Vodafone *live!*cast Protocol Format

1 Introduction

This document defines the *live!*cast protocol and message format to be used over UDP and Cell Brodcast bearers.

1.1 Contributors

Bruno Rodrigues	bruno.rodrigues@vodafone.com	VF-Group GM-TSM
Stefano Nardo	stefano.nardo@vodafone.com	VF-Group GM-TSM
Stephen Packer	stephen.packer@vodafone.com	VF-Group GT
Richard Bennett	richard.bennett@vodafone.com	VF-Group GT
Rui Carmo	rui.carmo@vodafone.com	VF-PT
João Machado	joao.machado@vodafone.com	VF-PT

1.2 Changes

2005.03.14	Bruno Rodrigues Richard Bennet	Merged R6 technical protocol from lc-spec 1.32 and lc-spec-alternative 1.17 and moved functional requirements to the TCD.

1.3 Notation Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", “MAY NOT” and "OPTIONAL" in this document are to be interpreted as described in RFC 2119[1].

1.3.1 Augmented BNF

All of the mechanisms specified in this document are described in both prose and an augmented Backus-Naur Form (BNF) similar to that used by RFC 822[2]. Implementors will need to be familiar with the notation in order to understand this specification. The augmented BNF includes the following constructs:

`name = definition`

The name of a rule is simply the name itself (without any enclosing "<" and ">") and is separated from its definition by the equal "=" character. White space is only significant in that indentation of continuation lines is used to indicate a rule definition that spans more than one line. Certain basic rules are in uppercase, such as SP, LWS, HT, CRLF, DIGIT, ALPHA, etc. Angle brackets are used within definitions whenever their presence will facilitate discerning the use of rule names.

`"literal"`

Quotation marks surround literal text. Unless stated otherwise, the text is case-insensitive.

`rule1 | rule2`

Elements separated by a bar ("|") are alternatives, e.g., "yes | no" will accept yes or no.

`(rule1 rule2)`

Elements enclosed in parentheses are treated as a single element. Thus, "(elem (foo | bar) elem)" allows the token sequences "elem foo elem" and "elem bar elem".

`*rule`

`n*mrule`

The character "*" preceding an element indicates repetition. The full form is "<n>*<m>element" indicating at least <n> and at most <m> occurrences of element. Default values are 0 and infinity so that "*(element)" allows any number, including zero; "1*element" requires at least one; and "1*2element" allows one or two.

`[rule]`

Square brackets enclose optional elements; "[foo bar]" is equivalent to "*1(foo bar)".

`N rule`

Specific repetition: "<n>(element)" is equivalent to "<n>*<n>(element)"; that is, exactly <n> occurrences of (element). Thus 2DIGIT is a 2-digit number, and 3ALPHA is a string of three alphabetic characters.

`#rule`

A construct "#" is defined, similar to "*", for defining lists of elements. The full form is "<n>#<m>element" indicating at least <n> and at most <m> elements, each separated by one or more commas (",") and OPTIONAL linear white space (LWS). This makes the usual form of lists very easy; a rule such as

```
( *LWS element *( *LWS "," *LWS element ))
```

can be shown as

```
1#element
```

Wherever this construct is used, null elements are allowed, but do not contribute to the count of elements present. That is, "(element), , (element) " is permitted, but counts as only two elements. Therefore, where at least one element is required, at least one non-null element MUST be present. Default values are 0 and infinity so that "#element" allows any number, including zero; "1#element" requires at least one; and "1#2element" allows one or two.

`; comment`

A semi-colon, set off some distance to the right of rule text, starts a comment that continues to the end of line. This is a simple way of including useful notes in parallel with the specifications.

`implied *LWS`

The grammar described by this specification is word-based. Except where noted otherwise, linear white space (LWS) can be included between any two adjacent words (token or quoted-string), and between adjacent words and separators, without changing the interpretation of a field. At least one delimiter (LWS and/or separators) MUST exist between any two tokens (for the definition of "token" below), since they would otherwise be interpreted as a single token.

1.3.2 Basic Rules

The following rules are used throughout this specification to describe basic parsing constructs. The US-ASCII coded character set is defined by ANSI X3.4-1986[3].

```
    OCTET       = <any 8-bit sequence of data>
    CHAR        = <any US-ASCII character (octets 0 - 127)>
    UPALPHA     = <any US-ASCII uppercase letter "A".."Z">
    LOALPHA     = <any US-ASCII lowercase letter "a".."z">
    ALPHA       = UPALPHA | LOALPHA
    DIGIT       = <any US-ASCII digit "0".."9">
    CTL         = <any US-ASCII control character
                  (octets 0 - 31) and DEL (127)>
    CR          = <US-ASCII CR, carriage return (13)>
    LF          = <US-ASCII LF, linefeed (10)>
    SP          = <US-ASCII SP, space (32)>
    <">         = <US-ASCII double-quote mark (34)>
    CRLF        = CR LF
    HEX         = "A" | "B" | "C" | "D" | "E" | "F"
                | "a" | "b" | "c" | "d" | "e" | "f" | DIGIT

    TEXT        = <any CHAR except CTL>
    BOOLEAN     = TRUE | FALSE
```

```
    TRUE         = TRUESTRING | YESSTRING | "1"
    FALSE        = FALSESTRING | NOSTRING | "0"
    TRUESTRING   = "T"|"t" "R"|"r" "U"|"u" "E|e"
    FALSESTRING  = "F"|"f" "A"|"a" "L"|"l" "S"|"s" "E"|"e"
    YESSTRING    = "Y"|"y" "E"|"e" "S"|"s"
    NOSTRING     = "N"|"n" "O"|"o"
    UINTVAR      = < 1*OCTET as defined in Error! Reference source not
found. >
    HTMLTEXT     = < *TEXT except "<" and ">", and where "&" starts a
valid entity >
```

2 Generic Information and Common Fields

This section defines generic information and common fields used in the Message Format, in UDP Protocol, in CB Protocol, or in the Channel Selection Service.

2.1 Generic Information

2.1.1 Message Size

The size of a message received in the UDP payload, or after concatenating all CB pages and before any 7bit decoding, MUST be a multiple of 82 octets, as there's no header in CB to define the real size of a page.

