given, the *RPL_UMODEIS* (221) numeric is sent back containing the current modes of the target user.

If `<modestring>` is given, the supplied modes will be applied, and a MODE message will be sent to the user containing the changed modes. If one or more modes sent are not implemented on the server, the server MUST apply the modes that are implemented, and then send the *ERR_UMODEUNKNOWNFLAG* (501) in reply along with the MODE message.

Channel mode

If `<target>` is a channel that does not exist on the network, the *ERR_NOSUCHCHANNEL* (403) numeric is returned.

If `<modestring>` is not given, the *RPL_CHANNELMODEIS* (324) numeric is returned. Servers MAY choose to hide sensitive information such as channel keys when sending the current modes. Servers MAY also return the *RPL_CREATIONTIME* (329) numeric following RPL_CHANNELMODEIS.

If `<modestring>` is given, the user sending the command MUST have appropriate channel privileges on the target channel to change the modes given. If a user does not have appropriate privileges to change modes on the target channel, the server MUST NOT process the message, and *ERR_CHANOPRIVSNEEDED* (482) numeric is returned. If

the user has permission to change modes on the target, the supplied modes will be applied based on the type of the mode (see below). For type A, B, and C modes, arguments will be sequentially obtained from <mode arguments>. If a type B or C mode does not have a parameter when being set, the server MUST ignore that mode. If a type A mode has been sent without an argument, the contents of the list MUST be sent to the user, unless it contains sensitive information the user is not allowed to access. When the server is done processing the modes, a `MODE` command is sent to all members of the channel containing the mode changes. Servers MAY choose to hide sensitive information when sending the mode changes.

<modestring> starts with a plus (`'+'`, 0x2B) or minus (`'-'`, 0x2D) character, and is made up of the following characters:

- `'+'`: Adds the following mode(s).

- `'-'`: Removes the following mode(s).

- `'a-zA-Z'`: Mode letters, indicating which modes are to be added/removed.

The ABNF representation for <modestring> is:

```
modestring  =  1*( modeset )
modeset     =  plusminus *( modechar )
plusminus   =  %x2B / %x2D
                 ; + or -
modechar    =  ALPHA
```

There are four categories of channel modes, defined as follows:

- **Type A**: Modes that add or remove an address to or from a list. These modes MUST always have a parameter when sent from the server to a client. A client MAY issue this type of mode without an argument to obtain the current contents of the list. The numerics used to retrieve contents of Type A modes depends on the specific mode. Also see the *EXTBAN* parameter.

- **Type B**: Modes that change a setting on a channel. These modes MUST always have a parameter.

- **Type C**: Modes that change a setting on a channel. These modes MUST have a parameter when being set, and MUST NOT have a parameter when being unset.

- **Type D**: Modes that change a setting on a channel. These modes MUST NOT have a parameter.

Channel mode letters, along with their types, are defined in the *CHANMODES* parameter. User mode letters are always **Type D** modes.

The meaning of standard (and/or well-used) channel and user mode letters can be found in the *Channel Modes* and *User Modes* sections. The meaning of any mode letters not in

this list are defined by the server software and configuration.

Type A modes are lists that can be viewed. The method of viewing these lists is not standardised across modes and different numerics are used for each. The specific numerics used for these are outlined here:

- **Ban List "+b"**: Ban lists are returned with zero or more *RPL_BANLIST* (367) numerics, followed by one *RPL_ENDOFBANLIST* (368) numeric.

- **Exception List "+e"**: Exception lists are returned with zero or more *RPL_EXCEPTLIST* (348) numerics, followed by one *RPL_ENDOFEXCEPTLIST* (349) numeric.

- **Invite-Exception List "+I"**: Invite-exception lists are returned with zero or more *RPL_INVITELIST* (336) numerics, followed by one *RPL_ENDOFINVITELIST* (337) numeric.

After the initial `MODE` command is sent to the server, the client receives the above numerics detailing the entries that appear on the given list. Servers MAY choose to restrict the above information to channel operators, or to only those clients who have permissions to change the given list.

Command Examples:

```
MODE dan +i                  ; Setting the "invisible" user mode on da

MODE #foobar +mb *@127.0.0.1 ; Setting the "moderated" channel mode an
                               adding the "*@127.0.0.1" mask to the ban
                               list of the #foobar channel.
```

Message Examples:

```
:dan!~h@localhost MODE #foobar -bl+i *@192.168.0.1
                             ; dan unbanned the "*@192.168.0.1" mask,
                               removed the client limit from, and set th
                               #foobar channel to invite-only.

:irc.example.com MODE #foobar +o bunny
                             ; The irc.example.com server gave channel
                               operator privileges to bunny on #foobar.
```

Requesting modes for a channel:

```
MODE #foobar
```

Getting modes for a channel (and channel creation time):

☾

```
    :irc.example.com 324 dan #foobar +nrt
    :irc.example.com 329 dan #foobar 1620807422
```

10.4 Sending Messages

10.4.1 PRIVMSG message

```
    Command: PRIVMSG
 Parameters: <target>{,<target>} <text to be sent>
```

The PRIVMSG command is used to send private messages between users, as well as to send messages to channels. `<target>` is the nickname of a client or the name of a channel.

If `<target>` is a channel name and the client is *banned* and not covered by a *ban exception*, the message will not be delivered and the command will silently fail. Channels with the *moderated* mode active may block messages from certain users. Other channel modes may affect the delivery of the message or cause the message to be modified before delivery, and these modes are defined by the server software and configuration being used.

If a message cannot be delivered to a channel, the server SHOULD respond with an *ERR_CANNOTSENDTOCHAN* (404) numeric to let the user know that this message could not be delivered.

If `<target>` is a channel name, it may be prefixed with one or more *channel membership prefix character (@, +, etc)* and the message will be delivered only to the members of that channel with the given or higher status in the channel. Servers that support this feature will list the prefixes which this is supported for in the *STATUSMSG* RPL_ISUPPORT parameter, and this SHOULD NOT be attempted by clients unless the prefix has been advertised in this token.

If `<target>` is a user and that user has been set as away, the server may reply with an *RPL_AWAY* (301) numeric and the command will continue.

The PRIVMSG message is sent from the server to client to deliver a message to that client. The `<source>` of the message represents the user or server that sent the message, and the `<target>` represents the target of that PRIVMSG (which may be the client, a channel, etc).

When the PRIVMSG message is sent from a server to a client and `<target>` starts with a dollar character (`'$', 0x24`), the message is a broadcast sent to all clients on one or multiple servers.

Numeric Replies:

- *ERR_NOSUCHNICK* (401)

- *ERR_NOSUCHSERVER* (402)

- *ERR_CANNOTSENDTOCHAN* (404)

- ERR_TOOMANYTARGETS (407)

- *ERR_NORECIPIENT* (411)

- *ERR_NOTEXTTOSEND* (412)

- ERR_NOTOPLEVEL (413)

- ERR_WILDTOPLEVEL (414)

- *RPL_AWAY* (301)

> There are strange "X@Y" target rules and such which are noted in the examples of the original PRIVMSG RFC section. We need to check to make sure modern servers actually process them properly, and if so then specify them.

Command Examples:

```
PRIVMSG Angel :yes I'm receiving it !
                                  ; Command to send a message to Angel.

PRIVMSG %#bunny :Hi! I have a problem!
                                  ; Command to send a message to halfops
                                  and chanops on #bunny.

PRIVMSG @%#bunny :Hi! I have a problem!
                                  ; Command to send a message to halfops
                                  and chanops on #bunny. This command is
                                  functionally identical to the above
                                  command.
```

Message Examples:

```
:Angel PRIVMSG Wiz :Hello are you receiving this message ?
                                  ; Message from Angel to Wiz.

:dan!~h@localhost PRIVMSG #coolpeople :Hi everyone!
                                  ; Message from dan to the channel
                                  #coolpeople
```

☾0.4.2   NOTICE message

```
    Command: NOTICE
 Parameters: <target>{,<target>} <text to be sent>
```

The `NOTICE` command is used to send notices between users, as well as to send notices to channels. `<target>` is interpreted the same way as it is for the *PRIVMSG* command.

The `NOTICE` message is used similarly to *PRIVMSG*. The difference between `NOTICE` and *PRIVMSG* is that automatic replies must never be sent in response to a `NOTICE` message. This rule also applies to servers – they must not send any error back to the client on receipt of a `NOTICE` command. The intention of this is to avoid loops between a client automatically sending something in response to something it received. This is typically used by 'bots' (a client with a program, and not a user, controlling their actions) and also for server messages to clients.

One thing for bot authors to note is that the `NOTICE` message may be interpreted differently by various clients. Some clients highlight or interpret any `NOTICE` sent to a channel in the same way that a `PRIVMSG` with their nickname gets interpreted. This means that users may be irritated by the use of `NOTICE` messages rather than `PRIVMSG` messages by clients or bots, and they are not commonly used by client bots for this reason.

10.5 User-Based Queries

10.5.1 WHO message

```
    Command: WHO
 Parameters: <mask>
```

This command is used to query a list of users who match the provided mask. The server will answer this command with zero, one or more *RPL_WHOREPLY*, and end the list with *RPL_ENDOFWHO*.

The mask can be one of the following:

- A channel name, in which case the channel members are listed.

- An exact nickname, in which case a single user is returned.

- A mask pattern, in which case all visible users whose nickname matches are listed. Servers MAY match other user-specific values, such as the hostname, server, real name or username. Servers MAY not support mask patterns and return an empty list.

Visible users are users who either aren't invisible (*user mode +i*) or have a common channel with the requesting client. Servers MAY filter or limit visible users replies arbitrarily.

Numeric Replies:

- *RPL_WHOREPLY* (352)

- *RPL_ENDOFWHO* (315)

See also:

- IRCv3 *multi-prefix Extension*

- *WHOX*

Examples

Command Examples:

```
WHO emersion        ; request information on user "emersion"
WHO #ircv3          ; list users in the "#ircv3" channel
```

Reply Examples:

```
:calcium.libera.chat 352 dan #ircv3 ~emersion sourcehut/staff/emersion ca
:calcium.libera.chat 315 dan emersion :End of WHO list
                            ; Reply to WHO emersion

:calcium.libera.chat 352 dan #ircv3 ~emersion sourcehut/staff/emersion ca
:calcium.libera.chat 352 dan #ircv3 ~val limnoria/val calcium.libera.chat
:calcium.libera.chat 315 dan #ircv3 :End of WHO list
                            ; Reply to WHO #ircv3
```

10.5.2   WHOIS message

```
   Command: WHOIS
Parameters: [<target>] <nick>
```

This command is used to query information about a particular user. The server SHOULD answer this command with numeric messages with information about the nick.

The server SHOULD end its response (to a syntactically well-formed client message) with *RPL_ENDOFWHOIS*, even if it did not send any other numeric message. This allows clients to stop waiting for new numerics. In exceptional error conditions, servers MAY not reply to a WHOIS command. Clients SHOULD implement a hard timeout to avoid waiting for a reply which won't come.

Client MUST NOT not assume all numeric messages are sent at once, as server can interleave other messages before the end of the WHOIS response.

If the <target> parameter is specified, it SHOULD be a server name or the nick of a user.

Servers SHOULD send the query to a specific server with that name, or to the server `<target>` is connected to, respectively. Typically, it is used by clients who want to know how long the user in question has been idle (as typically only the server the user is directly connected to knows that information, while everything else this command returns is globally known).

The following numerics MAY be returned as part of the whois reply:

- *ERR_NOSUCHNICK* (401)

- *ERR_NOSUCHSERVER* (402)

- *ERR_NONICKNAMEGIVEN* (431)

- *RPL_WHOISCERTFP* (276)

- *RPL_WHOISREGNICK* (307)

- *RPL_WHOISUSER* (311)

- *RPL_WHOISSERVER* (312)

- *RPL_WHOISOPERATOR* (313)

- *RPL_WHOISIDLE* (317)

- *RPL_WHOISCHANNELS* (319)

- *RPL_WHOISSPECIAL* (320)

- *RPL_WHOISACCOUNT* (330)

- *RPL_WHOISACTUALLY* (338)

- *RPL_WHOISHOST* (378)

- *RPL_WHOISMODES* (379)

- *RPL_WHOISSECURE* (671)

- *RPL_AWAY* (301)

Servers typically send some of these numerics only to the client itself and to servers operators, as they contain privacy-sensitive information that should not be revealed to other users.

Server implementers wishing to send information not covered by these numerics may send other vendor-specific numerics, such that:

- the first and second parameters MUST be the client's nick, and the target nick, and

- the last parameter SHOULD be designed to be human-readable, so that user

interfaces can display unknown numerics

Additionally, server implementers should consider submitting these to *IRCv3* for standardization, if relevant.

Optional extensions

This section describes extension to the common `WHOIS` command above. They exist mainly on historical servers, and are rarely implemented, because of resource usage they incur.

- Servers MAY allow more than one target nick. They can advertise the maximum the number of target users per `WHOIS` command via the *TARGMAX* RPL_ISUPPORT parameter, and silently drop targets if the number of targets exceeds the limit.

- Servers MAY allow wildcards in `<nick>`. Servers who do SHOULD reply with information about all matching nicks. They may restrict what information is available in this case, to limit resource usage.

- IRCv3 *multi-prefix Extension*

Examples

Command Examples:

```
WHOIS val                    ; request information on user "val"
WHOIS val val                ; request information on user "val",
                               from the server they are on
WHOIS calcium.libera.chat val ; request information on user "val",
                               from server calcium.libera.chat
```

Reply Example:

```
:calcium.libera.chat 311 val val ~val limnoria/val * :Val
:calcium.libera.chat 319 val val :#ircv3 #libera +#limnoria
:calcium.libera.chat 319 val val :#weechat
:calcium.libera.chat 312 val val calcium.libera.chat :Montreal, CA
:calcium.libera.chat 671 val val :is using a secure connection [TLSv1.3, 
:calcium.libera.chat 317 val val 657 1628028154 :seconds idle, signon tim
:calcium.libera.chat 330 val val pinkieval :is logged in as
:calcium.libera.chat 318 val val :End of /WHOIS list.
```

10.5.3   WHOWAS message

```
   Command: WHOWAS
Parameters: <nick> [<count>]
```

Whowas asks for information about a nickname which no longer exists. This may either

be due to a nickname change or the user leaving IRC. In response to this query, the server searches through its nickname history, looking for any nicks which are lexically the same (no wild card matching here). The history is searched backward, returning the most recent entry first. If there are multiple entries, up to `<count>` replies will be returned (or all of them if no `<count>` parameter is given).

If given, `<count>` SHOULD be a positive number. Otherwise, a full search is done.

Servers MUST reply with either *ERR_WASNOSUCHNICK* (406) or a non-empty list of WHOWAS entries, both followed with *RPL_ENDOFWHOWAS* (369)

A WHOWAS entry is a series of numeric messages starting with *RPL_WHOWASUSER* (314), optionally followed by other numerics relevant to that user, such as *RPL_WHOISACTUALLY* (338) and *RPL_WHOISSERVER* (312). Clients MUST NOT assume any particular numeric other than *RPL_WHOWASUSER* (314) is present in a WHOWAS entry.

If the `<nick>` argument is missing, they SHOULD send a single reply, using either *ERR_NONICKNAMEGIVEN* (431) or *ERR_NEEDMOREPARAMS* (461).

Examples

Command Examples:

```
WHOWAS someone
WHOWAS someone 2
```

Reply Examples:

```
:inspircd.server.example 314 val someone ident3 127.0.0.1 * :Realname
:inspircd.server.example 312 val someone My.Little.Server :Sun Mar 20 202
:inspircd.server.example 314 val someone ident2 127.0.0.1 * :Realname
:inspircd.server.example 312 val someone My.Little.Server :Sun Mar 20 202
:inspircd.server.example 369 val someone :End of WHOWAS

:ergo.server.example 314 val someone ~ident3 127.0.0.1 * Realname
:ergo.server.example 314 val someone ~ident2 127.0.0.1 * Realname
:ergo.server.example 369 val someone :End of WHOWAS

:solanum.server.example 314 val someone ~ident3 localhost * :Realname
:solanum.server.example 338 val someone 127.0.0.1 :actually using host
:solanum.server.example 312 val someone solanum.server.example :Sun Mar 2
:solanum.server.example 314 val someone ~ident2 localhost * :Realname
:solanum.server.example 338 val someone 127.0.0.1 :actually using host
:solanum.server.example 312 val someone solanum.server.example :Sun Mar 2
:solanum.server.example 369 val someone :End of WHOWAS

:server.example 406 val someone :There was no such nickname
:server.example 369 val someone :End of WHOWAS
```

10.6 Operator Messages

The following messages are typically reserved to server operators.

10.6.1 KILL message

```
    Command: KILL
 Parameters: <nickname> <comment>
```

The KILL command is used to close the connection between a given client and the server they are connected to. KILL is a privileged command and is available only to IRC Operators. `<nickname>` represents the user to be 'killed', and `<comment>` is shown to all users and to the user themselves upon being killed.

When a KILL command is used, the client being killed receives the KILL message, and the `<source>` of the message SHOULD be the operator who performed the command. The user being killed and every user sharing a channel with them receives a *QUIT* message representing that they are leaving the network. The `<reason>` on this QUIT message typically has the form: `"Killed (<killer> (<reason>))"` where `<killer>` is the nickname of the user who performed the KILL. The user being killed then receives the *ERROR* message, typically containing a `<reason>` of `"Closing Link: <servername> (Killed (<killer> (<reason>)))"`. After this, their connection is closed.

If a KILL message is received by a client, it means that the user specified by `<nickname>` is being killed. With certain servers, users may elect to receive KILL messages created for other users to keep an eye on the network. This behavior may also be restricted to operators.

Clients can rejoin instantly after this command is performed on them. However, it can serve as a warning to a user to stop their activity. As it breaks the flow of data from the user, it can also be used to stop large amounts of 'flooding' from abusive users or due to accidents. Abusive users may not care and promptly reconnect and resume their abusive behaviour. In these cases, opers may look at the *KLINE* command to keep them from rejoining the network for a longer time.

As nicknames across an IRC network MUST be unique, if duplicates are found when servers join, one or both of the clients MAY be KILLed and removed from the network. Servers may also handle this case in alternate ways that don't involve removing users from the network.

Servers MAY restrict whether specific operators can remove users on other servers (remote users). If the operator tries to remove a remote user but is not privileged to, they should receive the *ERR_NOPRIVS* (723) numeric.

🌙 `<comment>` SHOULD reflect why the KILL was performed. For user-generated KILLs, it is

up to the user to provide an adequate reason.

Numeric Replies:

- *ERR_NOSUCHSERVER* (402)

- *ERR_NEEDMOREPARAMS* (461)

- *ERR_NOPRIVILEGES* (481)

- *ERR_NOPRIVS* (723)

> NOTE: The KILL message is weird, and I need to look at it more closely, add some examples, etc.

10.6.2   REHASH message

```
    Command: REHASH
 Parameters: None
```

The REHASH command is an administrative command which can be used by an operator to force the local server to re-read and process its configuration file. This may include other data, such as modules or TLS certificates.

Servers MAY accept, as an optional argument, the name of a remote server that should be rehashed instead of the current one.

Numeric replies:

- *RPL_REHASHING* (382)

- *ERR_NOPRIVILEGES* (481)

Example:

```
REHASH                              ; message from user with operator
                                    status to server asking it to reread
                                    its configuration file.
```

10.6.3   RESTART message

```
    Command: RESTART
 Parameters: None
```

An operator can use the restart command to force the server to restart itself. This message is optional since it may be viewed as a risk to allow arbitrary people to connect to a server as an operator and execute this command, causing (at least) a disruption to service.

Numeric replies:

- *ERR_NOPRIVILEGES* (481)

Example:

```
RESTART                              ; no parameters required.
```

10.6.4   SQUIT message

```
   Command: SQUIT
 Parameters: <server> <comment>
```

The SQUIT command disconnects a server from the network. SQUIT is a privileged command and is only available to IRC Operators. <comment> is the reason why the server link is being disconnected.

In a traditional spanning-tree topology, the command gets forwarded to the specified server. And the link between the specified server and the last server to propagate the command gets broken.

Numeric replies:

- *ERR_NOSUCHSERVER* (402)

- *ERR_NEEDMOREPARAMS* (461)

- *ERR_NOPRIVILEGES* (481)

- *ERR_NOPRIVS* (723)

Examples:

```
SQUIT tolsun.oulu.fi :Bad Link ?  ; Command to uplink of the server
                                  tolson.oulu.fi to terminate its
                                  connection with comment "Bad Link".
```

10.7   Optional Messages

These messages are not required for a server implementation to work, but SHOULD be implemented. If a command is not implemented, it MUST return the

ERR_UNKNOWNCOMMAND (421) numeric.

10.7.1 AWAY message

```
     Command: AWAY
  Parameters: [<text>]
```

The AWAY command lets clients indicate that their user is away. If this command is sent with a nonempty parameter (the 'away message') then the user is set to be away. If this command is sent with no parameters, or with the empty string as the parameter, the user is no longer away.

The server acknowledges the change in away status by returning the *RPL_NOWAWAY* (306) and *RPL_UNAWAY* (305) numerics. If the *IRCv3 away-notify capability* has been requested by a client, the server MAY also send that client AWAY messages to tell them how the away status of other users has changed.

Servers SHOULD notify clients when a user they're interacting with is away when relevant, including sending these numerics:

1. *RPL_AWAY* (301), with the away message, when a *PRIVMSG* command is directed at the away user (not to a channel they are on).

2. *RPL_AWAY* (301), with the away message, in replies to *WHOIS* messages.

3. In the *RPL_USERHOST* (302) numeric, as the + or - character.

4. In the *RPL_WHOREPLY* (352) numeric, as the H or G character.

Numeric Replies:

- *RPL_UNAWAY* (305)

- *RPL_NOWAWAY* (306)

10.7.2 LINKS message

```
     Command: LINKS
  Parameters: None
```

With LINKS, a user can list all servers which are known by the server answering the query, usually including the server itself.

In replying to the LINKS message, a server MUST send replies back using zero or more *RPL_LINKS* (364) messages and mark the end of the list using a *RPL_ENDOFLINKS* (365) message.

Servers MAY omit some or all servers on the network, including itself.

Numeric Replies:

- *RPL_LINKS* (364)

- *RPL_ENDOFLINKS* (365)

10.7.3   USERHOST message

```
    Command: USERHOST
 Parameters: <nickname>{ <nickname>}
```

The USERHOST command is used to return information about users with the given nicknames. The USERHOST command takes up to five nicknames, each a separate parameters. The nicknames are returned in *RPL_USERHOST* (302) numerics.

Numeric Replies:

- *ERR_NEEDMOREPARAMS* (461)

- *RPL_USERHOST* (302)

Command Examples:

```
 USERHOST Wiz Michael Marty p    ;USERHOST request for information on
                                 nicks "Wiz", "Michael", "Marty" and "p"
```

Reply Examples:

```
 :ircd.stealth.net 302 yournick :syrk=+syrk@millennium.stealth.net
                                ; Reply for user syrk
```

10.7.4   WALLOPS message

```
    Command: WALLOPS
 Parameters: <text>
```

The WALLOPS command is used to send a message to all currently connected users who have set the 'w' user mode for themselves. The <text> SHOULD be non-empty.

Servers MAY echo WALLOPS messages to their sender even if they don't have the 'w' user mode.

Servers MAY send WALLOPS only to operators.

Servers may generate it themselves, and MAY allow operators to send them.

Numeric replies:

- *ERR_NEEDMOREPARAMS* (461)

- *ERR_NOPRIVILEGES* (481)

- *ERR_NOPRIVS* (723)

Examples:

```
:csd.bu.edu WALLOPS :Connect '*.uiuc.edu 6667' from Joshua
                              ;WALLOPS message from csd.bu.edu announcin
                              a CONNECT message it received and acted
                              upon from Joshua.
```

☾

Appendix A. Channel Types

IRC has various types of channels that act in different ways. What differentiates these channels is the character the channel name starts with. For instance, channels starting with # are regular channels, and channels starting with & are local channels.

Upon joining, clients are shown which types of channels the server supports with the *CHANTYPES* parameter.

Here, we go through the different types of channels that exist and are widely-used these days.

Regular Channels (#)

The prefix character for this type of channel is (`'#', 0x23`).

This channel is what's referred to as a normal channel. Clients can join this channel, and the first client who joins a normal channel is made a *channel operator*, along with the appropriate channel membership prefix. On most servers, newly-created channels have then *protected topic* `"+t"` and *no external messages* `"+n"` modes enabled, but exactly what modes new channels are given is up to the server.

Regular channels are persisted across the network. If two clients on different servers join the same regular channel, they'll be able to see that each other are joined, and will see messages sent to the channel by the other client.

On servers that support the concept of 'channel ownership' (a client being able to own a channel and retain control of it with their account), clients may not receive channel operator priveledges on joining an otherwise empty channel.

Local Channels (&)

The prefix character for this type of channel is (`'&', 0x26`).

This channel is what's referred to as a local channel. Clients can join this channel as normal, and the first client who joins a normal channel is made a *channel operator*, but the channel is not persisted across the network. In other words, each server has its own set of local channels that the other servers on the network don't see.

If a client on server A and a client on server B join the channel `&info`, they will not be able to see each other or the messages each posts to their server's local channel `&info`. However, if a client on server A and another client on server A join the channel `&info`, they will be able to see each other and the messages the other posts to that local channel.

Generally, the concept of channel ownership is not supported for local channels. Local channels also aren't as widely available as regular channels. As well, some networks disable or disallow local channels as opers across the network can neither see nor

administrate them.