2.1.2 Message Padding

The message MUST be padded with spaces in the end (before 7bit encoding if it's a 7bit message) by the sender to make the final packet a multiple of 82 octets.
The message MUST be stripped from spaces in the end (after 7bit decoding if it's a 7bit message) by the receiver.

2.1.3 Decoding 7bit Content

In a 7bit encoded Cell Broadcast message, the complete message is split in 93 character chunks and each chunk is 7bit encoded to a 82 bytes page.
To decode a 7bit encoded CB message, each page MUST be decoded first, and then all 93 characters chunks MUST be concatenated to produce the final 8bit message.
As there is no concatenation in UDP and the payload is exactly the same as when concatenating all CB pages before any decoding, to decode a 7bit encoded UDP message the 7bit payload MUST be split into 82 bytes chunks. Then each chunk MUST be 7bit decoded and each 93 characters chunk is then concatenated to produce the final 8 bit message.
7 Bit Encoding is defined in Section 6.1.2.2 of 3GPP 23.038[4].

2.1.4 Decoding 7bit Content inside Multipart

A 7bit encoded part inside a multipart (XHTML or SVG) doesn't need to be splited in chunks and the 7bit decoded can be applied to the whole payload.
The content parts inside multipart are not padded with spaces but the received MUST still strip them as defined in Section Message Padding.

2.1.5 GSM Alphabet and Special Characters

All 7bit encoded content parts (stand-alone and inside Multipart) are encoded with GSM Alphabet as defined in Section 6.2.1 of 3GPP 23.038[4].

In XHTML content parts, the characters not available on the GSM Alphabet table are encoded as HTML entities or compressed UCS-2 entities as commonly used for XHTML markup.
In SVG content parts, the characters not available on the GSM Alphabet table are encoded as compressed UCS-2 entities as commonly used for XML markup.
The client MUST decode the GSM alphabet and interpret correctly the HTML entities or compressed UCS-2 entities.

2.2 Common Fields

2.2.1 Date Format

Dates are defined as a sortable string consisting on the concatenation of the year, month, day, hour, minute and seconds, without any character in between, and an optional timezone in the end which should be used to define the timezone of the date.
If Timezone is not present, the message MUST be assumed as a GMT date. If date is GMT, it SHOULD NOT contain the redundant Timezone part.

```
Date           = Year Month Date Hour Minute Second [ Timezone ]
Year           = 4 DIGIT
Month          = 2 DIGIT
Day            = 2 DIGIT
Hour           = 2 DIGIT
Minute         = 2 DIGIT
Second         = 2 DIGIT
Timezone       = ( "+" | "-" ) HourTimezone [ MinuteTimezone ]
HourTimezone   = 2 DIGIT
MinuteTimezone = 2 DIGIT
```

Date examples (all examples are the same date value):

```
20040101120000
20040101120000+00
20040101130000+01
20040101110000-01
```

Real world example:

```
Date Sent: 20040101130000+01
In Portugal (GMT+0), device shows: 2004.01.01 12:00:00
In Azores (GMT-1), device shows: 2004.01.01 11:00:00
In Germany (GMT+1), device shows: 2004.01.01 13:00:00
```

Date is used in *live!*cast Messages, in UDP headers, and in the Channel Selection XML response.

2.2.2 Channel Format

In *live!*cast, channels are defined as an integer between 0 and 4095. *live!*cast uses a "reserved for Opco" CB channel band starting at 40960 (0xA000), which means that "*live!*cast channel 1" represents real CB channel 40961 (0xA001). The maximum value for channel is 4095, which is real CB channel 45055 (0xAFFF).
Channel is called "Message Identifier" in 3GPP 23041[5].

```
Channel = 1*4 DIGIT  ; between 0 and 4095 inclusive
```

Channel examples:

```
1
4095
```

Channel is used in UDP headers and in the Channel Selection.

2.2.3 Serial Number Format

Serial Number value is defined as a integer between 0 and 65535 and is based on CB's two bytes unsigned integer Serial Number, as specified in Section 9.4.1.2.1 of 3GPP 23.041[5].

Serial Number consists of three groups of bits. The two most significant bits defines Geographic Scope. The next 10 bits are Message Code, and the last 4 bits define Update Number.

```
Serial-Number = 1*5DIGIT  ; between 0 and 65535
    ; bit 15,14 – Geographic Scope
    ; bit 13,4  - Message Code
    ; bit 3,0   - Update Number
```

Serial-Number examples:

```
16384
```

The full Serial Number behavior for CB is defined in 3GPP 23.041[5] and the implementation SHOULD be compliant with the CB spec, if CB is implemented.

*live!*cast implementation MUST be compliant with the following rules, which are a sub-set of the rules defined for CB.

```
Geographical Scope = 1 | 2 | 3  ; 01 | 10 | 11; 00 is invalid
```

Message unique identifier is composed by Message Code and Update Number together, and optionally the rules of Geographical Scope. A simplified implementation MAY assume that the complete Serial Number identifies the message.

If this value is the same as one already received, the message MUST be discarded but the last received message date is still updated.

If this value is different from every value in the message queue, the message MUST be accepted and stored.

If full 3GPP is implemented, the device SHOULD NOT replace the message content in the *live!*cast queue when Message Code is the same and Update Number is bigger or different.

Serial Number is used in UDP headers, and encoded as a two byte integer in the CB header.