Appendix B. Modes

Modes affect the behaviour and reflect details about targets – clients and channels. The modes listed here are the ones that have been adopted and are used by the IRC community at large. If we say a mode is 'standard', that means it is defined in the official IRC specification documents.

The status and letter used for each mode is defined in the description of that mode.

We only cover modes that are widely-used by IRC software today and whose meanings should stay consistent between different server software. For more extensive lists (including conflicting and obsolete modes), see the external `irc-defs` *client* and *channel* mode lists.

10.1   User Modes

Invisible User Mode

This mode is standard, and the mode letter used for it is `"+i"`.

If a user is set to 'invisible', they will not show up in commands such as *WHO* or *NAMES* unless they share a channel with the user that submitted the command. In addition, some servers hide all channels from the *WHOIS* reply of an invisible user they do not share with the user that submitted the command.

Oper User Mode

This mode is standard, and the mode letter used for is it `"+o"`.

If a user has this mode, this indicates that they are a network *operator*.

Local Oper User Mode

This mode is standard, and the mode letter used for it is `"+O"`.

If a user has this mode, this indicates that they are a server *operator*. A local operator has *operator* privileges for their server, and not for the rest of the network.

Registered User Mode

This mode is widely-used, and the mode letter used for it is typically `"+r"`. The character used for this mode, and whether it exists at all, may vary depending on server software and configuration.

If a user has this mode, this indicates that they have logged into a user account.

IRCv3 extensions such as *account-notify*, *account-tag*, and *extended-join* provide the account name of logged-in users, and are more accurate than trying to detect this

user mode due to the capability name remaining consistent.

WALLOPS User Mode

This mode is standard, and the mode letter used for it is `"+w"`.

If a user has this mode, this indicates that they will receive *WALLOPS* messages from the server.

10.2   Channel Modes

Ban Channel Mode

This mode is standard, and the mode letter used for it is `"+b"`.

This channel mode controls a list of client masks that are 'banned' from joining or speaking in the channel. If this mode has values, each of these values should be a client mask.

If this mode is set on a channel, and a client sends a `JOIN` request for this channel, their nickmask (the combination of `nick!user@host`) is compared with each banned client mask set with this mode. If they match one of these banned masks, they will receive an *ERR_BANNEDFROMCHAN* `(474)` reply and the `JOIN` command will fail. See the *ban exception* mode for more details.

Exception Channel Mode

This mode is used in almost all IRC software today. The standard mode letter used for it is `"+e"`, but it SHOULD be defined in the *EXCEPTS* `RPL_ISUPPORT` parameter on connection.

This channel mode controls a list of client masks that are exempt from the *'ban'* channel mode. If this mode has values, each of these values should be a client mask.

If this mode is set on a channel, and a client sends a `JOIN` request for this channel, their nickmask is compared with each 'exempted' client mask. If their nickmask matches any one of the masks set by this mode, and their nickmask also matches any one of the masks set by the *ban* channel mode, they will not be blocked from joining due to the *ban* mode.

Client Limit Channel Mode

This mode is standard, and the mode letter used for it is `"+l"`.

This channel mode controls whether new users may join based on the number of users who already exist in the channel. If this mode is set, its value is an integer and defines the limit of how many clients may be joined to the channel.

If this mode is set on a channel, and the number of users joined to that channel matches

or exceeds the value of this mode, new users cannot join that channel. If a client sends a `JOIN` request for this channel, they will receive an `ERR_CHANNELISFULL` (471) reply and the command will fail.

Invite-Only Channel Mode

This mode is standard, and the mode letter used for it is `"+i"`.

This channel mode controls whether new users need to be invited to the channel before being able to join.

If this mode is set on a channel, a user must have received an `INVITE` for this channel before being allowed to join it. If they have not received an invite, they will receive an `ERR_INVITEONLYCHAN` (473) reply and the command will fail.

Invite-Exception Channel Mode

This mode is used in almost all IRC software today. The standard mode letter used for it is `"+I"`, but it SHOULD be defined in the `INVEX` RPL_ISUPPORT parameter on connection.

This channel mode controls a list of channel masks that are exempt from the *invite-only* channel mode. If this mode has values, each of these values should be a client mask.

If this mode is set on a channel, and a client sends a `JOIN` request for that channel, their nickmask is compared with each 'exempted' client mask. If their nickmask matches any one of the masks set by this mode, and the channel is in *invite-only* mode, they do not need to require an `INVITE` in order to join the channel.

Key Channel Mode

This mode is standard, and the mode letter used for it is `"+k"`.

This mode letter sets a 'key' that must be supplied in order to join this channel. If this mode is set, its' value is the key that is required. Servers may validate the value (eg. to forbid spaces, as they make it harder to use the key in `JOIN` messages). If the value is invalid, they SHOULD return `ERR_INVALIDMODEPARAM`. However, clients MUST be able to handle any of the following:

- *ERR_INVALIDMODEPARAM*

- *ERR_INVALIDKEY*

- `MODE` echoed with a different key (eg. truncated or stripped of invalid characters)

- the key changed ignored, and no `MODE` echoed if no other mode change was valid.

If this mode is set on a channel, and a client sends a `JOIN` request for that channel, they must supply <key> in order for the command to succeed. If they do not supply a <key>, or the key they supply does not match the value of this mode, they will receive an

ERR_BADCHANNELKEY (475) reply and the command will fail.

Moderated Channel Mode

This mode is standard, and the mode letter used for it is `"+m"`.

This channel mode controls whether users may freely talk on the channel, and does not have any value.

If this mode is set on a channel, only users who have channel privileges may send messages to that channel. The *voice* channel mode is designed to let a user talk in a moderated channel without giving them other channel moderation abilities, and users of higher privileges (such as *halfops* or *chanops*) may also speak in moderated channels.

Secret Channel Mode

This mode is standard, and the mode letter used for it is `"+s"`.

This channel mode controls whether the channel is 'secret', and does not have any value.

A channel that is set to secret will not show up in responses to the *LIST* or *NAMES* command unless the client sending the command is joined to the channel. Likewise, secret channels will not show up in the *RPL_WHOISCHANNELS* (319) numeric unless the user the numeric is being sent to is joined to that channel.

Protected Topic Mode

This mode is standard, and the mode letter used for it is `"+t"`.

This channel mode controls whether channel privileges are required to set the topic, and does not have any value.

If this mode is enabled, users must have channel privileges such as *halfop* or *operator* status in order to change the topic of a channel. In a channel that does not have this mode enabled, anyone may set the topic of the channel using the *TOPIC* command.

No External Messages Mode

This mode is standard, and the mode letter used for it is `"+n"`.

This channel mode controls whether users who are not joined to the channel can send messages to it, and does not have any value.

If this mode is enabled, users MUST be joined to the channel in order to send *private messages* and *notices* to the channel. If this mode is enabled and they try to send one of these to a channel they are not joined to, they will receive an *ERR_CANNOTSENDTOCHAN* (404) numeric and the message will not be sent to that channel.

10.3   Channel Membership Prefixes

Users joined to a channel may get certain privileges or status in that channel based on channel modes given to them. These users are given prefixes before their nickname whenever it is associated with a channel (ie, in *NAMES*, *WHO* and *WHOIS* messages). The standard and common prefixes are listed here, and MUST be advertised by the server in the *PREFIX* RPL_ISUPPORT parameter on connection.

Founder Prefix

This mode is used in a large number of networks. The prefix and mode letter typically used for it, respectively, are "~" and "+q".

This prefix shows that the given user is the 'founder' of the current channel and has full moderation control over it – ie, they are considered to 'own' that channel by the network. This prefix is typically only used on networks that have the concept of client accounts, and ownership of channels by those accounts.

Protected Prefix

This mode is used in a large number of networks. The prefix and mode letter typically used for it, respectively, are "&" and "+a".

Users with this mode cannot be kicked and cannot have this mode removed by other protected users. In some software, they may perform actions that operators can, but at a higher privilege level than operators. This prefix is typically only used on networks that have the concept of client accounts, and ownership of channels by those accounts.

Operator Prefix

This mode is standard. The prefix and mode letter used for it, respectively, are "@" and "+o".

Users with this mode may perform channel moderation tasks such as kicking users, applying channel modes, and set other users to operator (or lower) status.

Halfop Prefix

This mode is widely used in networks today. The prefix and mode letter used for it, respectively, are "%" and "+h".

Users with this mode may perform channel moderation tasks, but at a lower privilege level than operators. Which channel moderation tasks they can and cannot perform varies with server software and configuration.

Voice Prefix

This mode is standard. The prefix and mode letter used for it, respectively, are "+" and "+v".

Users with this mode may send messages to a channel that is *moderated*.

Appendix C. Numerics

As mentioned in the *numeric replies* section, the first parameter of most numerics is the target of that numeric (the nickname of the client that is receiving it). Underneath the name and numeric of each reply, we list the parameters sent by this message.

Clients MUST NOT fail because the number of parameters on a given incoming numeric is larger than the number of parameters we list for that numeric here. Most IRC servers extends some of these numerics with their own special additions. For example, if a message is listed here as having 2 parameters, and your client receives it with 5 parameters, your client should not fail to parse or handle that message correctly because of the extra parameters.

Optional parameters are surrounded with the standard square brackets (`[<optional>]`) – this means clients MUST NOT assume they will receive this parameter from all servers, and that servers SHOULD send this parameter unless otherwise specified in the numeric description. Parameters and parts of parameters surrounded with curly brackets (`{ <repeating>}`) may be repeated zero or more times.

Server authors that wish to extend one of the numerics listed here SHOULD make their extension into a *client capability*. If your extension would be useful to other client and server software, you should consider submitting it to the *IRCv3 Working Group* for standardisation.

Note that for numerics with "human-readable" informational strings for the last parameter which are not designed to be parsed, such as in `RPL_WELCOME`, servers commonly change this last-param text. Clients SHOULD NOT rely on these sort of parameters to have exactly the same human-readable string as described in this document. Clients that rely on the format of these human-readable final informational strings may fail. We do try to note numerics where this is the case with a message like *"The text used in the last param of this message varies wildly"*.

RPL_WELCOME (001)

```
"<client> :Welcome to the <networkname> Network, <nick>[!<user>@<host>]"
```

The first message sent after client registration, this message introduces the client to the network. The text used in the last param of this message varies wildly.

Servers that implement spoofed hostmasks in any capacity SHOULD NOT include the extended (complete) hostmask in the last parameter of this reply, either for all clients or for those whose hostnames have been spoofed. This is because some clients try to extract the hostname from this final parameter of this message and resolve this hostname, in order to discover their 'local IP address'.

Clients MUST NOT try to extract the hostname from the final parameter of this message and then attempt to resolve this hostname. This method of operation WILL BREAK and will cause issues when the server returns a spoofed hostname.

RPL_YOURHOST (002)

```
"<client> :Your host is <servername>, running version <version>"
```

Part of the post-registration greeting, this numeric returns the name and software/version of the server the client is currently connected to. The text used in the last param of this message varies wildly.

RPL_CREATED (003)

```
"<client> :This server was created <datetime>"
```

Part of the post-registration greeting, this numeric returns a human-readable date/time that the server was started or created. The text used in the last param of this message varies wildly.

RPL_MYINFO (004)

```
"<client> <servername> <version> <available user modes>
<available channel modes> [<channel modes with a parameter>]"
```

Part of the post-registration greeting. Clients SHOULD discover available features using RPL_ISUPPORT tokens rather than the mode letters listed in this reply.

RPL_ISUPPORT (005)

```
"<client> <1-13 tokens> :are supported by this server"
```

The ABNF representation for an RPL_ISUPPORT token is:

```
token     = *1"-" parameter / parameter *1( "=" value )
parameter = 1*20 letter
value     = * letpun
letter    = ALPHA / DIGIT
punct     = %d33-47 / %d58-64 / %d91-96 / %d123-126
letpun    = letter / punct
```

As the maximum number of message parameters to any reply is 15, the maximum number of RPL_ISUPPORT tokens that can be advertised is 13. To counter this, a server MAY issue multiple RPL_ISUPPORT numerics. A server MUST issue at least one

RPL_ISUPPORT numeric after client registration has completed. It MUST be issued before further commands from the client are processed.

When clients send a *VERSION* command to an external server (i.e. not the one they're currently connected to), they receive the appropriate information from that server. That external server's ISUPPORT tokens are sent to the client using the 105 (RPL_REMOTEISUPPORT) numeric instead of 005, to ensure that clients don't process and start using these tokens sent by an external server. The format of the 105 message is exactly the same as RPL_ISUPPORT – the numeric itself is the only difference.

A token is of the form PARAMETER, PARAMETER=VALUE or -PARAMETER. Servers MUST send the parameter as upper-case text.