2.2.4 Data Coding Format

Data Coding value is defined as an integer between 0 and 255 and is based on CB's one byte unsigned int Data Coding Scheme, as specified in 3GPP 23.038[4].

```
Data-Coding = 1*3DIGIT  ; between 0 and 255
```

Data Coding examples:

```
15
244
```

Data Coding values are defined in in 3GPP 23.038[4]. *live!*cast uses both 7bit and 8bit messages, doesn't use the language combinations, and only expects messages without message class, or with class 0 or 1. A complete list of valid DCS values for *live!*cast is described below. All these DCS values MUST be accepted. All other values MUST be discarded. The values below are in binary.

```
0000 1111  ; 7bit, No Language (prefered)
0100 00xx  ; 7bit, General DC Indication, No Message Class
0101 0000  ; 7bit, General DC Indication, Message Class 0
0101 0001  ; 7bit, General DC Indication, Message Class 1
1111 0000  ; 7bit, Data Coding/Message Handling, No Message Class
(prefered)
1111 0001  ; 7bit, Data Coding/Message Handling, Message Class 1
```

```
    0000 xxxx (except 0000 1111)    ; 7bit Text, With Language
(optional)
    0100 01xx  ; 8bit, General DC Indication, No Message Class
    0101 0100  ; 8bit, General DC Indication, Message Class 0
    0101 0101  ; 8bit, General DC Indication, Message Class 1
    1111 0100  ; 8bit, Data Coding/Message Handling, No Message Class
(preferred)
    1111 0101  ; 8bit, Data Coding/Message Handling, Message Class 1
```

The combinations from 0 to 14 SHOULD be accepted independently of the languages selected by the user in the regular CB settings.
The combinations 15 (Unspecified Language), MUST be accepted independently of the languages selected by the user in the regular CB settings.
Data Coding is used in the UDP headers, and encoded as one byte in the CB header.

3 *live!*cast Message Format

This section defines the *live!*cast message payload format, which is independent of the bearer.

3.1 Generic Values

3.1.1 Content-Type List

List of all content types values available for Message or for each Multipart Part:

Octet	Name	Mime-Type
0x20	XHTML-MP Snippet	Application/vnd.wap.xhtml+xml snippet
0x21	XHTML-MP	application/vnd.wap.xhtml+xml
0x30	SVGT1.1 (7bit)	image/svg+xml + 7bit encoding
0x31	SVGT1.1 (Gzip)	image/svg+xml + Gzip encoding
0x40	PNG	image/png
0x41	JPEG	image/jpeg
0x42	GIF	image/gif
0x50	Multipart	application/vnd.wap.multipart.mixed

3.2 live!cast Message Format

*live!*cast message format applies to final 8 bit decompressed payload. If the message is 7bit encoded, it MUST be decoded first as described in the Section Decoding 7bit Content.

3.2.1 Message Format

```
Message = Message-Content-Type Data
Data    = XHTML-MP-Snippet | XHTML-MP | Multipart
```

3.2.2 Message-Content-Type Value

The only valid formats for a *live!*cast message is XHTML-MP, XHTML-MP Snippet, and Multipart. Content-type values are defined in Section Content-Type List.

```
Content-Type-Octet = OCTET  ; valid 0x20, 0x21, 0x50
```

3.2.3 XHTML-MP Snippet Value

A XHTML-MP Snippet is a piece of XHTML-MP without redundant header and footer. As the header and footer are fixed, they MUST be stored locally into the device and the implementation MUST concatenate the header, the received XHTML-MP snippet, and the footer, and produce a valid XHTML-MP document. If the document is not valid, the message MAY be dropped.

```
XHTML-MP-Snippet = *TEXT

XHTML-MP-Content = Local-header decoded-snippet local-footer
Local-header = *TEXT  ; local file
Local-Footer = *TEXT  ; local file
Decoded-Snippet = gsm-decode(XHTML-MP-Snippet)
```

The header, footer and possible auxiliar files like CSS and Ecmascript MUST be optimized for the device.

Local Header example:

```
<html>
  <head>
    <title/>
    <link          rel="stylesheet"          type="text/css"
href="<somelocalpath>/livecast.css" />
  </head>
  <body>
```

Local Footer example:

```
  </body>
</html>
```

Local livecast.css example:

```
body {
  margin: 0;
  padding: 0;
  color: #000000;
  background: white;
  font-size: 10;
}
div#ti {
  font-weight: bold;
}
a[href], a:visited, a:focus {
  background-color: #595959;
  color: #FFFFFF;
  text-decoration: none;
  font-size:8;
}
```

XHTML-MP Content is defined in Section XHTML-MP Content Format.

3.2.4 XHTML-MP Type

*live!*cast messages in XHTML-MP type are similar to XHTML-MP Snippets except that there MUST NOT be any concatenation with the local header and footer. The MUST still be dropped if it's invalid.

```
XHTML-MP = *TEXT

XHTML Content = gsm-decode(XHTML-MP)
```

XHTML-MP Content is defined in Section XHTML-MP Content Format.

3.2.5 Multipart Type

*live!*cast uses a binary version of multipart very similar to OMA's WSP binary multipart, but simplified.

```
Multipart     = Num-Parts Parts
Num-parts     = OCTET  ; unsigned one byte integer
Parts         = Part1 1*Part2 ; doesn't make sense with 1 part
```

Each part contains a header with the data size and a content-type. The size header is also borrowed from OMA's WSP binary specification, but it's also included in this document in the Section **Error! Reference source not found.**.

The Content-Type header value is defined in the Section Content-Type List, but only some types are valid in each part. The first part MUST be markup, thus value 0x20 or 0x21, and the second part MUST be an object type (image or SVG), thus 0x30, 0x31, 0x40, 0x41 or 0x42.

```
Part1         = Data-Size Content-Type1 Data1
Part2         = Data-Size Content-Type2 Data2
Data-Size     = UINTVAR
Content-Type1 = OCTET  ; valid 0x20, 0x21
Content-Type2 = OCTET  ; valid 0x30, 0x31, 0x40, 0x41, 0x42
Data1         = XHTML-MP-Snippet | XHTML-MP
Data2         = SVG7BIT | SVGGZIP | PNG | JPEG | GIF
SVG7BIT       = decode_gsm( decode_7bit( *OCTET ) )
SVGGZIP       = decode_gzip( *OCTET )
PNG           = *OCTET  ; binary content
JPEG          = *OCTET  ; binary content
GIF           = *OCTET  ; binary content
```

3.3 XHTML-MP Content Format

*live!*cast XHTML-MP Content format is generic XHTML Mobile Profile markup that MUST be rendered as in regular browser.

A common format will be used and a set of rules are defined to enable the implementation to grab part of the content to display in other views, like a ticker showing an icon, a channel and a scrolling title.