Tokens of the form PARAMETER or PARAMETER=VALUE are used to advertise features or information to clients. A parameter MAY have a default value and value MAY be empty when sent by servers. Unless otherwise stated, when a parameter contains a value, the value MUST be treated as being case sensitive. The value MAY contain multiple fields, if this is the case the fields SHOULD be delimited with a comma character (",", 0x2C). The value MAY contain escape sequences: \x20 for the space character (" ", 0x20), \x5C for the backslash character ("\", 0x5C) and \x3D for the equal character ("=", 0x3D).

If the value of a parameter changes, the server SHOULD re-advertise the parameter with the new value in an RPL_ISUPPORT reply. An example of this is a client becoming an *IRC operator* and their *CHANLIMIT* changing.

Tokens of the form -PARAMETER are used to negate a previously specified parameter. If the client receives a token like this, the client MUST consider that parameter to be removed and revert to the behaviour that would occur if the parameter was not specified. The client MUST act as though the paramater is no longer advertised to it. These tokens are intended to allow servers to change their features without disconnecting clients. Tokens of this form MUST NOT contain a value field.

The server MAY negate parameters which have not been previously advertised; in this case, the client MUST ignore the token.

A single RPL_ISUPPORT reply MUST NOT contain the same parameter multiple times nor advertise and negate the same parameter. However, the server is free to advertise or negate the same parameter in separate replies.

See the *Feature Advertisement* section for more details on this numeric. A list of parameters is available in the *RPL_ISUPPORT Parameters* section.

RPL_BOUNCE (010)

```
"<client> <hostname> <port> :<info>"
```

Sent to the client to redirect it to another server. The `<info>` text varies between server software and reasons for the redirection.

Because this numeric does not specify whether to enable SSL and is not interpreted correctly by all clients, it is recommended that this not be used.

This numeric is also known as RPL_REDIR by some software.

RPL_STATSCOMMANDS (212)

```
"<client> <command> <count> [<byte count> <remote count>]"
```

Sent as a reply to the *STATS* command, when a client requests statistics on command usage.

`<byte count>` and `<remote count>` are optional and MAY be included in responses.

RPL_ENDOFSTATS (219)

```
"<client> <stats letter> :End of /STATS report"
```

Indicates the end of a STATS response.

RPL_UMODEIS (221)

```
"<client> <user modes>"
```

Sent to a client to inform that client of their currently-set user modes.

RPL_STATSUPTIME (242)

```
"<client> :Server Up <days> days <hours>:<minutes>:<seconds>"
```

Sent as a reply to the *STATS* command, when a client requests the server uptime. The text used in the last param of this message may vary.

RPL_LUSERCLIENT (251)

```
"<client> :There are <u> users and <i> invisible on <s> servers"
```

Sent as a reply to the *LUSERS* command. `<u>`, `<i>`, and `<s>` are non-negative integers, and represent the number of total users, invisible users, and other servers connected to this server.

RPL_LUSEROP (252)

```
"<client> <ops> :operator(s) online"
```

Sent as a reply to the *LUSERS* command. `<ops>` is a positive integer and represents the number of *IRC operators* connected to this server. The text used in the last param of this message may vary.

RPL_LUSERUNKNOWN (253)

```
"<client> <connections> :unknown connection(s)"
```

Sent as a reply to the *LUSERS* command. `<connections>` is a positive integer and represents the number of connections to this server that are currently in an unknown state. The text used in the last param of this message may vary.

RPL_LUSERCHANNELS (254)

```
"<client> <channels> :channels formed"
```

Sent as a reply to the *LUSERS* command. `<channels>` is a positive integer and represents the number of channels that currently exist on this server. The text used in the last param of this message may vary.

RPL_LUSERME (255)

```
"<client> :I have <c> clients and <s> servers"
```

Sent as a reply to the *LUSERS* command. `<c>` and `<s>` are non-negative integers and represent the number of clients and other servers connected to this server, respectively.

RPL_ADMINME (256)

```
"<client> [<server>] :Administrative info"
```

Sent as a reply to an *ADMIN* command, this numeric establishes the name of the server whose administrative info is being provided. The text used in the last param of this message may vary.

`<server>` is optional and MAY be included in responses, the server can also be gained from the `<source>` of this message.

RPL_ADMINLOC1 (257)

```
    "<client> :<info>"
```

Sent as a reply to an *ADMIN* command, `<info>` is a string intended to provide information about the location of the server (i.e. city, state and country). The text used in the last param of this message varies wildly.

RPL_ADMINLOC2 (258)

```
    "<client> :<info>"
```

Sent as a reply to an *ADMIN* command, `<info>` is a string intended to provide information about whoever runs the server (i.e. details of the institution hosting it). The text used in the last param of this message varies wildly.

RPL_ADMINEMAIL (259)

```
    "<client> :<info>"
```

Sent as a reply to an *ADMIN* command, `<info>` MUST contain the email address to contact the administrator(s) of the server. The text used in the last param of this message varies wildly.

RPL_TRYAGAIN (263)

```
    "<client> <command> :Please wait a while and try again."
```

When a server drops a command without processing it, this numeric MUST be sent to inform the client. The text used in the last param of this message varies wildly, and commonly provides the client with more information about why the command could not be processed (i.e., due to rate-limiting).

RPL_LOCALUSERS (265)

```
    "<client> [<u> <m>] :Current local users <u>, max <m>"
```

Sent as a reply to the *LUSERS* command. <u> and <m> are non-negative integers and represent the number of clients currently and the maximum number of clients that have been connected directly to this server at one time, respectively.

The two optional parameters SHOULD be supplied to allow clients to better extract these numbers.

☾

RPL_GLOBALUSERS (266)

```
"<client> [<u> <m>] :Current global users <u>, max <m>"
```

Sent as a reply to the *LUSERS* command. <u> and <m> are non-negative integers. <u> represents the number of clients currently connected to this server, globally (directly and through other server links). <m> represents the maximum number of clients that have been connected to this server at one time, globally.

The two optional parameters SHOULD be supplied to allow clients to better extract these numbers.

RPL_WHOISCERTFP (276)

```
"<client> <nick> :has client certificate fingerprint <fingerprint>"
```

Sent as a reply to the *WHOIS* command, this numeric shows the SSL/TLS certificate fingerprint used by the client with the nickname <nick>. Clients MUST only be sent this numeric if they are either using the WHOIS command on themselves or they are an *operator*.

RPL_NONE (300)

```
Undefined format
```

RPL_NONE is a dummy numeric. It does not have a defined use nor format.

RPL_AWAY (301)

```
"<client> <nick> :<message>"
```

Indicates that the user with the nickname <nick> is currently away and sends the away message that they set.

RPL_USERHOST (302)

```
"<client> :[<reply>{ <reply>}]"
```

Sent as a reply to the *USERHOST* command, this numeric lists nicknames and the information associated with them. The last parameter of this numeric (if there are any results) is a list of <reply> values, delimited by a SPACE character (' ', 0x20).

The ABNF representation for <reply> is:

```
    reply   =   nickname [ isop ] "=" isaway hostname
    isop    =   "*"
    isaway  =   ( "+" / "-" )
```

`<isop>` is included if the user with the nickname of `<nickname>` has registered as an *operator*. `<isaway>` represents whether that user has set an [away] message. `"+"` represents that the user is not away, and `"-"` represents that the user is away.

RPL_UNAWAY (305)

```
    "<client> :You are no longer marked as being away"
```

Sent as a reply to the *AWAY* command, this lets the client know that they are no longer set as being away. The text used in the last param of this message may vary.

RPL_NOWAWAY (306)

```
    "<client> :You have been marked as being away"
```

Sent as a reply to the *AWAY* command, this lets the client know that they are set as being away. The text used in the last param of this message may vary.

RPL_WHOISREGNICK (307)

```
    "<client> <nick> :has identified for this nick"
```

Sent as a reply to the *WHOIS* command, this numeric indicates that the client with the nickname `<nick>` was authenticated as the owner of this nick on the network.

See also *RPL_WHOISACCOUNT*, for information on the account name of the user.

RPL_WHOISUSER (311)

```
    "<client> <nick> <username> <host> * :<realname>"
```

Sent as a reply to the *WHOIS* command, this numeric shows details about the client with the nickname `<nick>`. `<username>` and `<realname>` represent the names set by the *USER* command (though `<username>` may be set by the server in other ways). `<host>` represents the host used for the client in nickmasks (which may or may not be a real hostname or IP address). `<host>` CANNOT start with a colon (`':'`, `0x3A`) as this would get parsed as a trailing parameter – IPv6 addresses such as `"::1"` are prefixed with a zero (`'0'`, `0x30`) to ensure this. The second-last parameter is a literal asterisk character (`'*'`, `0x2A`) and does not mean anything.

RPL_WHOISSERVER (312)

```
"<client> <nick> <server> :<server info>"
```

Sent as a reply to the *WHOIS* (or *WHOWAS*) command, this numeric shows which server the client with the nickname `<nick>` is (or was) connected to. `<server>` is the name of the server (as used in message prefixes). `<server info>` is a string containing a description of that server.

RPL_WHOISOPERATOR (313)

```
"<client> <nick> :is an IRC operator"
```

Sent as a reply to the *WHOIS* command, this numeric indicates that the client with the nickname `<nick>` is an *operator*. This command MAY also indicate what type or level of operator the client is by changing the text in the last parameter of this numeric. The text used in the last param of this message varies wildly, and SHOULD be displayed as-is by IRC clients to their users.

RPL_WHOWASUSER (314)

```
"<client> <nick> <username> <host> * :<realname>"
```

Sent as a reply to the *WHOWAS* command, this numeric shows details about one of the last clients that used the nickname `<nick>`. The purpose of each argument is the same as with the *RPL_WHOISUSER* (311) numeric.

RPL_ENDOFWHO (315)

```
"<client> <mask> :End of WHO list"
```

Sent as a reply to the *WHO* command, this numeric indicates the end of a WHO response for the mask `<mask>`.

`<mask>` MUST be the same `<mask>` parameter sent by the client in its WHO message, but MAY be casefolded.

This numeric is sent after all other WHO response numerics have been sent to the client.

RPL_WHOISIDLE (317)

```
"<client> <nick> <secs> <signon> :seconds idle, signon time"
```

Sent as a reply to the *WHOIS* command, this numeric indicates how long the client with the nickname `<nick>` has been idle. `<secs>` is the number of seconds since the client has been active. Servers generally denote specific commands (for instance, perhaps *JOIN*, *PRIVMSG*, *NOTICE*, etc) as updating the 'idle time', and calculate this off when the idle time was last updated. `<signon>` is a unix timestamp representing when the user joined the network. The text used in the last param of this message may vary.

RPL_ENDOFWHOIS (318)

```
"<client> <nick> :End of /WHOIS list"
```

Sent as a reply to the *WHOIS* command, this numeric indicates the end of a `WHOIS` response for the client with the nickname `<nick>`.

`<nick>` MUST be exactly the `<nick>` parameter sent by the client in its `WHOIS` message. This means the case MUST be preserved, and if the client sent multiple nicks, this MUST be the comma-separated list of nicks, even if some of them were dropped.

This numeric is sent after all other `WHOIS` response numerics have been sent to the client.

RPL_WHOISCHANNELS (319)

```
"<client> <nick> :[prefix]<channel>{ [prefix]<channel>}
```

Sent as a reply to the *WHOIS* command, this numeric lists the channels that the client with the nickname `<nick>` is joined to and their status in these channels. `<prefix>` is the highest *channel membership prefix* that the client has in that channel, if the client has one. `<channel>` is the name of a channel that the client is joined to. The last parameter of this numeric is a list of `[prefix]<channel>` pairs, delimited by a SPACE character (`' '`, `0x20`). Clients MUST ignore the trailing SPACE character, if any.

`RPL_WHOISCHANNELS` can be sent multiple times in the same whois reply, if the target is on too many channels to fit in a single message.

The channels in this response are affected by the *secret* channel mode and the *invisible* user mode, and may be affected by other modes depending on server software and configuration.

RPL_WHOISSPECIAL (320)

```
"<client> <nick> :blah blah blah"
```

Sent as a reply to the *WHOIS* command, this numeric is used for extra human-readable information on the client with nickname `<nick>`. This should only be used for non-

essential information that does not need to be machine-readable or understood by client software.

RPL_LISTSTART (321)

```
"<client> Channel :Users  Name"
```

Sent as a reply to the *LIST* command, this numeric marks the start of a channel list. As noted in the command description, this numeric MAY be skipped by the server so clients MUST NOT depend on receiving it.

RPL_LIST (322)

```
"<client> <channel> <client count> :<topic>"
```

Sent as a reply to the *LIST* command, this numeric sends information about a channel to the client. `<channel>` is the name of the channel. `<client count>` is an integer indicating how many clients are joined to that channel. `<topic>` is the channel's topic (as set by the *TOPIC* command).

RPL_LISTEND (323)

```
"<client> :End of /LIST"
```

Sent as a reply to the *LIST* command, this numeric indicates the end of a LIST response.

RPL_CHANNELMODEIS (324)

```
"<client> <channel> <modestring> <mode arguments>..."
```

Sent to a client to inform them of the currently-set modes of a channel. `<channel>` is the name of the channel. `<modestring>` and `<mode arguments>` are a mode string and the mode arguments (delimited as separate parameters) as defined in the *MODE* message description.

RPL_CREATIONTIME (329)

```
"<client> <channel> <creationtime>"
```

Sent to a client to inform them of the creation time of a channel. `<channel>` is the name of the channel. `<creationtime>` is a unix timestamp representing when the channel was created on the network.

RPL_WHOISACCOUNT (330)

```
"<client> <nick> <account> :is logged in as"
```

Sent as a reply to the *WHOIS* command, this numeric indicates that the client with the nickname `<nick>` was authenticated as the owner of `<account>`.

This does not necessarily mean the user owns their current nickname, which is covered by *RPL_WHOISREGNICK*.

RPL_NOTOPIC (331)

```
"<client> <channel> :No topic is set"
```

Sent to a client when joining a channel to inform them that the channel with the name `<channel>` does not have any topic set.

RPL_TOPIC (332)

```
"<client> <channel> :<topic>"
```

Sent to a client when joining the `<channel>` to inform them of the current *topic* of the channel.

RPL_TOPICWHOTIME (333)

```
"<client> <channel> <nick> <setat>"
```

Sent to a client to let them know who set the topic (`<nick>`) and when they set it (`<setat>` is a unix timestamp). Sent after *RPL_TOPIC* (332).

RPL_INVITELIST (336)

```
"<client> <channel>"
```

Sent to a client as a reply to the *INVITE* command when used with no parameter, to indicate a channel the client was invited to.

This numeric should not be confused with *RPL_INVEXLIST* (346), which is used as a reply to *MODE*.

☾

> Some rare implementations use 346 instead of 336 for this reply.

RPL_ENDOFINVITELIST (337)

```
"<client> :End of /INVITE list"
```

Sent as a reply to the *INVITE* command when used with no parameter, this numeric indicates the end of invitations a client received.

This numeric should not be confused with *RPL_ENDOFINVEXLIST* (347), which is used as a reply to *MODE*.

> Some rare implementations use 347 instead of 337 for this reply.

RPL_WHOISACTUALLY (338)

```
"<client> <nick> :is actually ..."
"<client> <nick> <host|ip> :Is actually using host"
"<client> <nick> <username>@<hostname> <ip> :Is actually using host"
```

Sent as a reply to the *WHOIS* and *WHOWAS* commands, this numeric shows details about the client with the nickname `<nick>`.

`<username>` represents the name set by the *USER* command (though `<username>` may be set by the server in other ways).

`<host>` and `<ip>` represent the real host and IP address the client is connecting from. `<host>` CANNOT start with a colon (`':', 0x3A`) as this would get parsed as a trailing parameter – IPv6 addresses such as `"::1"` are prefixed with a zero (`'0', 0x30`) to ensure this. The resulting IPv6 is equivalent, as this is a partial expansion of the `::` shorthand.

See also: *RPL_WHOISHOST* (378), for similar semantics on other servers.

RPL_INVITING (341)

```
"<client> <nick> <channel>"
```

Sent as a reply to the *INVITE* command to indicate that the attempt was successful and the client with the nickname `<nick>` has been invited to `<channel>`.

RPL_INVEXLIST (346)

```
"<client> <channel> <mask>"
```

Sent as a reply to the *MODE* command, when clients are viewing the current entries on a channel's *invite-exception list*. `<mask>` is the given mask on the invite-exception list.

This numeric should not be confused with *RPL_INVITELIST* (336), which is used as a reply to *INVITE*.

This numeric is sometimes erroneously called RPL_INVITELIST, as this was the name used in RFC2812.

RPL_ENDOFINVEXLIST (347)

```
"<client> <channel> :End of Channel Invite Exception List"
```

Sent as a reply to the *MODE* command, this numeric indicates the end of a channel's *invite-exception list*.

This numeric should not be confused with *RPL_ENDOFINVITELIST* (337), which is used as a reply to *INVITE*.

This numeric is sometimes erroneously called RPL_ENDOFINVITELIST, as this was the name used in RFC2812.

RPL_EXCEPTLIST (348)

```
"<client> <channel> <mask>"
```

Sent as a reply to the *MODE* command, when clients are viewing the current entries on a channel's *exception list*. `<mask>` is the given mask on the exception list.

RPL_ENDOFEXCEPTLIST (349)

```
"<client> <channel> :End of channel exception list"
```

Sent as a reply to the *MODE* command, this numeric indicates the end of a channel's *exception list*.

RPL_VERSION (351)

```
"<client> <version> <server> :<comments>"
```

Sent as a reply to the *VERSION* command, this numeric indicates information about the desired server. `<version>` is the name and version of the software being used (including any revision information). `<server>` is the name of the server. `<comments>` may contain any further comments or details about the specific version of the server.

RPL_WHOREPLY (352)

```
"<client> <channel> <username> <host> <server> <nick> <flags> :<hopcount>
```

Sent as a reply to the *WHO* command, this numeric gives information about the client with the nickname `<nick>`. Refer to *RPL_WHOISUSER* (311) for the meaning of the fields `<username>`, `<host>` and `<realname>`. `<server>` is the name of the server the client is connected to. If the *WHO* command was given a channel as the `<mask>` parameter, then the same channel MUST be returned in `<channel>`. Otherwise `<channel>` is an arbitrary channel the client is joined to or a literal asterisk character (`'*'`, `0x2A`) if no channel is returned. `<hopcount>` is the number of intermediate servers between the client issuing the `WHO` command and the client `<nick>`, it might be unreliable so clients SHOULD ignore it.

`<flags>` contains the following characters, in this order:

- Away status: the letter H (`'H'`, `0x48`) to indicate that the user is here, or the letter G (`'G'`, `0x47`) to indicate that the user is gone.

- Optionally, a literal asterisk character (`'*'`, `0x2A`) to indicate that the user is a server operator.

- Optionally, the highest *channel membership prefix* that the client has in `<channel>`, if the client has one.

- Optionally, one or more user mode characters and other arbitrary server-specific flags.

RPL_NAMREPLY (353)

```
"<client> <symbol> <channel> :[prefix]<nick>{ [prefix]<nick>}"
```

Sent as a reply to the *NAMES* command, this numeric lists the clients that are joined to `<channel>` and their status in that channel.

`<symbol>` notes the status of the channel. It can be one of the following:

- (`"="`, `0x3D`) - Public channel.

- (`"@"`, `0x40`) - Secret channel (*secret channel mode* `"+s"`).

- (`"*"`, `0x2A`) - Private channel (was `"+p"`, no longer widely used today).

`<nick>` is the nickname of a client joined to that channel, and `<prefix>` is the highest *channel membership prefix* that client has in the channel, if they have one. The last parameter of this numeric is a list of `[prefix]<nick>` pairs, delimited by a SPACE character (`' '`, `0x20`).

RPL_LINKS (364)

```
"<client> * <server> :<hopcount> <server info>"
```

Sent as a reply to the *LINKS* command, this numeric specifies one of the known servers on the network.

`<server info>` is a string containing a description of that server.

RPL_ENDOFLINKS (365)

```
"<client> * :End of /LINKS list"
```

Sent as a reply to the *LINKS* command, this numeric specifies the end of a list of channel member names.

RPL_ENDOFNAMES (366)

```
"<client> <channel> :End of /NAMES list"
```

Sent as a reply to the *NAMES* command, this numeric specifies the end of a list of channel member names.

RPL_BANLIST (367)

```
"<client> <channel> <mask> [<who> <set-ts>]"
```

Sent as a reply to the *MODE* command, when clients are viewing the current entries on a channel's *ban list*. `<mask>` is the given mask on the ban list.

`<who>` and `<set-ts>` are optional and MAY be included in responses. `<who>` is either the nickname or nickmask of the client that set the ban, or a server name, and `<set-ts>` is the UNIX timestamp of when the ban was set.

RPL_ENDOFBANLIST (368)

```
"<client> <channel> :End of channel ban list"
```

Sent as a reply to the *MODE* command, this numeric indicates the end of a channel's *ban list*.

RPL_ENDOFWHOWAS (369)

```
"<client> <nick> :End of WHOWAS"
```

Sent as a reply to the *WHOWAS* command, this numeric indicates the end of a WHOWAS reponse for the nickname `<nick>`. This numeric is sent after all other WHOWAS response numerics have been sent to the client.

RPL_INFO (371)

```
"<client> :<string>"
```

Sent as a reply to the *INFO* command, this numeric returns human-readable information describing the server: e.g. its version, list of authors and contributors, and any other miscellaneous information which may be considered to be relevant.

RPL_MOTD (372)

```
"<client> :<line of the motd>"
```

When sending the *Message of the Day* to the client, servers reply with each line of the MOTD as this numeric. MOTD lines MAY be wrapped to 80 characters by the server.

RPL_ENDOFINFO (374)

```
"<client> :End of INFO list"
```

Indicates the end of an INFO response.

RPL_MOTDSTART (375)

```
"<client> :- <server> Message of the day - "
```

Indicates the start of the *Message of the Day* to the client. The text used in the last param of this message may vary, and SHOULD be displayed as-is by IRC clients to their users.

RPL_ENDOFMOTD (376)

```
🌙  "<client> :End of /MOTD command."
```

Indicates the end of the *Message of the Day* to the client. The text used in the last param of this message may vary.

RPL_WHOISHOST (378)

```
"<client> <nick> :is connecting from *@localhost 127.0.0.1"
```

Sent as a reply to the *WHOIS* command, this numeric shows details about where the client with nickname `<nick>` is connecting from.

See also: *RPL_WHOISACTUALLY* (338), for similar semantics on other servers.

RPL_WHOISMODES (379)

```
"<client> <nick> :is using modes +ailosw"
```

Sent as a reply to the *WHOIS* command, this numeric shows the client what user modes the target users has.

RPL_YOUREOPER (381)

```
"<client> :You are now an IRC operator"
```

Sent to a client which has just successfully issued an *OPER* command and gained *operator* status. The text used in the last param of this message varies wildly.

RPL_REHASHING (382)

```
"<client> <config file> :Rehashing"
```

Sent to an *operator* which has just successfully issued a *REHASH* command. The text used in the last param of this message may vary.

RPL_TIME (391)

```
"<client> <server> [<timestamp> [<TS offset>]] :<human-readable time>"
```

Reply to the *TIME* command. Typically only contains the human-readable time, but it may include a UNIX timestamp.

Clients SHOULD NOT parse the human-readable time.

`<TS offset>` is used by some servers using a TS-based server-to-server protocol (eg.

TS6), and represents the offset between the server's system time, and the TS of the network. A positive value means the server is lagging behind the TS of the network. Clients SHOULD ignore its value.

ERR_UNKNOWNERROR (400)

```
"<client> <command>{ <subcommand>} :<info>"
```

Indicates that the given command/subcommand could not be processed. `<subcommand>` may repeat for more specific subcommands.

For example, for an issue with a hypothetical command PACK, this may be returned:

```
:example.com 400 dan!~d@n PACK :Could not process multiple invalid parame
```

For an issue with a hypothetical command PACK with the subcommand BOX, this may be returned:

```
:example.com 400 dan!~d@n PACK BOX :Could not find box to pack
```

This numeric indicates a very generalised error (which `<info>` should further explain). If there is another more specific numeric which represents the error occuring, that should be used instead.

ERR_NOSUCHNICK (401)

```
"<client> <nickname> :No such nick/channel"
```

Indicates that no client can be found for the supplied nickname. The text used in the last param of this message may vary.

ERR_NOSUCHSERVER (402)

```
"<client> <server name> :No such server"
```

Indicates that the given server name does not exist. The text used in the last param of this message may vary.

ERR_NOSUCHCHANNEL (403)

```
"<client> <channel> :No such channel"
```

Indicates that no channel can be found for the supplied channel name. The text used in the last param of this message may vary.

ERR_CANNOTSENDTOCHAN (404)

```
    "<client> <channel> :Cannot send to channel"
```

Indicates that the PRIVMSG / NOTICE could not be delivered to <channel>. The text used in the last param of this message may vary.

This is generally sent in response to channel modes, such as a channel being *moderated* and the client not having permission to speak on the channel, or not being joined to a channel with the *no external messages* mode set.

ERR_TOOMANYCHANNELS (405)

```
    "<client> <channel> :You have joined too many channels"
```

Indicates that the JOIN command failed because the client has joined their maximum number of channels. The text used in the last param of this message may vary.

ERR_WASNOSUCHNICK (406)

```
    "<client> <nickname> :There was no such nickname"
```

Returned as a reply to *WHOWAS* to indicate there is no history information for that nickname.

ERR_NOORIGIN (409)

```
    "<client> :No origin specified"
```

Indicates a PING or PONG message missing the originator parameter which is required by old IRC servers. Nowadays, this may be used by some servers when the PING <token> is empty.