XHTML-MP Snippet Example:

```
      <div><div id="ti">Title</div>Some description.<br/>More
description<br/><a id="li" href="/livecast?id=123">More…</a><span
id="me" style="--vf-lc-ch:Channel;--vf-lc-ic:default;--vf-lc-
dt:20040101120000"/></div>


      <div><div id="ti">Title</div><img alt="." src=cid:1/>Some
description.<br/>More description<br/><a id="li"
href="/livecast?id=123">More…</a><span id="me" style="--vf-lc-
ch:Channel;--vf-lc-ic:default;--vf-lc-dt:20040101120000"/></div>


      <div><a id="li" href="/livecast?id=123"><object width="100%"
data=cid:1/></a><span id="me" style="--vf-lc-ch:Channel;--vf-lc-
ic:default;--vf-lc-dt:20040101120000"/></div>
```

3.3.1 Message Title

A message MAY contain a message title. The title doesn't have a maximum size but it's recommended to be smaller than 64 characters. XHTML entities MUST NOT be used.

```
Message-Title = *HTMLTEXT
```

3.3.2 Message Link

A message may contain one or more XHTML anchor links. The default link MUST be identified by the first anchor element with an id=”li” attribute. If there is no id=”li”, the client SHOULD NOT have a default link or MAY use the first anchor as the default link.

The implementation MUST support the same URI schemas as in a regular browsing session, which means supporting at least, but not limited to, the following schemas: “http”, “https”, “wtai” and “mailto”.

```
Message-Link = URI
URI          = <The same URI schemas supported by the browser>
```

When the link is a http or https link, the client MUST support relative url’s by concatenating it with a defined BASE_URL.

The BASE_URL concatenation MUST be performed and stored in queue when the message arrives and not later when the message is retrieved.

3.3.3 Object/Img Reference

A message may contain one or more “object” or “img” elements pointing to objects inside the multipart package. In that case, the “href” or “data” attribute MUST be in the form “cid:x”, where “x” is the next part inside the multipart, after the markup. For example, a markup referencing “cid:1” MUST use the first object present in the second multipart part.

```
Object-Link = "cid:" cid
cid         = DIGIT  ; between 1 and 9
```

3.3.4 External References

A message may contain an element that references an external object (e.g. “img”, “object”, “link”, “script”). A *live!*cast message is always self contained and the implementation MUST NOT request the external object, neither locally stored nor remotely.

3.3.5 Message Channel Name

Every message may contain its own “Channel Name”. This string MUST be concatenation with default channel name in the format “Channel Name” “-“ “Message Channel Name”, unless the string is too big to fit the view space. In that case, the client MUST show only the Message Channel Name. If this value is ommited, the default Channel Name MUST be used.

```
Message-Channel-Name = *HTMLTEXT  ;  max 32 OCTETS
```

3.3.6 Message Icon

Every message may contain its own icon. This value is a string referencing an icon name and doesn’t include any file extension. The client MUST understand the value and convert it to a URI pointing to a local icon, with the right extension. If the value is invalid, or the icon doesn’t exist, the client MUST use the “default” value.

```
Message-Icon = *HTMLTEXT  ;  max 16 OCTETS
```

3.3.7 Message Date

Every message may contain its own date. This value is a string as defined in Section Date Format. The client MUST understand the value and convert it to a human readable string. The client MUST convert the date’s timezone to the device’s timezone. The client SHOULD convert the date to the format specified by the user in

the date settings menu. If message date is ommited, the client MUST use the message arrival time.

```
Message-Date = Date  ; Section Date Format
```

3.4 Content Identification

Content MUST be identified by the XML "id" attribute and never by element name (e.g. using Ecmascript's getElementByID()).

4 UDP Packet Format

*live!*cast content is restricted to a maximum of 1230 bytes (compressed). As a UDP is maximum 1500 bytes (usual MTU), one *live!*cast message is exactly one UDP packet, thus there is no concatenation or reassembly in UDP.

4.1 UDP Packet Format

The *live!*cast UDP Packet format is similar to HTTP and SIP, with requests, responses, methods, headers and body payload.

4.1.1 Message Types

*live!*cast messages consists on Request and Response messages sent either from the terminal to the server or vice-versa.
Request and Response messages use the generic message format of RFC 822[2] for transferring the payload of the message. Both types of messages consist of a start-line, zero or more header fields (also known as "headers"), an empty line (i.e., a line with nothing preceding the CRLF) indicating the end of the header fields, and possibly a message-body.

```
live!cast   = Start-Line
              *(Message-Header)
              CRLF
              [ Message-Body ]
Start-Line = Request-Line
           | Response-Line
```

4.1.2 Request Line

The Request-Line begins with a method token, followed by a fixed string ("livecast") and the protocol version, and ending with CRLF. The elements are separated by SP characters. No CR or LF is allowed except in the final CRLF sequence.

```
Request-Line = Method SP "livecast" SP livecast-Version CRLF
```

4.1.3 Response Line

The Response-Line begins with a protocol version followed by a numeric status code and its associated textual phrase, with each element separated by SP characters. No CR or LF is allowed except in the final CRLF sequence.

```
Response-Line = livecast-Version SP Status-Code SP Reason-Phrase
```

*4.1.4 live!*cast-Version Token

*live!*cast uses a "<major>.<minor>" numbering scheme to indicate versions of the protocol. The protocol versioning policy is intended to allow the sender to indicate the format of a message and its capacity for understanding further *live!*cast communication. The current version is "1.0".

```
livecast-Version = "LC" "/" 1*DIGIT "." 1*DIGIT
```

4.1.5 Method Token

The Method token indicates the method to be performed. The method is case-sensitive. Some methods are available in both communication ways, while other methods only applies from terminal to server, and vice-versa. “Terminal-Method” are the methods sent by the terminal to the server. “Server-Method” are the methods sent by the server to the terminal.

```
    Method = General-Method | Terminal-Method | Server-Method
    General-Method   = "NOT"    ; Section Notification (NOT) Request
Message
    Terminal-Method  = "REG"      ; Section Register (REG) Request
Message
                     | "UNREG" ; Section Unregister (UNREG) Request
MessageUnregister (UNREG) Request Message
    Server-Method    = "MSG"   ; Section Message (MSG) Request Message
                     | XXX
```

4.1.6 Status-Code Token

The Status-Code token is a 3-digit integer result code of the attempt to understand and satisfy the request. These codes are fully defined in section XXX. The Reason-Phrase is intended to give a short textual description of the Status-Code. The Status-Code is intended for use by automata and the Reason-Phrase is intended for the human user. The receiver is not required to examine or display the Reason-Phrase.