ERR_NORECIPIENT (411)

```
    "<client> :No recipient given (<command>)"
```

Returned by the *PRIVMSG* command to indicate the message wasn't delivered because there was no recipient given.

ERR_NOTEXTTOSEND (412)

```
"<client> :No text to send"
```

Returned by the *PRIVMSG* command to indicate the message wasn't delivered because there was no text to send.

ERR_INPUTTOOLONG (417)

```
"<client> :Input line was too long"
```

Indicates a given line does not follow the specified size limits (512 bytes for the main section, 4094 or 8191 bytes for the tag section).

ERR_UNKNOWNCOMMAND (421)

```
"<client> <command> :Unknown command"
```

Sent to a registered client to indicate that the command they sent isn't known by the server. The text used in the last param of this message may vary.

ERR_NOMOTD (422)

```
"<client> :MOTD File is missing"
```

Indicates that the *Message of the Day* file does not exist or could not be found. The text used in the last param of this message may vary.

ERR_NONICKNAMEGIVEN (431)

```
"<client> :No nickname given"
```

Returned when a nickname parameter is expected for a command but isn't given.

ERR_ERRONEUSNICKNAME (432)

```
"<client> <nick> :Erroneus nickname"
```

Returned when a *NICK* command cannot be successfully completed as the desired nickname contains characters that are disallowed by the server. See the *NICK command* for more information on characters which are allowed in various IRC servers. The text used in the last param of this message may vary.

ERR_NICKNAMEINUSE (433)

```
"<client> <nick> :Nickname is already in use"
```

Returned when a *NICK* command cannot be successfully completed as the desired nickname is already in use on the network. The text used in the last param of this message may vary.

ERR_NICKCOLLISION (436)

```
"<client> <nick> :Nickname collision KILL from <user>@<host>"
```

Returned by a server to a client when it detects a nickname collision (registered of a NICK that already exists by another server). The text used in the last param of this message may vary.

ERR_USERNOTINCHANNEL (441)

```
"<client> <nick> <channel> :They aren't on that channel"
```

Returned when a client tries to perform a channel+nick affecting command, when the nick isn't joined to the channel (for example, MODE #channel +o nick).

ERR_NOTONCHANNEL (442)

```
"<client> <channel> :You're not on that channel"
```

Returned when a client tries to perform a channel-affecting command on a channel which the client isn't a part of.

ERR_USERONCHANNEL (443)

```
"<client> <nick> <channel> :is already on channel"
```

Returned when a client tries to invite <nick> to a channel they're already joined to.

ERR_NOTREGISTERED (451)

```
"<client> :You have not registered"
```

Returned when a client command cannot be parsed as they are not yet registered. Servers offer only a limited subset of commands until clients are properly registered to

the server. The text used in the last param of this message may vary.

ERR_NEEDMOREPARAMS (461)

```
"<client> <command> :Not enough parameters"
```

Returned when a client command cannot be parsed because not enough parameters were supplied. The text used in the last param of this message may vary.

ERR_ALREADYREGISTERED (462)

```
"<client> :You may not reregister"
```

Returned when a client tries to change a detail that can only be set during registration (such as resending the *PASS* or *USER* after registration). The text used in the last param of this message varies.

ERR_PASSWDMISMATCH (464)

```
"<client> :Password incorrect"
```

Returned to indicate that the connection could not be registered as the *password* was either incorrect or not supplied. The text used in the last param of this message may vary.

ERR_YOUREBANNEDCREEP (465)

```
"<client> :You are banned from this server."
```

Returned to indicate that the server has been configured to explicitly deny connections from this client. The text used in the last param of this message varies wildly and typically also contains the reason for the ban and/or ban details, and SHOULD be displayed as-is by IRC clients to their users.

ERR_CHANNELISFULL (471)

```
"<client> <channel> :Cannot join channel (+l)"
```

Returned to indicate that a *JOIN* command failed because the *client limit* mode has been set and the maximum number of users are already joined to the channel. The text used in the last param of this message may vary.

🌙

ERR_UNKNOWNMODE (472)

```
"<client> <modechar> :is unknown mode char to me"
```

Indicates that a mode character used by a client is not recognized by the server. The text used in the last param of this message may vary.

ERR_INVITEONLYCHAN (473)

```
"<client> <channel> :Cannot join channel (+i)"
```

Returned to indicate that a *JOIN* command failed because the channel is set to [invite-only] mode and the client has not been *invited* to the channel or had an *invite exception* set for them. The text used in the last param of this message may vary.

ERR_BANNEDFROMCHAN (474)

```
"<client> <channel> :Cannot join channel (+b)"
```

Returned to indicate that a *JOIN* command failed because the client has been *banned* from the channel and has not had a *ban exception* set for them. The text used in the last param of this message may vary.

ERR_BADCHANNELKEY (475)

```
"<client> <channel> :Cannot join channel (+k)"
```

Returned to indicate that a *JOIN* command failed because the channel requires a *key* and the key was either incorrect or not supplied. The text used in the last param of this message may vary.

Not to be confused with *ERR_INVALIDKEY*, which may be returned when setting a key.

ERR_BADCHANMASK (476)

```
"<channel> :Bad Channel Mask"
```

Indicates the supplied channel name is not a valid.

This is similar to, but stronger than, *ERR_NOSUCHCHANNEL* (403), which indicates that the channel does not exist, but that it may be a valid name.

The text used in the last param of this message may vary.

ERR_NOPRIVILEGES (481)

```
"<client> :Permission Denied- You're not an IRC operator"
```

Indicates that the command failed because the user is not an *IRC operator*. The text used in the last param of this message may vary.

ERR_CHANOPRIVSNEEDED (482)

```
"<client> <channel> :You're not channel operator"
```

Indicates that a command failed because the client does not have the appropriate *channel privileges*. This numeric can apply for different prefixes such as *halfop*, *operator*, etc. The text used in the last param of this message may vary.

ERR_CANTKILLSERVER (483)

```
"<client> :You cant kill a server!"
```

Indicates that a *KILL* command failed because the user tried to kill a server. The text used in the last param of this message may vary.

ERR_NOOPERHOST (491)

```
"<client> :No O-lines for your host"
```

Indicates that an *OPER* command failed because the server has not been configured to allow connections from this client's host to become an operator. The text used in the last param of this message may vary.

ERR_UMODEUNKNOWNFLAG (501)

```
"<client> :Unknown MODE flag"
```

Indicates that a *MODE* command affecting a user contained a MODE letter that was not recognized. The text used in the last param of this message may vary.

ERR_USERSDONTMATCH (502)

```
"<client> :Cant change mode for other users"
```

Indicates that a *MODE* command affecting a user failed because they were trying to set or

view modes for other users. The text used in the last param of this message varies, for instance when trying to view modes for another user, a server may send: `"Can't view modes for other users"`.

ERR_HELPNOTFOUND (524)

```
    "<client> <subject> :No help available on this topic"
```

Indicates that a *HELP* command requested help on a subject the server does not know about.

The `<subject>` MUST be the one requested by the client, but may be casefolded; unless it would be an invalid parameter, in which case it MUST be *.

ERR_INVALIDKEY (525)

```
  "<client> <target chan> :Key is not well-formed"
```

Indicates the value of a key channel mode change (+k) was rejected.

Not to be confused with *ERR_BADCHANNELKEY*, which is returned when someone tries to join a channel.

RPL_STARTTLS (670)

```
    "<client> :STARTTLS successful, proceed with TLS handshake"
```

This numeric is used by the IRCv3 *tls* extension and indicates that the client may begin a TLS handshake. For more information on this numeric, see the linked IRCv3 specification.

The text used in the last param of this message varies wildly.

RPL_WHOISSECURE (671)

```
    "<client> <nick> :is using a secure connection"
```

Sent as a reply to the *WHOIS* command, this numeric shows the client is connecting to the server in a way the server considers reasonably safe from eavesdropping (e.g. connecting from localhost, using TLS, using Tor).

ERR_STARTTLS (691)

```
    "<client> :STARTTLS failed (Wrong moon phase)"
```

This numeric is used by the IRCv3 *tls* extension and indicates that a server-side error occured and the STARTTLS command failed. For more information on this numeric, see the linked IRCv3 specification.

The text used in the last param of this message varies wildly.

ERR_INVALIDMODEPARAM (696)

```
"<client> <target chan/user> <mode char> <parameter> :<description>"
```

Indicates that there was a problem with a mode parameter. Replaces various implementation-specific mode-specific numerics.

RPL_HELPSTART (704)

```
"<client> <subject> :<first line of help section>"
```

Indicates the start of a reply to a *HELP* command. The text used in the last parameter of this message may vary, and SHOULD be displayed as-is by IRC clients to their users; possibly emphasized as the title of the help section.

The `<subject>` MUST be the one requested by the client, but may be casefolded; unless it would be an invalid parameter, in which case it MUST be *.

RPL_HELPTXT (705)

```
"<client> <subject> :<line of help text>"
```

Returns a line of *HELP* text to the client. Lines MAY be wrapped to a certain line length by the server. Note that the final line MUST be a *RPL_ENDOFHELP* (706) numeric.

The `<subject>` MUST be the one requested by the client, but may be casefolded; unless it would be an invalid parameter, in which case it MUST be *.

RPL_ENDOFHELP (706)

```
"<client> <subject> :<last line of help text>"
```

Returns the final *HELP* line to the client.

The `<subject>` MUST be the one requested by the client, but may be casefolded; unless it would be an invalid parameter, in which case it MUST be *.

ERR_NOPRIVS (723)

```
    "<client> <priv> :Insufficient oper privileges."
```

Sent by a server to alert an IRC *operator* that they they do not have the specific operator privilege required by this server/network to perform the command or action they requested. The text used in the last param of this message may vary.

`<priv>` is a string that has meaning in the server software, and allows an operator the privileges to perform certain commands or actions. These strings are server-defined and may refer to one or multiple commands or actions that may be performed by IRC operators.

Examples of the sorts of privilege strings used by server software today include: `kline`, `dline`, `unkline`, `kill`, `kill:remote`, `die`, `remoteban`, `connect`, `connect:remote`, `rehash`.

RPL_LOGGEDIN (900)

```
    "<client> <nick>!<user>@<host> <account> :You are now logged in as <usern
```

This numeric indicates that the client was logged into the specified account (whether by *SASL authentication* or otherwise). For more information on this numeric, see the IRCv3 *sasl-3.1* extension.

The text used in the last param of this message varies wildly.

RPL_LOGGEDOUT (901)

```
    "<client> <nick>!<user>@<host> :You are now logged out"
```

This numeric indicates that the client was logged out of their account. For more information on this numeric, see the IRCv3 *sasl-3.1* extension.

The text used in the last param of this message varies wildly.

ERR_NICKLOCKED (902)

```
    "<client> :You must use a nick assigned to you"
```

This numeric indicates that *SASL authentication* failed because the account is currently locked out, held, or otherwise administratively made unavailable. For more information on this numeric, see the IRCv3 *sasl-3.1* extension.

The text used in the last param of this message varies wildly.

☾

RPL_SASLSUCCESS (903)

```
    "<client> :SASL authentication successful"
```

This numeric indicates that *SASL authentication* was completed successfully, and is normally sent along with *RPL_LOGGEDIN* (`900`). For more information on this numeric, see the IRCv3 `sasl-3.1` extension.

The text used in the last param of this message varies wildly.

ERR_SASLFAIL (904)

```
    "<client> :SASL authentication failed"
```

This numeric indicates that *SASL authentication* failed because of invalid credentials or other errors not explicitly mentioned by other numerics. For more information on this numeric, see the IRCv3 `sasl-3.1` extension.

The text used in the last param of this message varies wildly.

ERR_SASLTOOLONG (905)

```
    "<client> :SASL message too long"
```

This numeric indicates that *SASL authentication* failed because the *AUTHENTICATE* command sent by the client was too long (i.e. the parameter was longer than 400 bytes). For more information on this numeric, see the IRCv3 `sasl-3.1` extension.

The text used in the last param of this message varies wildly.

ERR_SASLABORTED (906)

```
    "<client> :SASL authentication aborted"
```

This numeric indicates that *SASL authentication* failed because the client sent an *AUTHENTICATE* command with the parameter (`'*'`, `0x2A`). For more information on this numeric, see the IRCv3 `sasl-3.1` extension.

The text used in the last param of this message varies wildly.

ERR_SASLALREADY (907)

```
    "<client> :You have already authenticated using SASL"
```

This numeric indicates that *SASL authentication* failed because the client has already authenticated using SASL and reauthentication is not available or has been administratively disabled. For more information on this numeric, see the IRCv3 `sasl-3.1` and `sasl-3.2` extensions.

The text used in the last param of this message varies wildly.

RPL_SASLMECHS (908)

```
"<client> <mechanisms> :are available SASL mechanisms"
```

This numeric specifies the mechanisms supported for *SASL authentication*. <mechanisms> is a list of SASL mechanisms, delimited by a comma (`','`, `0x2C`). For more information on this numeric, see the IRCv3 `sasl-3.1` extension.

IRCv3.2 also specifies this information in the `sasl` client capability value. For more information on this, see the IRCv3 `sasl-3.2` extension.

The text used in the last param of this message varies wildly.

☾

Appendix D. RPL_ISUPPORT Parameters

Used to *advertise features* to clients, the *RPL_ISUPPORT* (005) numeric lists parameters that let the client know which features are active and their value, if any.

The parameters listed here are standardised and/or widely-advertised by IRC servers today and do not include deprecated parameters. Servers SHOULD support at least the following parameters where appropriate, and may advertise any others. For a more extensive list of parameters advertised by this numeric, see the `irc-defs` *RPL_ISUPPORT list*.

Certain parameters described here may not be standardised nor widely-advertised. These parameters are noted with the descriptor `"Status: Proposed"`. However, we try to be conservative with the parameters we're proposing, both in terms of having a small number of them and them being fairly understandable extensions to the current widely-used parameters.

If a 'default value' is listed for a parameter, this is the assumed value of the parameter until and unless it is advertised by the server. This is primarily to interoperate with servers that don't advertise particular well-known and well-used parameters. If an 'empty value' is listed for a parameter, this is the assumed value of the parameter if it is advertised without a value.

AWAYLEN Parameter

```
Format: AWAYLEN=<number>
```

The AWAYLEN parameter indicates the maximum length for the `<reason>` of an *AWAY* command. If an *AWAY* `<reason>` has more characters than this parameter, it may be silently truncated by the server before being passed on to other clients. Clients MAY receive an *AWAY* `<reason>` that has more characters than this parameter.

The value MUST be specified and MUST be a positive integer.

Examples:

```
AWAYLEN=200

AWAYLEN=307
```

CASEMAPPING Parameter

```
Format: CASEMAPPING=<casemap>
```

☾

The CASEMAPPING parameter indicates what method the server uses to compare equality of case-insensitive strings (such as channel names and nicks).

The value MUST be specified and MUST be a string representing the method that the server uses.

The specified casemappings are as follows:

- **ascii**: Defines the characters a to z to be considered the lower-case equivalents of the characters A to Z only.

- **rfc1459**: Same as 'ascii', with the addition of the characters '{', '}', '|', and '^' being considered the lower-case equivalents of the characters '[', ']', '\', and '~' respectively.

- **rfc1459-strict**: Same casemapping as 'ascii', with the characters '{', '}', and '|' being the lower-case equivalents of '[', ']', and '\', respectively. Note that the difference between this and rfc1459 above is that in rfc1459-strict, '^' and '~' are not casefolded.

- **rfc7613**: Proposed casemapping which defines a method based on PRECIS, allowing additional Unicode characters to be correctly casemapped [link].

The value MUST be specified and is a string. Servers MAY advertise alternate casemappings to those above, but clients MAY NOT be able to understand or perform them. If the parameter is not published by the server at all, clients SHOULD assume CASEMAPPING=rfc1459.

Servers SHOULD AVOID using the rfc1459 casemapping unless explicitly required for compatibility reasons or for linking with servers using it. The equivalency of the extra characters is not necessary nor useful today, and issues such as incorrect implementations and a conflict between matching masks exists.

Examples:

```
CASEMAPPING=ascii

CASEMAPPING=rfc1459
```

CHANLIMIT Parameter

```
Format: CHANLIMIT=<prefixes>:[limit]{,<prefixes>:[limit]}
```

The CHANLIMIT parameter indicates the number of channels a client may join.

The value MUST be specified and is a list of "<prefixes>:<limit>" pairs, delimited by comma (',', 0x2C). <prefixes> is a list of channel prefix characters as defined in

the *CHANTYPES* parameter. `<limit>` is OPTIONAL and if specified is a positive integer indicating the maximum number of these types of channels a client may join. If there is no limit to the number of these channels a client may join, `<limit>` will not be specified.

Clients should not assume other clients are limited to what is specified in the `CHANLIMIT` parameter.

Examples:

```
CHANLIMIT=#:25              ; indicates that clients may join 25 '#' channel

CHANLIMIT=#&:50             ; indicates that clients may join 50 '#' and 50

CHANLIMIT=#:70,&:           ; indicates that clients may join 70 '#' channel
                             number of '&' channels
```

CHANMODES Parameter

```
Format: CHANMODES=A,B,C,D[,X,Y...]
```

The `CHANMODES` parameter specifies the channel modes available and which types of arguments they do or do not take when using them with the *MODE* command.

The value lists the channel mode letters of **Type A**, **B**, **C**, and **D**, respectively, delimited by a comma (`','`, `0x2C`). The channel mode types are defined in the the *MODE* message description.

To allow for future extensions, a server MAY send additional types, delimited by a comma (`','`, `0x2C`). However, server authors SHOULD NOT extend this parameter without good reason, and SHOULD CONSIDER whether their mode would work as one of the existing types instead. The behaviour of any additional types is undefined.

Server MUST NOT list modes in this parameter that are also advertised in the *PREFIX* parameter. However, modes within the *PREFIX* parameter may be treated as type B modes.

Examples:

```
CHANMODES=b,k,l,imnpst

CHANMODES=beI,k,l,BCMNORScimnpstz

CHANMODES=beI,kfL,lj,psmntirRcOAQKVCuzNSMTGZ
```

CHANNELLEN Parameter

☾

```
    Format: CHANNELLEN=<string>
```

The CHANNELLEN parameter specifies the maximum length of a channel name that a client may join. A client elsewhere on the network MAY join a channel with a larger name, but network administrators should take care to ensure this value stays consistent across the network.

The value MUST be specified and MUST be a positive integer.

Examples:

```
    CHANNELLEN=32

    CHANNELLEN=50

    CHANNELLEN=64
```

CHANTYPES Parameter

```
    Format: CHANTYPES=[string]
    Default: CHANTYPES=#
```

The CHANTYPES parameter indicates the channel prefix characters that are available on the current server. Common channel types are listed in the *Channel Types* section.

The value is OPTIONAL; if it is not present, it indicates that no channel types are supported. If the parameter is not published by the server at all, clients SHOULD assume CHANTYPES=#&, corresponding to the RFC1459 behavior.

Examples:

```
    CHANTYPES=#

    CHANTYPES=&#

    CHANTYPES=#&
```

ELIST Parameter

```
    Format: ELIST=<string>
```

The ELIST parameter indicates that the server supports search extensions to the *LIST* command.

☾

The value MUST be specified, and is a non-delimited list of letters, each of which denote an extension. The letters MUST be treated as being case-insensitive.

The following search extensions are defined:

- **C**: Searching based on channel creation time, via the "`C<val`" and "`C>val`" modifiers to search for a channel that was created either less than `val` minutes ago, or more than `val` minutes ago, respectively

- **M**: Searching based on a mask.

- **N**: Searching based on a non-matching !mask. i.e., the opposite of `M`.

- **T**: Searching based on topic set time, via the "`T<val`" and "`T>val`" modifiers to search for a topic time that was set less than `val` minutes ago, or more than `val` minutes ago, respectively.

- **U**: Searching based on user count within the channel, via the "`<val`" and "`>val`" modifiers to search for a channel that has less or more than `val` users, respectively.

Examples:

```
ELIST=MNUCT

ELIST=MU

ELIST=CMNTU
```

> A widespread bug in existing implementations is to swap the semantics of "`C<val`" with "`C>val`", and/or "`T<val`" with "`T>val`", due to ambiguous legacy specifications. You should check the server you are using implements them as expected.

EXCEPTS Parameter

```
Format: EXCEPTS=[character]
 Empty: e
```

The EXCEPTS parameter indicates that the server supports ban exceptions, as specified in the *ban exception* channel mode section.

The value is OPTIONAL and when not specified indicates that the letter "e" is used as the channel mode for ban exceptions. If the value is specified, the character indicates the letter which is used for ban exceptions.

Examples:

```
EXCEPTS

EXCEPTS=e
```

EXTBAN Parameter

```
Format: EXTBAN=[<prefix>],<types>
```

The EXTBAN parameter indicates the types of "extended ban masks" that the server supports.

`<prefix>` denotes the character that indicates an extban to the server and `<types>` is a list of characters indicating the types of extended bans the server supports. If `<prefix>` does not exist then the server does not require a prefix for extbans, and they should be sent with no prefix.

Extbans may allow clients to issue bans based on account name, SSL certificate fingerprints and other attributes, based on what the server supports.

Extban masks SHOULD also be supported for the *ban exception* and *invite exception* modes.

> Ensure that extban masks are actually typically supported in ban exception and invite exception modes.
>
> We should include a list of 'typical' extban characters and their associated meaning, but make sure we specify that these are not standardised and may change based on server software. See also the irc-defs *EXTBAN list*.

Examples:

```
EXTBAN=~,cqnr

EXTBAN=~,qjncrRa

EXTBAN=,ABCNOQRSTUcjmprsz
```

HOSTLEN Parameter

☾

```
    Format: HOSTLEN=<number>
    Status: Proposed
```

The HOSTLEN parameter indicates the maximum length that a hostname may be on the server (whether cloaked, spoofed, or a looked-up domain name). Networks SHOULD be consistent with this value across different servers.

If a looked-up domain name is longer than this length, the server SHOULD opt to use the IP address instead, so that the hostname is underneath this length.

The value MUST be specified and MUST be a positive integer.

Examples:

```
    HOSTLEN=63
    HOSTLEN=64
```

INVEX Parameter

```
    Format: INVEX=[character]
     Empty: I
```

The INVEX parameter indicates that the server supports invite exceptions, as specified in the *invite exception* channel mode section.

The value is OPTIONAL and when not specified indicates that the letter "I" is used as the channel mode for invite exceptions. If the value is specified, the character indicates the letter which is used for invite exceptions.

Examples:

```
    INVEX

    INVEX=I
```

KICKLEN Parameter

```
    Format: KICKLEN=<length>
```

The KICKLEN parameter indicates the maximum length for the <reason> of a *KICK* command. If a *KICK* <reason> has more characters than this parameter, it may be silently truncated by the server before being passed on to other clients. Clients MAY receive a *KICK* <reason> that has more characters than this parameter.

The value MUST be specified and MUST be a positive integer.

Examples:

```
KICKLEN=255

KICKLEN=307
```

MAXLIST Parameter

```
Format: MAXLIST=<modes>:<limit>{,<modes>:<limit>}
```

The `MAXLIST` parameter specifies how many "variable" modes of type A that have been defined in the *CHANMODES* parameter that a client may set in total on a channel.

The value MUST be specified and is a list of `<modes>:<limit>` pairs, delimited by a comma (`',', 0x2C`). `<modes>` is a list of type A modes defined in *CHANMODES*. `<limit>` is a positive integer specifying the maximum number of entries that all of the modes in `<modes>`, combined, may set on a channel.

A client MUST NOT make any assumptions on how many mode entries may actually exist on any given channel. This limit only applies to the client setting new modes of the given types, and other clients may have different limits.

Examples:

```
MAXLIST=beI:25          ; indicates that a client may set up to a total
                        combination of "b", "e", and "I" modes.

MAXLIST=b:60,e:60,I:60  ; indicates that a client may set up to 60 "b" m
                        "e" modes, and 60 "I" modes.

MAXLIST=beI:100,q:50    ; indicates that a client may set up to a total
                        a combination of "b", "e", and "I" modes, and th
                        may set up to 50 "q" modes.
```

MAXTARGETS Parameter

```
Format: MAXTARGETS=[number]
```

The `MAXTARGETS` parameter specifies the maximum number of targets a *PRIVMSG* or *NOTICE* command may have, and may apply to other commands based on server software.

The value is OPTIONAL and if specified, `[number]` is a positive integer representing the maximum number of targets those commands may have. If there is no limit, then

`[number]` MAY not be specified.

The *TARGMAX* parameter SHOULD be advertised instead of or in addition to this parameter. *TARGMAX* is intended to replace `MAXTARGETS` as that parameter is more clear about which commands limits apply to.

Examples:

```
MAXTARGETS=4

MAXTARGETS=20
```

MODES Parameter

```
Format: MODES=[number]
```

The `MODES` parameter specifies how many 'variable' modes may be set on a channel by a single *MODE* command from a client. A 'variable' mode is defined as being a type A, B or C mode as defined in the *CHANMODES* parameter, or in the channel modes specified in the *PREFIX* parameter.

A client SHOULD NOT issue more 'variable' modes than this in a single *MODE* command. A server MAY however issue more 'variable' modes than this in a single *MODE* message. The value is OPTIONAL and when not specified indicates that there is no limit to the number of 'variable' modes that may be set in a single client *MODE* command. If the parameter is not published by the server at all, clients SHOULD assume `MODES=3`, corresponding to the RFC1459 behavior.

If the value is specified, it MUST be a positive integer.