The first digit of the Status-Code defines the class of response. There are 4 values for the first digit:

- 2xx: Success – The action was successfully received, understood, and accepted
- 3xx: Redirection – Further action must be taken in order to complete the request
- 4xx: Client Error – The request contains bad syntax or cannot be fulfilled
- 5xx: Server Error – The server failed to fulfill an apparently valid request

The individual values of the numeric status codes defined for *live!*cast 1.0, and an example set of corresponding Reason-Phrase’s, are presented below. The reason phrases listed here are only recommendations – they MAY be replaced by local equivalents without affecting the protocol.

Some status codes are available in both communication ways, while other status only applies from terminal to server, and vice-versa. “Terminal-Status” are the status sent back from the terminal, and “Server-Status” are the status codes sent from the server back to the terminal.

```
    Status-Code = General-Status | Terminal-Status | Server-Status

    General-Status  = "200"  ; "OK", Section 200 OK Response Message
                    | "400"  ; "Bad Request", Section 400 Bad Request
Response Message
                    | "500"  ; "Server Error", Section XXX

    Terminal-Status  = "202"    ; "Accepted", Section 202 Accepted
Response Message

    Server-Status   = "301"  ; "Moved Permanently", Section 301 Moved
Permanently Response Message
                    | "302"  ; "Moved Temporarly", Section 302 Moved
Temporarly Response Message
                    | "403"   ; "Forbidden", Section 403 Forbidden
Response Message
```

```
                    | "423"   ; "Interval Too Brief", Section 423
Interval Too Brief Response Message
Reason-Phrase = TEXT
                    | "503"  ; "Service Unavailable", Section XXX
```

4.1.7 Message Header Lines

live!cast headers, which include general-header, request-header and response-header fields, follow the same generic format as tha tiven in Section 3.1 of RFC 822[2]. Each header field consists of a name followed by a colon (":") and a field value. Field names are case-insensitive. Field values MAY be sent with SP characters before or after, and those SP characters MUST be removed before processing the value.

```
    Message-Header  = Field-Name ":" Field-Value CRLF
    Field-Name      = General-Header
                    | Request-Header
                    | Response-Header
    Field-Value     = *SP TEXT *SP
    General-Header  = "seq"  ; Section Sequence (seq) Header
                    | XXX
    Request-Header  = "ch"  ; Section Channels (ch) Header
                    | "lm"  ; Section Last-Message (lm) Header
                    | "f"   ; Section From (f) Header
                    | "aid" ; Section AreaID (aid) Header
                    | "cid" ; Section CellID (cid) Header
                    | "mc"  ; Section Mobile-Code (mc) Header
                    | "ua"  ; Section User-Agent (ua) Header
                    | "cn"  ; Section Channel Number (cn) Header
                    | "sn"  ; Section Serial Number (sn) Header
                    | "dc"  ; Section Data-Coding (dc) Header
                    | "l"   ; Section Content-Length (l) Header
                    | "dr"  ; Section Deliver-Report (dr) Header
Response-Header = "lo"  ; Section Location (lo) Header
                    | "ra"  ; Section Repeat-After (ra) Header
                    | XXX
```

4.1.8 Message Body

4.1.9 The Message-Body (if any) is used to carry the entity-body associated with the request or response. Message-Body format is defined in the Section Message Size

The size of a message received in the UDP payload, or after concatenating all CB pages and before any 7bit decoding, MUST be a multiple of 82 octets, as there's no header in CB to define the real size of a page.

4.1.10 Message Padding

The message MUST be padded with spaces in the end (before 7bit encoding if it's a 7bit message) by the sender to make the final packet a multiple of 82 octets.
The message MUST be stripped from spaces in the end (after 7bit decoding if it's a 7bit message) by the receiver.
Decoding 7bit .

```
    Message-body = *(OCTET)
```

4.2 Headers

4.2.1 Sequence (seq) Header

"Sequence (seq)" header is an integer used to correlate Request with Response messages. Terminal originated Request sequence MUST be independent of server originated Request sequence. A Response MUST include the same sequence value as the corresponding Request.
It is RECOMMENDED to use an incremental integer wrapping around at 65535, and restarting everytime UDP is connected.

```
seq-Header-Value = 1*5DIGIT
```

"Sequence (seq)" header examples:

```
Seq: 12345
```

4.2.2 Channels (ch) Header

"Channels (ch)" header contains a list of channels numbers separated by commas. A range of channels can be used by separating the first and last channel with a dash. "Channel" is defined in Section Channel Format.

```
ch-Header-Value = 1#Channeldef
Channeldef      = *SP Channel | Channel-Range *SP
Channel-Range   = Channel "-" Channel
```

"Channels (ch)" header examples:

```
ch: 1
ch: 1,3,5,7
ch: 1,3,5-9,11
```

4.2.3 Last-Message (lm) Header

"Last-Message (lm)" header contains the date of the last received *live!*cast message. This date will be used by the server to decide which messages will be sent back to the terminal.
"Date" is defined in Section **Error! Reference source not found.**.

```
lm-Header-Value = Date
```

"Last-Message (lm)" header examples:

```
lm: 20040101120000
lm: 20040101130000+01
```

4.2.4 From (f) Header

"From (f)" header contains a string that might identify the terminal or the user. It MAY be the user's MSISDN, the terminal's IMEI, or some other unique token.

```
f-Header-Value = TEXT
```

"From (f)" header examples:

```
f: +491000000000
f: 54F5C78B79F56A57
```

4.2.5 AreaID (aid) Header

"AreaID (aid)" header contains the network Area ID as detected by the device.

```
aid-Header-Value = TEXT
```