Examples:

```
MODES=4

MODES=12

MODES=20
```

NETWORK Parameter

```
Format: NETWORK=<string>
```

The `NETWORK` parameter indicates the name of the IRC network that the client is connected to. This parameter is advertised for INFORMATIONAL PURPOSES ONLY. Clients SHOULD NOT use this value to make assumptions about supported features on the

server as networks may change server software and configuration at any time.

Examples:

```
NETWORK=EFNet

NETWORK=Rizon

NETWORK=Example\x20Network
```

NICKLEN Parameter

```
Format: NICKLEN=<number>
```

The `NICKLEN` parameter indicates the maximum length of a nickname that a client may set. Clients on the network MAY have longer nicks than this.

The value MUST be specified and MUST be a positive integer. 30 or 31 are typical values for this parameter advertised by servers today.

Examples:

```
NICKLEN=9

NICKLEN=30

NICKLEN=31
```

PREFIX Parameter

```
 Format: PREFIX=[(modes)prefixes]
Default: PREFIX=(ov)@+
```

Within channels, clients can have different statuses, denoted by single-character prefixes. The PREFIX parameter specifies these prefixes and the channel mode characters that they are mapped to. There is a one-to-one mapping between prefixes and channel modes. The prefixes in this parameter are in descending order, from the prefix that gives the most privileges to the prefix that gives the least.

The typical prefixes advertised in this parameter are listed in the *Channel Membership Prefixes* section.

The value is OPTIONAL and when it is not specified indicates that no prefixes are supported. If the parameter is not published by the server at all, clients SHOULD assume PREFIX=(ov)@+, corresponding to the RFC1459 behavior.

Examples:

```
PREFIX=(ov)@+

PREFIX=(ohv)@%+

PREFIX=(qaohv)~&@%+
```

SAFELIST Parameter

```
Format: SAFELIST
```

If SAFELIST parameter is advertised, the server ensures that a client may perform the *LIST* command without being disconnected due to the large volume of data the *LIST* command generates.

The SAFELIST parameter MUST NOT be specified with a value.

Examples:

```
SAFELIST
```

SILENCE Parameter

```
Format: SILENCE[=<limit>]
```

The SILENCE parameter indicates the maximum number of entries a client can have in their silence list.

The value is OPTIONAL and if specified is a positive integer. If the value is not specified, the server does not support the *SILENCE* command.

Most IRC clients also include client-side filter/ignore lists as an alternative to this command.

Examples:

```
SILENCE

SILENCE=15

SILENCE=32
```

STATUSMSG Parameter

```
        Format: STATUSMSG=<string>
```

The STATUSMSG parameter indicates that the server supports a method for clients to send a message via the *PRIVMSG* / *NOTICE* commands to those people on a channel with (one of) the specified *channel membership prefixes*.

The value MUST be specified and MUST be a list of prefixes as specified in the *PREFIX* parameter. Most servers today advertise every prefix in their *PREFIX* parameter in STATUSMSG.

Examples:

```
        STATUSMSG=@+

        STATUSMSG=@%+

        STATUSMSG=~&@%+
```

TARGMAX Parameter

```
        Format: TARGMAX=[<command>:[limit]{,<command>:[limit]}]
```

Certain client commands MAY contain multiple targets, delimited by a comma (`,`, `0x2C`). The TARGMAX parameter defines the maximum number of targets allowed for commands which accept multiple targets. If this parameter is not advertised or a value is not sent then a client SHOULD assume that no commands except the `JOIN` and `PART` commands accept multiple parameters.

The value is OPTIONAL and is a set of `<command>:<limit>` pairs, delimited by a comma (`,`, `0x2C`). `<command>` is the name of a client command. `<limit>` is the maximum number of targets which that command accepts. If `<limit>` is specified, it is a positive integer. If `<limit>` is not specified, then there is no maximum number of targets for that command. Clients MUST treat `<command>` as case-insensitive.

Examples:

```
        TARGMAX=PRIVMSG:3,WHOIS:1,JOIN:

        TARGMAX=NAMES:1,LIST:1,KICK:1,WHOIS:1,PRIVMSG:4,NOTICE:4,ACCEPT:,MONITOR:

        TARGMAX=ACCEPT:,KICK:1,LIST:1,NAMES:1,NOTICE:4,PRIVMSG:4,WHOIS:1
```

TOPICLEN Parameter

☾

```
    Format: TOPICLEN=<number>
```

The TOPICLEN parameter indicates the maximum length of a topic that a client may set on a channel. Channels on the network MAY have topics with longer lengths than this.

The value MUST be specified and MUST be a positive integer. 307 is the typical value for this parameter advertised by servers today.

Examples:

```
    TOPICLEN=307

    TOPICLEN=390
```

USERLEN Parameter

```
    Format: USERLEN=<number>
    Status: Proposed
```

The USERLEN parameter indicates the maximum length that a username may be on the server. Networks SHOULD be consistent with this value across different servers. As noted in the *USER* message, the tilde prefix ("~"), if it exists, contributes to the length of the username and would be included in this parameter.

The value MUST be specified and MUST be a positive integer.

Examples:

```
    USERLEN=12
    USERLEN=18
```

☾

Appendix E. Current Architectural Problems

There are a number of recognized problems with the IRC protocol. This section only addresses the problems related to the architecture of the protocol.

10.1   Scalability

It is widely recognized that this protocol may not scale sufficiently well when used in a very large arena. The main problem comes from the requirement that all servers know about all other servers, clients, and channels, and that information regarding them be updated as soon as it changes.

Server-to-server protocols can attempt to alleviate this by, for example, only sending 'necessary' state information to leaf servers. These sort of optimisations are implementation-specific and are not covered in this document. However, server authors should take great care in their protocols to ensure race conditions and other network instability does not result from these attempts to improve the scalability of their protocol.

10.2   Reliability

As the only network configuration used for IRC servers is that of a spanning tree, each link between two servers is an obvious and serious point of failure.

Software authors are and have been experimenting with alternative topologies such as mesh networks. However, there is not yet a production implementation or specification of any topology other than spanning-tree.

☾

Appendix F. Implementation Notes

The IRC protocol is reasonably complex. When writing software that interacts with it, there are certain choices that are implementation-defined, as well as certain areas that are commonly incorrectly implemented.

This section raises discussion, questions, and recommendations intended to help implementors. In particular, the advice/discussion here may be sloppy compared to the above, and the questions may be less well-defined or without strict answers, but regardless should help you when writing software that interacts with the IRC protocol.

10.1   Character Encodings

Character encodings in IRC are hard. *UTF-8* is recommended, the mess of *Latin-1/ISO-8859-1(5)/CP1252* also seems common, but all sorts of other encodings are also used in practice. Particularly on networks that support other languages, and were created before UTF-8 became as widespread as it has.

When sending, we always recommend UTF-8. When decoding, we generally recommend trying UTF-8 and falling back to Latin-1 (what has been called the Hybrid encoding).

For clients, this is fine. Even if they incorrectly decode a private message, the user should see that the message has been decoded incorrectly and be able to resolve the issue (hopefully by telling the sending user to use UTF-8).

However, servers are in a trickier position (especially for PRIVMSG/NOTICE or any other command that takes arbitrary user input such as USER, TOPIC, etc). Servers should simply treat this input from the user as a character array they accept and then spit out again, no trouble.

Servers implemented in languages with first-class Unicode strings may wish to treat IRC lines and messages as Unicode text internally. For servers to treat messages in this way, they need to decode lines as they're received and later encode the lines before they're sent out.

This presents an issue. What if the line from the user is decoded incorrectly, modified (eg. by casefolding), and then sent out? (see also: *Mojibake*). What these servers may instead do is either:

1. follow the lead of the majority of existing servers and treat these parameters as byte arrays not to be parsed or decoded in any way.

2. attempt to decode all incoming lines as UTF-8 (possibly using Hybrid encoding like clients do) and if the line cannot be decoded it is ignored or returns an error. The *IRCv3 UTF8ONLY specification* allows them to signal this to clients.

The former ensures all messages are sent correctly, and the latter simplifies server

implementations and allows clients to disable decoding heuristics.

10.2   Message Parsing and Assembly

Message parsing/assembly is one area where implementations can differ wildly, and is a common vector for both security issues and general runtime problems.

Message Parsing is turning raw IRC messages into the various message parts (tags, prefix, command, parameters). Message Assembly is the opposite – taking the various message parts and creating an IRC line to be sent over the wire.

Implementors should ensure that their message parsing and assembly responds in expected ways, by running their software through test cases. I recommend these public-domain *irc-parser-tests*, which are reasonably extensive.

Trailing

Trailing is *a completely normal parameter*, except for the fact that it can contain spaces. When parsing messages, the 'normal params' and trailing should be appended and returned as a single list containing all the message params.

This is an example of an incorrect parser, that specifically separates normal params and trailing. When returning messages after parsing, **don't return a struct/object containing these variables:**

```
Message
    .Tags
    .Source
    .Verb
    .Params (containing all but the trailing param)
    .Trailing (containing just the trailing param)
```

Trailing *is a normal parameter*. Separating the parameter types in this way *will cause many breakages and weird issues*, as logic code will depend on the final param being in either `.Params` or `.Trailing`, when the simple fact is that it can be in either. Make sure that your message parser instead outputs parsed messages more like this:

```
Message
    .Tags
    .Source
    .Verb
    .Params (including all normal params, and the trailing param if it ex
```

This will make sure that you don't run into silly trailing parameter errors.

Direct String Comparisons on IRC Lines

Some software decides that the best way to process incoming lines is with something

along the lines of this:

```
Line = NewIRCLineFromSocket()
If Line.StartsWith("PART") {
    Part(...etc...)
} Else If Line.StartsWith("QUIT") {
    Quit(...etc...)
}
```

This is bad. This will break. Here's why: *Any IRC message can choose to include or not include the `source`*.

If you directly compare the beginning of lines like this, then you will break when servers decide to start including sources on messages (for example, some newer IRCds decide to include the source on all messages that they output). This results in clients that don't correctly parse incoming messages and break as a result.

Instead, you should make sure that you send incoming lines through a message parser, and then do things based on what's output by that parser. For instance:

```
Message = IRCMessageParser(Line)
If Message.Verb == "PART" {
    Part(...etc...)
} Else If Message.Verb == "QUIT" {
    Quit(...etc...)
}
```

This will ensure that your software doesn't break when clients or servers send extra, or omit unnecessary, message elements.

Something to keep in mind is that the message verb is always case insensitive, so you should casemap it appropriately before doing comparisons similar to the above. In my own IRC libraries, I convert the verb to uppercase before returning the message.

10.3 Casemapping

Casemapping, at least right now, is a topic where implementations differ greatly.

Servers

- Does your server use `"rfc1459"` or `"rfc1459-strict"` casemapping? If so, can you use a casemapping with less ambiguity such as `"ascii"`?

- Does your server store state using nicks/channel names as keys? If so, is your server written in such a way that keys are casefolded automatically, or that ensures keys are casefolded before using them in this way?

Clients

- Does your client store state using nicks/channel names as keys, and if so do you casefold those keys appropriately?

- Does your client discover the casemapping to use from the *CASEMAPPING* RPL_ISUPPORT parameter on connection? If so, does your client use the appropriate casemapping based on it?

Appendix G. Obsolete Commands and Numerics

10.1   Obsolete Commands

- *SUMMON*: was used to request people to connect to the network, by writing to their TTY. This only made sense back when users had shells on the same server as the IRC daemon.

- *TRACE*: showed a path in the server graph, between the user and a target. Nowadays, many servers either don't implement it, or return redacted data.

- *ISON*: replaced by the *IRCv3 Monitor* specification

- *WATCH*: was never formally specified, and is also replaced by *IRCv3 Monitor*.

10.2   Obsolete Numerics

These are numerics contained in *RFC1459* and *RFC2812* that are not contained in this document or that should be considered obsolete.

- **RPL_BOUNCE (005)**: `005` is now used for *RPL_ISUPPORT* (`005`). *RPL_BOUNCE* (`010`) was moved to `010`

- **RPL_SUMMONING (342)**: Was a reply to the deprecated SUMMON command.

Acknowledgements

This document draws heavily from the original *RFC1459* and *RFC2812* IRC protocol specifications.

Parts of this document come from the "IRC RPL_ISUPPORT Numeric Definition" Internet Draft authored by L. Hardy, E. Brocklesby, and K. Mitchell. Parts of this document come from the "IRC Client Capabilities Extension" Internet Draft authored by K. Mitchell, P. Lorier, L. Hardy, and P. Kucharski. Parts of this document come from the *IRCv3 Working Group* specifications.

Thanks to the following people for contributing to this document, or to helping with IRC specification efforts:

Simon Butcher, dx, James Wheare, Stephanie Daugherty, Sadie, and all the IRC developers and documentation writers throughout the years.

The canonical version of this document is hosted at *http://modern.ircdocs.horse*

You can talk to us at *#ircdocs on Libera.Chat*

Pull requests may be submitted to and the source code for it can be found at *http://github.com/ircdocs/modern-irc*

☾