"AreaID (aid)" header examples:

```
aid: 144
```

4.2.6 CellID (cid) Header

"CellID (cid)" header contains the network Cell ID as detected by the device.

```
cid-Header-Value = TEXT
```

"CellID (cid)" header examples:

```
cid: 28340
```

4.2.7 Mobile-Code (mc) Header

"Mobile-Code (mc)" header contains the Mobile Country Code (Country ID) and Mobile Network Code (Operator ID), separated by a slash character.
This header SHOULD be omited when the device is in its home network.

```
mc-Header-Value = mcc "/" mnc
mcc             = 3DIGIT
mnc             = 2DIGIT
```

"Mobile-Code (mc)" header examples:

```
mc: 262/02
mc: 268/01
```

4.2.8 User-Agent (ua) Header

"User-Agent (ua)" header contains a string that identifies the *live!*cast software, terminal brand and model, or other relevant information. This header SHOULD be similar to a browser User-Agent and MAY even be the same.

```
ua-Header-Value = TEXT
```

"User-Agent (ua)" header examples:

```
ua: live!cast/1.0 Platform/x.y OS/x.y Vendor/x.y
```

4.2.9 Channel Number (cn) Header

"Channel Number (cn)" header contains an integer representing a channel number relative to the 0xA000 band.

```
cn-Header-Value = Channel
```

"Channel Number (cn)" header examples:

```
cn: 1
```

4.2.10 Serial Number (sn) Header

"Serial Number (sn)" header contains an integer between 0 and 65535 as defined in Section Serial Number. This value is in the same format as CB's Serial Number and MUST be processed used in the same way as CB to detect duplicated messages.

```
sn-Header-Value = Serial-Number
```

"Serial Number (sn)" header examples:

```
sn: 16384
```

4.2.11 Data-Coding (dc) Header

"Data-Coding (dc)" header contains an integer between 0 and 255 as defined in Section Data Coding Format.

```
dc-Header-Value = Data-Coding
```

"Data-Coding (dc)" header examples:

```
dc: 15
dc: 244
```

4.2.12 Content-Length (l) Header

"Content-Length (l)" header contains the number of bytes present in the body payload. If this value is different from the real payload size, the message MUST be discarded. As the payload is the same as in CB, a message with a value greater than 1230 MUST be ignored.

```
l-Header-Value = *DIGIT
```

"Content-Length (l)" header examples:

```
l: 123
```

4.2.13 Deliver-Report (dr) Header

"Deliver-Report (dr)" header contains a boolean that informs the terminal that a Deliver Report Response message MUST be returned. If the header is ommited, the value False MUST be assumed.

```
dr-Header-Value = BOOLEAN
```

"Deliver-Report (dr)" header examples:

```
dr: True
dr: 1
```

4.2.14 Location (lo) Header

"Location (lo)" header contains an network location consisting of an IP address and an optional port. If the port is omited, the terminal MUST use the same port of the default server settings.

```
lo-Header-Value = IP [ ":" PORT ]
IP              = *DIGIT "." *DIGIT "." *DIGIT "." *DIGIT
PORT            = *DIGIT
```

"Location (lo)" header examples:

```
lo: 1.2.3.4
lo: 1.2.3.4:5678
```

4.2.15 Repeat-After (ra) Header

"Repeat After (ra)" header contains an integer value representing an amount of seconds to wait before repeating something.

```
ra-Header-Value = *DIGIT
```

"Repeat After (ra)" header examples:

```
ra: 3600
```

4.3 Messages Requests

4.3.1 Terminal Request Message

When the terminal sends a Request packet and no response is received after some seconds (RECOMMENDED 5 seconds), the terminal SHOULD retry the packet some times (RECOMMENDED 3). The terminal MUST NOT retry indefinidely.
The same procedure SHOULD be used when receiving a 5xx status code, where the amount of retries SHOULD be the same, but the amount of seconds MUST be bigger.

4.3.2 Register (REG) Request Message

A "Register" request message is to be used by the terminal to register to *live!*cast service into the server.

```
Method = "REG"
Mandatory Headers = "seq", "ch"
```

```
Optional Headers  = "lm", "f", "aid", "cid", "mn", "ua"
Body Payload      = None
```

Message example:

```
REG livecast LC/1.0
seq: 1
ch: 1,3,5,20-30,45
lm: 20040101120000+01
f: +491000000000
aid: 144
cid: 28340
mn: 262/02
ua: live!cast/1.0 Platform/x.y OS/x.y Vendor/x.y
```

Expected Responses:

```
Response Status = "200", "301", "302", "400", "403", "423", "500"
```

4.3.3 Unregister (UNREG) Request Message

An "Unregister" request message is to be used by the terminal to unregister from the *live!*cast service.

```
Method = "UNREG"
Mandatory Headers = "seq"
Optional Headers  = None
Body Payload      = None
```

Message example:

```
UNREG livecast LC/1.0
seq: 1
```

Expected Responses:

```
Response Status = "200", "400", "500"
```

4.3.4 Message (MSG) Request Message

A "Message (MSG)" request message is the message type used to deliver content from server to the terminal. It contains the same body payload as after concatenating all received pages in CB (but before 7bit decoding) and a set of headers containing the same values as the ones in CB headers.

```
Method = "MSG"
Mandatory Headers = "seq", "cn", "sn", "dc", "l"
Optional Headers  = "dr"
Body Payload      = live!cast formatted content
```

Message example:

```
MSG livecast LC/1.0
seq: 1
cn: 0
sn: 16384
dc: 15
l: 82
dr: true

<7bit encoded markup snippet>
```

Expected Responses:

```
Response Status = None, "202"
```

4.3.5 Notification (NOT) Request Message

A “Notification (NOT)” request message is a control message used to inform the other party about something, or to request some control information. The type of notification is present in the “not” header.

```
Method = "NOT"
Mandatory Headers = "seq", "not"
Optional Headers  = "aid", "cid"
Body Payload      = None
```

Message example:

```
NOT livecast LC/1.0
seq: 1
not: 100
aid: 144
cid: 28340
```

Expected Responses:

```
Response Status = None, "200"
```

4.3.6 Notification (NOT) Request Message - Keepalive

A Notification: Keepalive request message MAY be used to probe the other party and check if the connection is still available. The terminal SHOULD use them anytime it wants but its usage MUST not affect battery life.

If the terminal supports Network Location headers, it MUST use this request to inform the server about changed Area or Cell by using the optional headers “aid” and “cid”.

If the terminal receives those headers, it MUST ignore their values and MUST reply back to the Keepalive packet.

When the terminal sends this message, the response code from the server MUST be processed like in the REG message if it is a 4xx or a 5xx. A response code like 3xx MUST be ignored.

Message example:

```
NOT livecast LC/1.0
seq: 1
not: 100

NOT livecast LC/1.0
seq: 1
not: 100
aid: 144
cid: 28340
```

Expected Responses:

```
Response Status = "200"
```

4.3.7 Notification (NOT) Request Message – No More Messages

A Notification: No More Messages request message will be used by the server to inform the client that all old messages with date before the current time have been sent.

The terminal SHOULD NOT respond to this message.

The terminal SHOULD use this message to optimize the refreshing of the messages in ticker or navigation mode view.

Message example:

```
NOT livecast LC/1.0
seq: 1
```

```
not: 100

NOT livecast LC/1.0
seq: 1
not: 100
aid: 144
cid: 28340
```

Expected Responses:

```
Response Status = "200"
```

4.4 Messages Responses

4.4.1 200 OK Response Message

A "200 OK" MUST be used when the corresponding request was understood, accepted and processed.

```
Status = 200
Mandatory Headers = "seq"
Optional Headers  = None
Body Payload      = None
```

Message example:

```
LC/1.0 200 OK
seq: 1
```

4.4.2 202 Accepted Response Message

A "202 Accepted" MUST be used to confirm that a message was received by the terminal when the Delivery-Report header is defined as true in the received Message Request.

```
Status = 202
Mandatory Headers = "seq"
Optional Headers  = None
Body Payload      = None
```

Message example:

```
LC/1.0 202 Accepted
seq: 1
```

4.4.3 301 Moved Permanently Response Message

A "301 Moved Permanently" MAY be used by the server to tell the terminal to register again into a different server. The terminal MUST use the new location until the terminal or *live!*cast is restarted, or the transaction fails with the new server location.

```
Status = 301
Mandatory Headers = "seq", "lo"
Optional Headers  = None
Body Payload      = None
```

Message example:

```
LC/1.0 301 Moved Permanently
seq: 1
lo: 1.2.3.4

LC/1.0 301 Moved Permanently
seq: 1
lo: 1.2.3.4:5678
```

4.4.4 302 Moved Temporarly Response Message

A “302 Moved Temporarly” MAY be used by the server to tell the terminal to register again into a different server. The terminal MUST use the new location only once and use the default server again in the next register transaction.

```
Status = 302
Mandatory Headers = “seq”, “lo”
Optional Headers  = None
Body Payload      = None
```

Message example:

```
LC/1.0 302 Moved Temporarly
seq: 1
lo: 1.2.3.4

LC/1.0 302 Moved Temporarly
seq: 1
lo: 1.2.3.4:5678
```

4.4.5 400 Bad Request Response Message

A “400 Bad Request” MUST be used when the corresponding request was not understood.

The terminal (and the server) MUST NOT retry the message request after receiving a 400 Bad Request.

```
Status = 400
Mandatory Headers = “seq”
Optional Headers  = None
Body Payload      = None
```

Message example:

```
LC/1.0 400 Bad Request
seq: 1
```

4.4.6 403 Forbidden Response Message

A “403 Forbidden” MAY be used by the server to inform the terminal that the request was not accepted and the terminal MUST NOT try again until the terminal or *live!*cast is restarted.

```
Status = 403
Mandatory Headers = “seq”
Optional Headers  = None
Body Payload      = None
```

Message example:

```
LC/1.0 403 Forbidden
seq: 1
```

4.4.7 423 Interval Too Brief Response Message

A “423 Interval Too Brief” MAY be used by the server to inform the terminal that the request was not accepted and the terminal MUST NOT try again until the amount of seconds defined in the “Repeat-After (ra)” header passes, or the terminal or *live!*cast is restarted.

```
Status = 423
Mandatory Headers = “seq”, “ra”
Optional Headers  = None
Body Payload      = None
```

Message example:

```
LC/1.0 423 Interval Too Brief
seq: 1
ra: 3600
```

4.4.8 500 Server Error Message

A “500 Server Error” MUST be used when something is uttery wrong and a response cannot be fulfilled. The other party MAY retry the packet but MUST NOT retry more than three times.

```
Status = 500
Mandatory Headers = “seq”
Optional Headers  = None
Body Payload      = None
```

Message example:

```
LC/1.0 500 Server Error
seq: 1
```

4.4.9 503 Service Unavailable Message

A “503 Service Unavailable” MUST be used when something known is wrong and the response cannot be fulfilled. The other party SHOULD NOT retry.

```
Status = 503
Mandatory Headers = “seq”
Optional Headers  = None
Body Payload      = None
```

Message example:

```
LC/1.0 503 Service Unavailable
seq: 1
```

5 Cell Broadcast

5.1.1 CB Page Parameter Format

Page Parameter value is defined in the 3GPP 23.038 XXXLINKXXX specs.
*live!*cast SHOULD store each received page in the *live!*cast queue and keep the incomplete pages until the message expires due to the FIFO logic. Alternatively, it SHOULD keep incomplete pages at least 10 minutes.

6 Online Channel Selection

The Channel Selection service is realized as an regular online HTTP/Web application, with three enhancements:

- When the user starts the Channel Selection Service, the browser is launched with a specific and automatic generated URL as specified in Section XXX.
- The browser interface MUST be as similar as possible to an offline session, thus similar to the device’s menus, which means a specific set of menus and softkeys, and maybe a different screen view mode.
- The browser must understand a specific XML format and its respective content-type as defined in Section XXX.

6.1 Channel Selection Request

The browser instance used to start the Channel Selection service must be started with an hardcoded URL and a list of the subscribed channels in the URL’s query string, or in the POST payload with a regular “application/x-url-encoded” content-type.

```
      Query          = [ Channels-Arg *( "&" Channels-Arg) ]
      Channels-Arg   = Channels-Range | Channel-Arg
      Channel-Arg      = "c" Channel "=on"   ; defined in Section
Channel Format
      Channels-Range = "c" Start-Channel "=" Start-Channel "-" End-
Channel
      Start-Channel  = Channel
      End-Channel    = Channel
```

The list of channels SHOULD be sent in the Query-String field of a GET request. If the request line is too big (e.g. bigger than 1024 bytes), the client SHOULD send the query as a "application/x-url-encoded" POST request.

When doing a POST request, if the response is a 3xx, the client MUST POST again into the new location URL.

6.2 Channel List Response

When navigating through the Channel Selection service, the user may perform an action that would make the server respond with a special content-type and carrying the newly selected channel list.
This browsing session MUST end when the user cancels it (exit or back button), or when the browser receives the special mime-type as described below.
This content-type MUST be accepted and interpreted in regular browsing outside the Channel Selection service.

```
      Mime-Type = application/vnd.vf.cb-channels
```

A domain whitelist MAY apply and the client MUST ignore the type and inform the user.

The CB XML is defined by range blocks, an optional date and an optional confirmation link.
The range blocks (one or more) defines a range of channels to which the channel list must apply. Any channel in that range but not present in the xml MUST be unsubscribed. Any channel not in those ranges MUST be preserved (keep subscribed or unsubscribed as before).
In the Channel element, name is mandatory but icon is optional and if ommited, the client MUST assume it as "default".
The optional "days" attribute MUST be used to unsubscribe automatically the channel in a certain time calculated by adding the number of days with the "date" element, if existent. If date is omited, the client MUST use the server's HTTP header date. If this header is omitted, the client MUST use the device's current date. The number of days to be added to that date MAY be exactly the number of seconds in the defined amount of days, or the end time MAY be rounded to the 23:59:59 of that day.
If the "confirm" element is present, the client MUST request the defined URL (always absolute) and only if the client manually accepts the channels. The client MUST NOT render the result of this request, and MUST ignore any error response codes.

CB XML DTD.

```
    <!-- CB-Channels DTD -->
    <!ELEMENT cb-channels (range*|date|confirm) >
    <!ELEMENT range (channel*) >
```

```
<!ATTLIST range
    from  CDATA  #REQUIRED
    to    CDATA  #REQUIRED
>
<!ELEMENT channel EMPTY >
<!ATTLIST channel
    number  CDATA  #REQUIRED
    name    CDATA  #REQUIRED
    icon    CDATA  #IMPLIED
    days    CDATA  #IMPLIED
>
<!ELEMENT date EMPTY >
<!ATTLIST date
    now  CDATA  #REQUIRED
>
<!ELEMENT confirm EMPTY >
<!ATTLIST confirm
    href  #PCDATA  #REQUIRED
>
```

CB XML Examples.

```
<?xml version="1.0"?>
<cb-channels>
  <range from="0" to="127">
    <channel number="0" name="Vodafone live!" icon="vodafone" />
    <channel number="1" name="News" icon="news" />
  </range>
  <date now="20040101120000" />
  <confirm href=http://live.vodafone.com/livecast/confirm?id=1234
/>
</cb-channel>

<?xml version="1.0"?>
<cb-channels>
  <range from="3" to="4">
    <channel number="3" name="Football" icon="football" />
    <channel   number="4"   name="World   Cup"   icon="football"
days="30" />
  </range>
  <date now="20040101120000" />
  <confirm href=http://live.vodafone.com/livecast/confirm?id=1234
/>
</cb-channel>
```

7 References

1. RFC 2119, Key words for use in RFCs to indicate requirement levels.
2. RFC 822, Standard for the format of ARPA Internet text messages.
3. US-ASCII. Coded Character Set - 7-Bit American Standard Code for Information Interchange. Standard ANSI X3.4-1986, ANSI, 1986.
4. 3GPP 23.038
5. 3GPP 23.041, Technical Realization of Cell Broadcast Service (CBS)

7.1 UIntVar

Each octet of the variable length unsigned integer is comprised of a single Continue bit and 7 bit of payload as shown:

XXX <Continue Bit> <7bit payload>

To encode a large unsigned integer, split it into 7-bit fragments and place them in the payloads of multiple octets. The most significant bits are placed in the first octet with the least significant bits ending up in the last octet. All octets MUST set the Continue bit to 1 except the last octet, which MUST set the Continue bit to 0.

Examples:

XXX

0gfe dcba => 0gfe dcba

hgfe dcba (with h=1) => 1000 000h 0gfe dcba

00nm lkji hgfe dcba => 1nml kjih 0gfe dcba

Two octets of UINTVAR can represent a number from 0 to 16383.

```
#include <stdlib.h>

  long uintvar_to_int (char *uintvar) {
      int c, count;
      unsigned int ui = 0;

      if(uintvar == NULL) return -1;
      for (count = 0; count < 5; count++) {
          c = uintvar[count];
          ui = (ui << 7) | (c & 0x7f);
          if (!(c & 0x80))
              return ui;
      }
      return -1;
  }

  int int_to_uintvar(char *uintvar, unsigned int value) {
      unsigned char octets[5];
      int i,j, start;

      if(uintvar == NULL) return -1;
      octets[5] = value & 0x7f;
      value >>= 7;
      for (i = 4; value > 0 && i >= 0; i--) {
          octets[i] = 0x80 | (value & 0x7f);
          value >>= 7;
      }
      start = i + 1;

      for(i=start,j=0;i<=5;i++,j++)
          uintvar[j]=octets[i];
      return j-1; // size of string
  }



  int main(void) {
      unsigned int myint, myint2;
      char myuintvar[6];

      // 0000 lkji  hgfe dcba
      myint = 0x06A9;

      if( int_to_uintvar(myuintvar, myint) == -1 ) {
          printf("Error converting %l", myint);
```

```
        return(1);
    }
    printf("%s\n", myuintvar);
    myint2 = uintvar_to_int(myuintvar);
    printf("\"%d\"->\"%d\"\n", myint, myint2);
}
```